UNISYS

DCP Series

Distributed Communications Processor Operating System (DCP/OS)

Programming Reference Manual

Release Level 5R1

December 1990

Printed in U S America

Priced Item

UP-11540.2-A

Correspondence regarding this publication should be forwarded to Unisys Corporation either by using the Business Reply Mail form at the back of this manual or by addressing remarks directly to Communication Systems Product Information, P.O. Box 64942 MS:WE4A, St Paul, Minnesota, 55164-0942, U.S.A.

UNISYS

Product Information Announcement

○ New Release ○ Revision ● Update ○ New Mail Code

Title:

DCP Series Distributed Communications Processor Operating System (DCP/OS) Programming Reference Manual Level 5R2

This Product Information Announcement presents the release of update B to the *DCP Series Distributed Communications Processor Operating System (DCP/OS) Programming Reference Manual*, Level 5R1 (UP-11540.2).

The Distributed Communications Processor Operating System (DCP/OS) supports a multiuser environment for building, loading, and executing programs. The operating system is part of a total communications environment that uses Distributed Communications Processors (DCP/5,15,25,30,35,40,50, and 55) to implement intelligent network applications.

The DCP/OS level 5R2 Programming Reference Manual provides information necessary to write programs to operate under DCP/OS level 5R2 in a communications network environment.

This update contains information associated with DCP/OS level 5R2, which includes enhancements to the system service calls.

Remove:	Insert:
Existing pages iii through iv	New pages iii through iv
	New pages iv-a through iv-b
Existing pages xvii through xxiii	New pages xvii through xxiii
Existing pages 3-17 through 3-18	New pages 3-17 through 3-18
Existing pages 7-1 through 7-2	New pages 7-1 through 7-2
Existing pages 7-2a through 7-2b	New pages 7-2a through 7-2d
Existing pages 7-11 through 7-12	New pages 7-11 through 7-12
Existing pages 7-12a through 7-12b	New pages 7-12a through 7-12b
Existing pages 7-19 through 7-20	New pages 7-19 through 7-20
Existing pages 7-23 through 7-24	New pages 7-23 through 7-24
Existing pages 7-27 through 7-30	New pages 7-27 through 7-30

Continued

Announcement and attachments: MBWA, AF01, MU99, MU59

Systems: DCP Series
Release: Level 5R2
Part number: UP-11540.2–B

UNISYS

Product Information Announcement

○ New Release ○ Revision ○ Update ○ New Mail Code

Title:

Existing pages 7-35 through 7-38	New pages 7-35 through 7-38
Existing pages 7-38a through 7-38b	New pages 7-38a through 7-38b
Existing pages 7-39 through 7-42	New pages 7-39 through 7-42
Existing pages 7-57 through 7-58	New pages 7-57 through 7-58
Existing pages 7-73 through 7-74	New pages 7-73 through 7-74
Existing pages 7-79 through 7-80	New pages 7-79 through 7-80
Existing pages 7-85 through 7-86	New pages 7-85 through 7-86
Existing pages 7-91 through 7-94	New pages 7-91 through 7-94
	New pages F-1 through F-10
Existing pages Glossary 11 through 14	New pages Glossary 11 through 14

Changes are indicated by vertical bars in the margins of the replacement pages.

To receive the update package alone, order UP-11540.2–B. To receive the complete manual including the update packages, order UP-11540.2.

To order additional copies of this document

- United States customers call Unisys Direct at 1-800-448-1424
- All other customers contact your Unisys Subsidiary Librarian
- Unisys personnel use the Electronic Literature Ordering (ELO) system

Page Status

Page	Issue
iii through iv	B
iv-a	B
iv-b	Blank
v through xiii	Original
xiv	Blank
xv	Original
xvi	A
xxvii through xxi	B
xxii through xxiii	B
xxiv through xxv	B
xxvi	Blank
xxvii	Original
xxviii	Blank
xxix	Original
xxx	Blank
1–1 through 1–10	Original
2–1 through 2–4	Original
3–1 through 3–11	Original
3–12 through 3–13	A
3–14	A
3–15	Original
3–16	A
3–16a	A
3–16b	Blank
3–17 through 3–18	B
4–1 through 4–5	Original
4–6	Blank
5–1	Original
5–2	A
5–3	Original
5–4	A
5–4a through 5–4b	A
5–5 through 5–10	Original
5–11	A

Page	Issue
5–12	Original
5–13 through 5–17	A
5–18 through 5–19	Original
5–20	A
5–21 through 5–26	Original
5–27	A
5–28	Original
5–29 through 5–32	A
5–32a through 5–32c	A
5–32d	Blank
5–33 through 5–41	Original
5–42	Blank
6–1 through 6–40	Original
7–1 through 7–2	B
7–2a through 7–2d	B
7–3 through 7–4	B
7–5	A
7–6 through 7–10	Original
7–11 through 7–12	B
7–12a	B
7–12b	Blank
7—13 through 7–19	Original
7–20	B
7–20a	B
7–20b	Blank
7–21 through 7–23	Original
7–24	B
7–25 through 7–26	Original
7–27	B
7–28	Original
7–29 through 7–30	B
7—30a	B
7–30b	Blank
7–31 through 7–35	Original
7–36 through 7—38	B
7–38a through 7—38b	B
7–39 through 7–41	B
7–42 through 7–56	Original
7–57 through 7–58	B
7–59 through 7–72	Original

Page	Issue
7–73 through 7–74	B
7–75 through 7–78	Original
7–79 through 7–80	B
7–80a	B
7–80b	Blank
7–81 through 7–85	Original
7–86	B
7–87 through 7–90	Original
7–91 through 7–93	B
7–94 through 7–96	A
8–1 through 8–23	Original
8–24	Blank
9–1 through 9–9	Original
9–10	Blank
A–1 through A–3	Original
A–4	A
A–5 through A–7	Original
A–8	A
A–9 through A–15	Original
A–16	Blank
B–1	Original
B–2	Blank
C–1	Original
C–2	Blank
D–1 through D–2	Original
D–3	A
D–4	Original
D–5 through D–6	A
D–7	Original
D–8	Blank
E–1 through E–3	Original
E–4	Blank
F–1 through F–9	B
F–10	Blank
Glossary–1 through 13	Original
Glossary–14	Blank
Bibliography–1	Original
Bibliography–2	Blank
Index–1 through 27	Original
Index–28	Blank

UNISYS

Product Information Announcement

○ New Release ○ Revision ● Update ○ New Mail Code

Title:

DCP Series Distributed Communications Processor Operating System (DCP/OS) Programming Reference Manual Level 5R1

This Product Information Announcement announces the release of update A to the *DCP Series Distributed Communications Processor Operating System (DCP/OS) Programming Reference Manual,* Level 5R1 (UP-11540.2).

The Distributed Communications Processor Operating System (DCP/OS) supports a multiuser environment for building, loading, and executing programs. The operating system is part of a total communications environment that uses Distributed Communications Processors (DCP/5,15,25,30,35,40, and 50) to implement intelligent network applications.

The DCP/OS level 5R1 Programming Reference Manual provides information necessary to write programs to operate under DCP/OS level 5R1 in a communications network environment.

This update contains information associated with DCP/OS level 5R1, which includes the following:

- Enhancements to the profile and logging debug features
- The addition of 15 new service calls to replace the TELSERV module on DCP/30 and DCP/50 for Telcon and program products Contingency handling

To receive the update package alone, order UP-11540.2–A. To receive the complete manual including the update package, order UP-11540.2.

To order additional copies of this document

- United States customers call Unisys Direct at 1-800-228-9224
- All other customers contact your Unisys Subsidiary Librarian
- Unisys personnel use the Electronic Literature Ordering (ELO) system

Announcement only:
MAC, MBW, MBZ, MCZ, MII0, MMZ, MU1G, MU1J, MU1K, MU1L, MU1N, MU1Y, MU59, MV7B, MV7C, MX3, MX5, MX6, MX8, MY1, MY3, MY5, MY6, MY7, M154

Announcement and attachments:
MBWA, AF01, MU99, MU59

System: DCP Series
Release: Level 5R1

Part number: UP-11540.2–A

Page Status

Page	Issue
iii through iv	A
iv-a	A
iv-b	Blank
v through xiii	Original
xiv	Blank
xv	Original
xvi through xxvi	A
xxvii	Original
xxviii	Blank
xxix	Original
xxx	Blank
1–1 through 1–10	Original
2–1 through 2–4	Original
3–1 through 3–11	Original
3–12 through 3–13	A
3–14 through 3–15	Original
3–16	A
3–16a	A
3–16b	Blank
3–17 through 3–18	Original
4–1 through 4–5	Original
4–6	Blank
5–1	Original
5–2	A
5–3	Original
5–4	A
5–4a through 5–4b	A
5–5 through 5–10	Original
5–11	A
5–12	Original
5–13 through 5–17	A
5–18 through 5–19	Original
5–20	A

Page	Issue
5–21 through 5–26	Original
5–27	A
5–28	Original
5–29 through 5–32	A
5–32a through 5–32c	A
5–32d	Blank
5–33 through 5–41	Original
5–42	Blank
6–1 through 6–40	Original
7–1	Original
7–2	A
7–2a	A
7–2b	Blank
7–3	A
7–4	Original
7–5	A
7–6 through 7–16	Original
7–17 through 7–18	A
7–19 through 7–28	Original
7–29 through 7–30	A
7–31 through 7–35	Original
7–36 through 7–38	A
7–38a	Blank
7–38b	A
7–39 through 7–85	Original
7–86	A
7–87 through 7–91	Original
7–92 through 7–96	A
8–1 through 8–23	Original
8–24	Blank
9–1 through 9–9	Original
9–10	Blank
A–1 through A–3	Original
A–4	A
A–5 through A–7	Original
A–8	A
A–9 through A–15	Original
A–16	Blank
B–1	Original
B–2	Blank
C–1	Original

Page	Issue
C–2	Blank
D–1 through D–2	Original
D–3	A
D–4	Original
D–5 through D–6	A
D–7	Original
D–8	Blank
E–1 through E–3	Original
E–4	Blank
Glossary–1 through 13	Original
Glossary–14	Blank
Bibliography–1	Original
Bibliography–2	Blank
Index–1 through 27	Original
Index–28	Blank

UNISYS

DCP Series

Distributed Communications Processor Operating System (DCP/OS)

Programming Reference Manual

Release 4R1

June 1989

Printed in U S America

Priced Item

UP-11540.2

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are furnished only pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the form at the back of this manual, or may be addressed directly to Unisys Corporation, Communication Systems Product Information, P.O. Box 64942 MS:WE4A, St. Paul, Minnesota, 55164-0942, U.S.A.

Page Status

Page	Issue
iii through xiii	Original
xiv	Blank
xv through xxv	Original
xxvi	Blank
xxvii	Original
xxviii	Blank
xxix	Original
xxx	Blank
1–1 through 1–10	Original
2–1 through 2–4	Original
3–1 through 3–18	Original
4–1 through 4–5	Original
4–6	Blank
5–1 through 5–41	Original
5–42	Blank
6–1 through 6–40	Original
7–1 through 7–91	Original
7–92	Blank
8–1 through 8–23	Original
8–24	Blank
9–1 through 9–9	Original
9–10	Blank
A–1 through A–15	Original
A–16	Blank
B–1	Original
B–2	Blank
C–1	Original
C–2	Blank
D–1 through D–7	Original
D–8	Blank
E–1 through E–3	Original
E–4	Blank

Page	Issue
Glossary–1 through 13	Original
Glossary–14	Blank
Bibliography–1	Original
Bibliography–2	Blank
Index–1 through 27	Original
Index–28	Blank

About This Manual

Purpose

This manual describes how to write programs to run under the Distributed Communications Processor Operating System (DCP/OS) on a Distributed Communications Processor (DCP) in a communications networking environment.

The DCP family of communications processors includes the following:

- DCP/5
- DCP/10A
- DCP/15
- DCP/20
- DCP/30
- DCP/40
- DCP/50

The DCP/OS supports a multiuser environment for building, loading, and executing programs. The operating system is part of a total communications environment that uses DCPs to implement intelligent network applications.

Scope

This manual describes the following:

- User programming
- Communications Processor Architecture (CPA) module definitions
- Contingency handling
- System service calls and functions
- Common utility subroutines

udience

This manual is written for programmers and systems analysts.

rerequisites

The DCP/OS philosophy requires no knowledge or experience with any particular programming language. This manual explains and specifies the various system interfaces in the context of assembler programming.

ow to Use This Manual

This manual is a reference document. It is not expected that you will read it from cover to cover. Rather, read the introduction, then refer to the table of contents and the index for information about a particular subject.

rganization

This manual is divided into nine sections and five appendixes.

Section 1: Introduction to DCP/OS Programming

This section describes the programming environment. It outlines the source code, assembly, collection, and linking of user programs, and describes the block structure of the DCP/OS file system.

Section 2: System Message Formats

This section describes standard system messages and system information packets.

Section 3: CPA Module Definition MASM Procedures

This section describes the OS 1100 Meta-Assembler (MASM) procedures that create the Communications Processor Architecture (CPA) entries in user programs.

Section 4: Contingency Handling

This section describes the process of contingency handling for specific port processor (PP) state items, central processor (CP) specification exceptions, inter-program message (IPM) state changes, and throttle level changes.

Section 5: Debug Facilities

This section tells you how to use the debug facilities to debug programs that run under the DCP/OS.

Section 6: MASM Utility Procedures

This section describes optional MASM utility procedures available from the library file, DCPOSEQU, on the release tape.

Section 7: System Service Calls (SVCs)

This section outlines the system service calls (SVCs) available to the user programs. It describes the system service calls by category and provides call, entry, and exit specifications.

Section 8: Common Utility Subroutines

This section describes common utility subroutines and provides call, entry, and exit specifications.

Section 9: Memory Management

This section explains how to apportion and manage system memory for various purposes.

Appendix A: Data Structures

This appendix presents diagrams with field definitions of various packets, blocks, and headers.

Appendix B: Segment Names of Available User System Structures and Code

This appendix lists data structures and code, segment names and field definitions, for programs.

Appendix C: Definition Elements

This appendix lists definition elements with their context and definition.

Appendix D: Service Function Codes

This appendix lists the service function codes in hexadecimal order.

Appendix E: Programming Examples

This appendix provides four programming examples: CP programs, assembly on OS 1100, collection on OS 1100, and build on DCP/OS.

elated Product Information

Additional information is available in the following manuals. Programming reference manuals must be ordered separately.

OS 1100/DCP Series Communications Delivery Software Configuration Guide, Level 4R1 (UP-9957.10)

This guide tells how to configure Communications Delivery software level 4R1 for a data communications network. It also tells you how to reconfigure these software products as your network evolves.

OS 1100/DCP Series Communications Delivery Software Configuration Reference Manual, Level 4R1 (UP-11580.1)

This manual provides reference material for configuring data communications networks for CD level 4R1 software.

OS 1100/DCP Series Communications Delivery Software Installation Guide, Level 4R1 (UP-9956.10)

This manual tells you how to generate, install, and verify Communications Delivery level 4R1 (CD level 4R1) software on an OS 1100 host and its Distributed Communications Processors (DCPs). Generating and installing involves copying the CD level 4R1 software components and related software products from release tapes to mass storage and preparing the software for use with your communications network.

DCP Series Telcon Operations Reference Manual, Level 8R1 (UP-9256.9)

This manual is part of the operations subset of the Communications Delivery library. It is a reference to a full range of options on NMS commands and online configuration commands. It lists online hardware verification operations, RFS commands used to transfer files, hardware instrumentation parameters on the TRON command, general NMS console messages, and CENL console messages.

DCP Series Telcon Operations Guide, Level 8R1 (UP-13431)

This manual is part of the operations subset of the Communications Delivery library. It serves as a guide to Telcon operations. It explains how a DCP network is organized, how to use NMS consoles and commands, how to use Telcon online configuration, how to transfer

files in a DCP network environment, how to turn on instrumentation, how to interpret messages, and how to control console and logged messages.

DCP Series Telcon Internals Programming Reference Manual, Level 8R1 (UP-9255.8-A)

This manual describes the modular structure of Telcon and its relationship to the DCP Operating System. It also includes details on interfaces, registers, queues, and table and message structures that help the programmer integrate site-developed code with Telcon.

DCP Series Telcon Terminal Handler Platform Programming Reference Manual, Level 8R1 (UP-13460)

This manual provides information for writing terminal handler programs for terminals in your network that are not provided by Unisys.

DCP Series Distributed Communications Processor Operating System (DCP/OS) Operations Reference Manual, Level 4R1 (UP-11541.2)

This manual is part of the operations subset of the Communications Delivery library. Its purpose is to familiarize the user with DCP/OS procedures and commands.

DCP Series Implementation Reference Manual, Volume 1, Volume 2, Rev. 1, and Volume 3 (UP-12728)

Unisys Distributed Communications Processors implement Communications Processor Architecture (CPA) in a range of sizes and capacities, and offer the same general selection of line modules.

Volume 1 describes communications processor architecture and summarizes the processor hardware.

Volume 2, Rev. 1 contains information on the line module interfaces and protocols. The revision includes the DCP 802.3 local area network (LAN) line module, its associated line module exchange protocol (LMPX), and the 8441 mass storage subsystem.

Volume 3 contains procedural and reference information on DCP Series offline diagnostics and the maintenance control feature (MCF) for the DCP/50 and DCP/30 models.

OS 1100 Collector Programming Reference Manual, Level 32R2 (UP-8721.5)

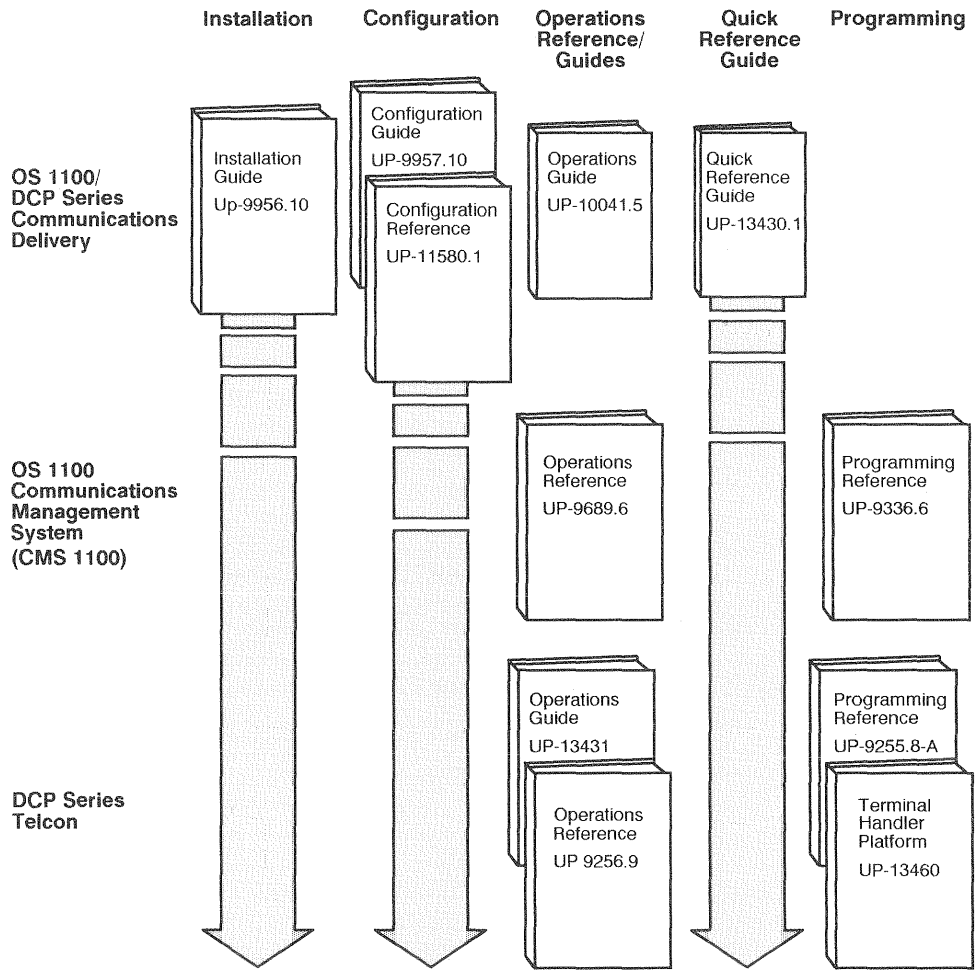
This manual provides detailed instructions on how to use the OS 1100 Collector program. It describes how to include and exclude elements, and how to use banking directives, bank-implied collections, and bank switching. It explains how to specify program parameters and use other program directives. It also discusses Collector-defined symbols and how to use the Collector efficiently.

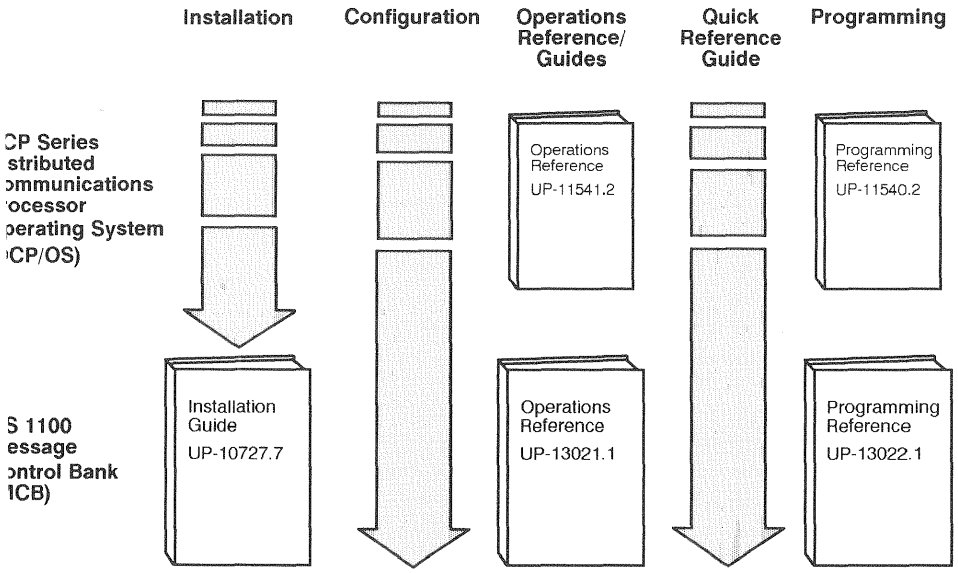
OS 1100 Meta-Assembler (MASM) Programming Reference Manual, Level 5R1 (UP-8453.4)

This reference manual provides programmers and analysts with detailed information about the OS 1100 Meta-Assembler (MASM) level 5R1 processor and language. It includes comprehensive information about this release level of MASM. It is not a tutorial; thus, it does not explain how to write MASM programs.

The Communications Delivery Level 4R1 Library

The following illustration shows how all books in the CD4R1 library relate to one another. Use this illustration to help you look for information in the library.


Installation
Configuration
Operations Reference/ Guides
Quick Reference Guide
Programming
OS 1100/ DCP Series Communications Delivery
OS 1100 Communications Management System (CMS 1100)
DCP Series Telcon
Installation Guide
Up-9956.10
Configuration Guide
UP-9957.10
Configuration Reference
UP-11580.1
Operations Guide
UP-10041.5
Quick Reference Guide
UP-13430.1
Operations Reference
UP-9689.6
Programming Reference
UP-9336.6
Operations Guide
UP-13431
Operations Reference
UP 9256.9
Programming Reference
UP-9255.8-A
Terminal Handler Platform
UP-13460



otation Conventions

The general command format is similar to English language format. The special symbol command format is defined in Table 6-1 (see Section 6). Each command begins with a keyword indicating the action performed.

COMMAND *parameter-string*

where:

COMMAND	the name of a command. Command names are presented in uppercase letters to indicate that you must enter them exactly as shown.
parameter-string	the parameter string for the command. These parameter strings are lowercase letters or italic when the exact input is variable.

- Brackets ([ ]) indicate optional information.

- Braces ({ }) indicate that you must choose one of the items shown.
- Braces within brackets ([{ }]) indicate that you may choose only one of the optional items.
- Names enclosed in angle brackets (< >) describe a class of symbols. These are sometimes called nonterminals. Names not enclosed in angle brackets are called terminal symbols and make up numbers of a class.
- The symbol (::=) means "is defined as".
- An ellipsis (...) means the preceding items can be repeated.
- Lowercase *italic* character strings indicate names you must provide.

Contents

Appendix A. Data Structures

Appendix B. Segment Names of Available User System Structures and Code

Appendix C. Definition Elements

Appendix D. Service Function Codes

Appendix E. Programming Examples

Figures

Tables

Section 1
Introduction to DCP/OS Programming

This section describes the following:

- The DCP/OS programming environment
- The stages for building programs
- The DCP/OS file system

1.1. DCP/OS Programming Environment

A DCP/OS program runs on Communications Processor Architecture (CPA)-defined computers. The DCP/OS is composed of procedures, segments, port processor programs, and queues -- all of which are CPA-defined architectural entities. These structures ensure protection and security. The instructions supported by CPA to manipulate these structures are privileged.

To maintain the protection afforded by CPA, only DCP/OS procedures may run in privileged mode. If other programs need to manipulate certain CPA structures at run time to create segments and extra queues, a system service call (SVC) instruction activates privileged DCP/OS service. See Section 7 of this manual for further information regarding system service calls.

Some basic concepts of the DCP/OS environment are described in 1.1.1 to 1.1.6.

1.1.1. File System

The DCP/OS file system manager provides services to create and use contiguous files on a variety of mass storage media, including diskettes, cartridges, and Winchester disks. During program development, the file system holds omnibus elements (modules), absolute elements (executable programs), and add streams (symbolic elements). Files are named as follows: *qualifier*filename.*

Elements are referenced in general as *qualifier*filename.element*. System files use the special qualifier SYS$.

.1.2. MASM Procedures and Definition Elements

The OS 1100 Meta-Assembler (MASM) processor assembles programs for any target machine by processing each instruction as a MASM procedure. All system-wide definitions, including the assembler instructions, are fully defined in the form of definition elements.

The elements are categorized according to use and context. Appendix C contains a list of the definition elements. The primary definition element is AAWRENCH, which contains the central processor (CP) instruction set, extended instruction definitions, and module definition MASM procedures. See Section 6 for more information on MASM utility procedures.

.1.3. Register Sets

The CPA environment supports separate register sets for process mode and control mode. Both the process mode and control mode sets have two subsets. One subset contains segment descriptor registers (SDRs), and the other subset contains general registers (R0-R15).

Programs must use the process mode register set. You can automatically specify this mode by using the PROCDEF MASM procedure to define a user procedure.

.1.4. Parameter Passing

Parameters may be passed between procedures in registers or in data areas. The user can completely define the register convention. Note, however, that any data area must be in the caller's visibility before the call, and cannot be in the virtual address range X'0000' to X'1FFF' because CPA replaces the first four SDRs on a CALL.

.1.5. Process Control

The maximum number of concurrent tasks or processes is fixed at build time, and defaults to one. The maximum number of tasks may also be specified at assembly time, using the CPADEF MASM procedure. However, tasks themselves are not fixed.

A new task (or process) is created when

- Initial procedure is dispatched (on loading).
- Queue threshold is crossed.
- Service call is invoked to dispatch a procedure.

Each task (process) starts at a given initial procedure which may call (using CALL or SCALL) additional procedures and subroutines. The initial procedure of the task (process) works back up the CALL/RETURN stack through a series of RTN and SRTN instructions. When the initial procedure of the process receives control and issues an RTN instruction, control is then returned to the DCP/OS dispatcher.

Any procedure that services several input queues is made more efficient by using cascaded queues. See the *DCP Series Implementation Reference Manual, Volume 1, Volume 2 Rev.1, and Volume 3* (UP-12728) for a description of cascading.

1.1.6. Exception Handling

The CPA traps all errors in CP programs. All control and errors are transferred to a DCP/OS procedure. This mechanism is termed a forced call.

The CPA also traps port processor (PP) program errors, but the mechanism is different. The current state of the PP program is captured, stored in a buffer, and queued to DCP/OS. See the DCP Series Implementation Reference Manual (UP-12728) for a full description of these CPA mechanisms.

Once DCP/OS detects a CP or PP fault, the action taken depends upon the mode in which the program was running.

Programs may specify a contingency handling procedure for specific errors. This feature enables programs to survive trivial errors.

In interactive mode, you may automatically invoke full-screen debug when any program faults. In batch mode, the program is aborted. The program also aborts if no auto-debug trap is set. A dump is taken if it was previously specified. See the @CRASH command in the *DCP Series, Distributed Communications Processor Operating System (DCP/OS) Operations Reference Manual* (UP-11541).

1.7. HELP Facilities

Most DCP/OS utilities provide HELP facilities. The DCP/OS reserves the H option on the command line for help. It is recommended that programs provide HELP facilities and indicate on the screen how to access help by displaying a prompt similar to the following:

```
Please enter command (H=help) -
```

By displaying HELP screens only upon user requests (through prompts), you can avoid perpetual bombardment with menus containing myriad help options.

.2. Building Programs

Figure 1-1 illustrates the stages of program building. The current software availability dictates the following split of the development environment:

- OS 1100 - Editing (Source Code), Assembly, Collection, Building

  The OS 1100 environment Remote Symbiont Interface (RSI) DEMAND is specified in the OS 1100 documentation. The DCP/OS environment is compatible with the OS 1100 environment, and provides a simple facility to copy elements between the two.

- DCP - Building, Debugging

  The DCP/OS is a multiuser system. A user can invoke a single or multitasking program with complete protection from other users.

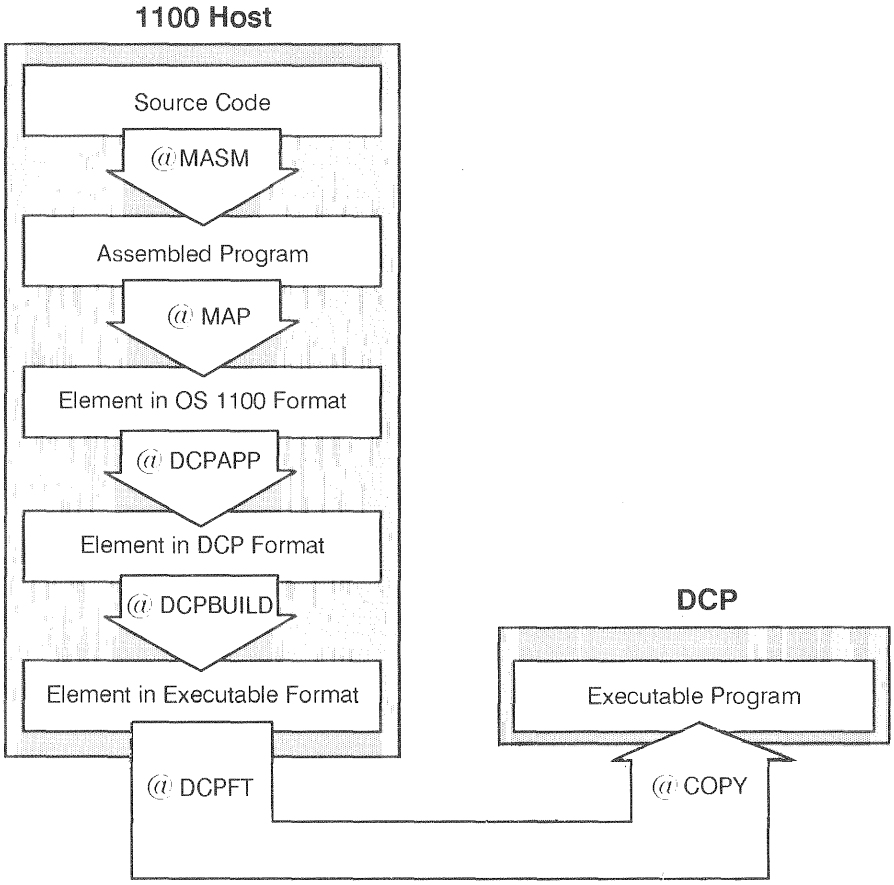


Figure 1-1. Stages in Building Programs

1.2.1. Source Code

For communications processor programs, the host program file maintains the source code in symbolic elements. The typical assembler program begins with $INCLUDE statements, followed by MASM procedures defining the procedure-related structures (GPL, PST, QL, and LA). These MASM procedures are GPLDEF , PSTDEF, QUEDEF, and LNKDEF.

The main body of code then follows. The segments of code and data generated by the assembly are defined by system segment table (SST) definition MASM procedures (SSTDEF) at the end of the element.

The assembler program ends with any CPA procedure definitions (PROCDEF) and the MASM terminator ($END). For a full description of the MASM facilities, refer to the *OS 1100 Meta-Assembler (MASM) Programming Reference Manual* (UP-8453).

The port processor (PP) programs follow the same principle, except that the procedures are defined by IOPDEF MASM procedures.

See Appendix E for program examples.

2.2. Assembly

Programs are assembled by the MASM assembler on a host system. The required definition elements are specified in the $INCLUDE statements at the beginning of the program. These elements are user-dependent.

The CP programs are initialized with the WRENCH MASM procedure, as follows:

```
$INCLUDE 'AAWRENCH'
WRENCH
```

The PP programs are initialized with the WREC MASM procedure, as follows:

```
$INCLUDE 'AAWRENCH'
$INCLUDE 'AAPPROC' WREC
```

The MASM facility creates OS 1100 relocatable elements.

2.3. Collection

The collection process has two stages:

- @MAP—collects relocatable elements
- @DCPAPP—converts OS 1100 format into DCP format

The @MAP (collector) utility is specified in the *OS 1100 Collector Programming Reference Manual* (UP-8721). The @MAP utility produces an intermediate element of modules in 36-bit format.

This element must be post-processed (by DCPAPP) into the DCP format. The output to DCPAPP is called the module library file (MLF).

The MASM output is organized under separate location counters (LCs). The BUILDER preambles that describe the various CPA modules, such as segments and queues, appear under special LCs 60 to 63 as follows:

```
special LCs63--'DY' header
62--PROCDEF,ENTDEF,IOPDEF,QUEUE preambles
61--SSTDEF preambles
60--Other data associated with LC 62 preambles. GPL, PST
etc.
```

It is important that the Collector include segments (SSTDEFs) in the same order as stated in the assembly. You can define location counter sequence and use a simple IN directive to @MAP. Otherwise you must explicitly request the chosen sequence with multiple IN statements at MAP time.

1.2.4. Build

The build process can be achieved on the OS 1100 host with the @DCPBUILD utility program or on the DCP with the @BUILD utility program. The @DCPBUILD utility program is faster and provides greater mass storage capabilities. In this manual, when the DCP/OS utility program @BUILD is not specified, the host utility program @DCPBUILD will be assumed.

To create an executable program, independent components of a program are linked using the @DCPBUILD utility program. The @DCPBUILD utility program resolves intercomponent CPA entity references and automatically generates a dictionary containing the names of these CPA entities. Creating executable programs from independently created components allows and encourages a structured program discipline. Refer to Appendix A for more information on formatting a dictionary entry.

The required run-time tables for the absolute program are in the header block. These may be overridden at execution time by parameters on the call line. Refer to the DCP/OS Operations Reference Manual (UP-11541) for details on the @DCPBUILD and @BUILD utility programs.

The module library file (MLF) input elements contain definitions of the CPA structures (such as queues and segments) and satisfies all cross-references by name. For example, a procedure structure (defined by PROCDEF) contains references to the initial segments that are defined in segment structures (defined by SSTDEF). The @DCPBUILD utility program satisfies these cross-references when building a program.

2.5. Host/DCP Copy

The host utility program DCPFT is used to pair with DCP/OS across a channel. The DCP/OS command @COPY can then be used to copy files and elements from host to DCP and vice versa. See the DCP Series, DCP/OS Operations Reference Manual (UP-11541) for details.

2.6. Debugging

The DCP/OS has a powerful debug facility. It provides a facility for looking at the contents of memory, changing the contents of memory, trapping errors, displaying CPA structure, and a novel debugging methodology. Debug commands are defined in Section 5.

.3. DCP/OS File System

The DCP/OS file system supports contiguous files accessed by logical block numbers. To copy files from one device to another, the logical block size is fixed to facilitate internal block chaining independent of device characteristics.

The file manager regards each disk file as a logically contiguous number of fixed-length blocks. This block size is fixed at 256 bytes for disk files, regardless of the physical device sector sizes. (For example, cartridge disk sector size is 256 bytes, but diskette sector size is 128 bytes.)

There are two basic file formats: data file and program file. The data file format is user-defined. A program file, however, is defined by the DCP/OS. The program file structure supports subfiles called elements.

Elements are logically contiguous sets of blocks. The program file includes one or more TOC (table of contents) blocks. These TOC blocks are used by the file system and are transparent to the user program.

The supported element types are as follows:

- Omnibus (user defined)
- Symbolic (see 1.3.2.)
- Absolute (see 1.3.3.)

1.3.1. Naming Convention

The DCP/OS uses the *qualifier*filename.element* naming convention. The maximum character length of the qualifier is six characters, the file name is eight characters, and the element name is 12 characters. All files have qualifiers.

The DCP/OS uses various files for system functions. These file names all begin with SYS and have the file qualifier SYS$. Users should avoid naming files in this fashion.

For example, the following filenames designate system files:

- SYS$*SYSLIB (system library)
- SYS$*SYSJOB (system runstreams)
- SYS$*SYSLMC (line module code) line module

1.3.2. Symbolic Element

A symbolic element contains variable-length text records terminated by end-of-file sentinels. Each record begins with a 16-bit count that specifies the number of data bytes in the record. The ASCII data bytes then follow. Each element ends with an end-of-file terminator, which is a record byte count of X'FFFF'.

1.3.3. Absolute Element (Executable Program)

The absolute element (ABS) shown in Figure 1-2 is described as follows:

- The first block of each ABS is a header record that describes the executable system contained in following blocks. See Appendix A.
- The header fields specify such characteristics as the size of the SST, procedure table (PT), and the initial procedure number (IPN).

- The arrangement of segments and other CPA structures in the element is not fixed, but depends on the order in which modules are presented to the build process.
- The SST starts at a fixed block number in the element and is used as a directory for the rest of the system.

See Appendix A for the layout of the header block.

Block	
0	Header Block
1	First SST segment
10	
	Other Segments
	- PT, PPPT, QT, GPL, PST, QL
	- Queue headers
	- Code and data segments
	- Dictionary segments
>end<	

Figure 1-2. Absolute Element Layout

.3.4. Dump Files

The dump files are in normal data file format, but are created by the system. Dump files have a defined internal structure, mapped by the header block described in Appendix A. The dump file can be analyzed using the IDUMP or DMPI commands, or transferred to an OS-1100 host and formatted by DCPFOR/DCPDUMP. See the DCP Series, DCP/OS Operations Reference Manual (UP-11541) for details. Figure 1-3 shows the dump file layout.

Block	
0	Header Record
1	Dump data

Figure 1-3. Dump File Layout

Section 2
System Message Formats

This section describes the following:

- The DCP/OS system messages
- The DCP/OS system information packet structure

2.1. Standard Message

The system supports a program-user dialog with system service calls (such as READ$ and PRINT$). Data is exchanged as discrete messages in CPA message control tables (MCT) and buffers.

The system message format is a DCP/OS convention that describes the message by message type, length, and position. The message field offsets and message types are defined in the definition element AARUNDEF. Fields used within the message header vary, depending on the message type. The message header format is shown in Figure 2-1.

WORD	CONTENT	
	15 0	
0	MH$MTYP	message type
1	MH$CNSN	workstation ordinal number
2	MH$DBO	data byte offset
3	MH$DBC	data byte count
4	MH$ID	run-id (or zero)
5	MH$SUPPA	supplementary word A
6	MH$SUPPB	supplementary word B
7	MH$SUPPC	supplementary word C

Figure 2-1. Message Header Format

The currently defined user message types are

MH$MCO	Output message
MH$MCOT	Output message transparent (no scroll)
MH$MCI	Input message
MH$MCIT	Input message transparent (escape mode)
MH$MCIF	Input function key (logical 1-65)
MH$MREAD	Input data
MH$MEOF	End-of-file to read request
MH$MINFOR	INFOR$ or KEYWORD structured message
MH$MRDRQT	read

.2. System Information Packet (INFOR$)

The INFOR$ packet is an encoded version of a program execution statement. It is created by the system manager when it detects the traffic as control mode data. It is also created by the run manager on data read from a batch READ$ file if the run is in control mode. The input to an INFOR$ packet is data similar to the program call notation structure used by the OS 1100. After the packet is created, it is saved for reference by the user and is accessed by invoking the run service READ$. Once the user has acquired the packet, the packet becomes the user's responsibility.

The INFOR$ packet output is a structure using control bytes and length bytes. You can reference this structure directly, or you may use the string handling subroutines to retrieve pertinent information about the input. The packet is built with a standard MH$ header with a message type of MH$MINFOR. The originating work station or run unit is stored in the header. The packet attributes are stored within the supplementary words MH$SUPPA through MH$SUPPC. This encompasses a byte containing the highest specification created (IF$LSPEC), whether it has OS 1100 type file references in the packet (IF$FGN), or whether it is a nonstandard INFOR$ structure (IF$IPF). If the structure is nonstandard (the IF$IPF bit is set), it means that a user has parsed an input by calling the S$CIPF utility subroutine using SCALL. This packet then contains the KEYWORD=(PARM,PARM) structures and does not reflect standard file notation data.

If the packet is the standard INFOR$ type, then MH$SUPPA and MH$SUPPC contain a bit map of the options supplied on specification zero of the input. You can reference the option bits by the names IF$AOPT through IF$ZOPT inclusively.

To reference a standard INFOR$ structure, use the specification number and field type. The typical format of an INFOR$ input is

```
@QUAL*FILE.ELEMENT,OPTIONS Q*F/R/W.E...
```

The read and write keys (fields R and W) are not processed by the file manager on local files. The defaults defined in OS 1100 file notation apply. OS 1100 file cycles, versions, and element cycles are not supported.

The following are the currently supported control bytes:

IF$*b*OPT	*b* is an option of A through Z. This is a bit.
IF$SPEC	Data is a specification number.
IF$CTYP	Data is a keyword card type.
IF$NAME	Data is a keyword card name.
IF$KEYW	Data is a keyword.
IF$PARM	Data is keyword parameter.
IF$VOL	Data is a host or volume name.
IF$QUAL	Data is a file qualifier.
IF$READK	Data is an 1100 file read key.
IF$WRITK	Data is an 1100 file write key.
IF$FILE	Data is a file name.
IF$ELT	Data is an element name.
IF$ASCII	Data is an ASCII string.
IF$FOE	Data is a file or element.
IF$END	This is the END of the packet.
IF$SLNT1	Data following first slash (/).
IF$SLNT2	Data following second slash (/).
IF$SLNT3	Data following third slash (/).

Figure 2-2 describes the INFOR$ packet structure of the following example:

```
@QUAL*MYFILE.MYELT,ABC MYFILE1.,MYELT2
```

WORD	CONTENT	
	15 0	
0	MH$MINFOR	(message type)
1	n	(workstation ordinal number)
2	x'0010'	(data byte offset)
3	x'002D'	(data byte count)
4	n	(run-id)
5	0 (no special flag)	2 (highest spec number)
6	IF$AOPT,IF$BOPT,IF$COPT	
7	(option bits set)	
8	IF$SPEC	0 (spec number)
9	IF$QUAL	4 (length)
A	'Q'	'U'
B	'A'	'L'
C	IF$FILE	6 (length)
D	'M'	'Y'
E	'F'	'I'
F	'L'	'E'
10	IF$ELT	5 (length)
11	'M'	'Y'
12	'E'	'L'
13	'T'	IF$SPEC
14	1 (spec 1)	IF$FILE
15	7 (length)	'M'
16	'Y'	'F'
17	'I'	'L'
18	'E'	'1'
19	IF$SPEC	2 (spec number)
1A	IF$ELT	6 (length)
1B	'M'	'Y'
1C	'E'	'L'
1D	'T'	'2'
1E	IF$END	0

Figure 2-2. System Information Packet

Section 3
CPA Module Definition MASM Procedures

This section describes the following:

- The MASM procedures that create the CPA entities

These module definition MASM procedures (defined in AAWRENCH) produce the information required to subsequently build a program. Refer to the @BUILD utility program in the DCP Series, DCP/OS Operations Reference Manual (UP-11541) for more information on the build program.

Each of the following MASM procedures generates either a full CPA entity module (such as SSTDEF -> segment) or part of a module (such as GPLDEF -> procedure). The difference is transparent, because each MASM procedure is used for a specific purpose within the program.

3.1. ATXREF - Sharing Lists

The GPL, PST, QL, and LA lists are normally constrained to use within one assembly.

The ATXREF MASM procedure allows sharing of the GPL, PST, QL, and LA lists between assemblies. The ATXREF MASM procedure does not allow forward references. A referenced table must be defined before build time. See the PROCDEF MASM procedure definition.

Format

ATXREF[,'< *export-reference*>'] < *import/export list*>

Parameters

< *export-reference*> A 1- to 4-character name.

<import/export list> ::='<import>'/
'<export>'/
'<import>'<import/export list>

'<export>'<import/export list>
and
<export>::= GPL / QUE / PST / LNK
<import>::= <export>=<export reference>

Note: *Blanks are not allowed within strings.*

Example

```
ATXREF, 'NEWA' 'GPL','LNK','QUE'
ATXREF, 'QUE=NEWA','LNK=PQR'
ATXREF, 'PQR' 'LNK','GPL=NEWA'
```

Function

The ATXREF MASM procedure generates import and export capabilities for selected architectural MASM procedures.

Note: *The ATXREF MASM procedure should be coded before any architectural MASM procedures.*

An export reference is a name used to label architectural tables (AT) and prefix their entry points. It is the name that a potential sharer of an AT uses to specify where it should be picked up.

Architectural tables that can be exported are the GPL, PST, QL, and LA. To export one of these, code GPL, PST, QUE, or LNK in the ATXREF call line, along with an export name.

To import references, that is, to pick up CFACCS from a GPL, the importer must encode the relevant GPL. The entries, in any order, require encoding. This encoding does not need to correspond exactly to that encoded by the exporter. Similarly, only the names of the references are necessary. Add any subfields for clarity.

If no references are needed within an assembly, no corresponding architectural MASM procedure need be coded by an importer. For example, the table and not its contents is important (as is always the case with the LA), so no corresponding MASM procedure need be coded by the importer.

To import an AT, the relevant AT identifier is equated to the export reference of the exporter, for example, 'GPL=ABC'. Up to four tables may be simultaneously exported or imported, but a table marked for export cannot be imported.

Always include the MASM procedure because no AT is exported unless you invoke the relevant MASM procedure. All the affected MASM procedures (PROCDEF, GPLDEF, LNKDEF, and PSTDEF) are unchanged in their external form.

Example of ATXREF Exporting GPL and QL

```
          $INCLUDE  'AAWRENCH'
          WRENCH
          ATXREF,'ABC' 'GPL','QUE'     . ABC is export reference
                                       . GPL is marked for export
                                       . QL  is marked for export
          GPLDEF    'SICOMP';          . Define GPL as normal
                    'CFACCS';          .
                       etc
          QUEDEF,20 'Q1',G,P,A;        . Define QL as normal
                    'Q2',P,A;
                       etc
          PSTDEF                       . PST required but not exported
ENTRY
          .....
          LOADC  R1,Q2                 . Use current entry references
          .....
          CALL   CFACCS                . Ditto
          .....
S1        SSTDEF,1                     . Still need old label (1)
                                       . for PSTDEF
TEST      PROCDEF,1 'ENTRY'            . Ditto
          $END
```

The following external entry points (EEPs) are created:

```
ABCSUABRT  $EQU  SUABRT
ABCCFACCS  $EQU  CFACCS
.....
ABCQ1      $EQU  Q1
ABCQ2      $EQU  Q2
```

Note: *No external entries are generated for the PST.*

Example of ATXREF Importing the GPL and QL

```
      $INCLUDE  'AAWRENCH'             .
      WRENCH                           .
      ATXREF    'GPL=ABC','QUE=ABC'    . GPL, QL marked for import
      GPLDEF    'CFACCS'               . Only require CFACCS
      QUEDEF    'Q1' 'Q2'              . No point in specifying
                                       . access
      PSTDEF                           . This is our very own PST ENTRY
      .....
      LOADC     R4,Q2                  . Queues and procedures used normally
      .....
```

```
        CALL      CFACCS                        . Ditto
        .....
        SEG       SSTDEF,1                      . Must define segment for code
        TEST1 PROCDEF,1 'ENTRY',,,,*TEST,,*TEST . GPL and QL from procedure TEST
        $END
```

ATXREF creates the following equates:

```
CFACCS    $EQU    ABCCFACCS
Q2        $EQU    ABCQ2
Q1        $EQU    ABCQ1
```

Example of ATXREF Cross-Referencing from a Set of Service Routines

If you have a collection of separately assembled subroutines, you can use the import technique. The only difference is that you do not use PROCDEF.

```
          $INCLUDE  'AAWRENCH'           .
          WRENCH                         .
          ATXREF    'GPL=ABC','QUE=ABC'  . GPL and LQ marked for import
          GPLDEF    'CFACCS'             . Requires CFACCS only
          QUEDEF    'Q1' 'Q2'            . No point in specifying access
          .....
          CALL      CFACCS               . Ditto
          .....
NEWSEG    SSTDEF                         . Must define segment for code
          $END
```

ATXREF generates the same definitions as for the previous import case.

```
CFACCS    $EQU    ABCCFACCS
Q2        $EQU    ABCQ2
Q1        $EQU    ABCQ1
```

3.2. CPADEF - Define Defaults

The CPADEF MASM procedure defines default reservations for dynamic segments, dynamic queues, process control stacks, and lock table. Although the system automatically expands the program space as segments or queues are requested, if a specific reserve is required, CPADEF signals to the DCP/OS loader that it requires this amount of space at load time. This procedure can also help you avoid unnecessarily fragmenting bank space.

The CPADEF MASM procedure must be the last CPA module MASM procedure in the element.

Format

```
CPADEF  xqt, xsst, stks, locks
```

Parameters

xqt	The additional number of spare queue table entries.
xsst	The additional number of spare system segment table entries.
stks	The maximum number of active process control stacks.
locks	The size of the lock table.

3. ENTDEF - Additional Entry Points

The ENTDEF MASM procedure specifies additional entry points that share all the properties of a paired procedure number (PN). See the PROCDEF/PRIVDEF or INITDEF MASM procedures.

Format

```
pname    ENTDEF    'ep','pair'
```

Parameters

pname	The procedure name (maximum eight characters). Because this is the name other users must specify in their GPLDEF, it is the name used to call the procedure.
ep	The entry point label.
pair	The paired procedure name (PNAME).

3.4. GPLDEF - Define Gated Procedure List

The GPLDEF MASM procedure defines a gated procedure list (GPL). It specifies the names of all procedures called by the procedure.

Format

```
[gplnm]   GPLDEF,m   'pname',BLOCK,;
                     'pname',BLOCK,k;
                     ... etc ...
```

Parameters

gplnm	The name of this GPL (optional).
m	Number of blank entries to reserve at the start of the table.
pname	The name of a called procedure.
BLOCK	Causes initial blocking of access to this PN. The builder creates a flag from this field which is stored in the most significant bit of the PN.
k	Number of slots to reserve before inserting next PN.

5. INITDEF - Generate Initial Procedure

The INITDEF MASM procedure defines a procedure in the same way as PROCDEF, but also specifies that the generated procedure is to be the initial procedure of the built program. See 3.8 for additional discussion on procedure entry points and shared lists.

Format

```
pname   INITDEF,n,BLK 'ep',[icw],psw,,gpl,pst,ql,la;
            'snamea',WRITE  'snameb',WRITE
```

Parameters

pname	The procedure name (maximum eight characters). Because this is the name that other users must use in their GPLDEF, it is the name used to call the procedure.
n	The SSTDEF cross-reference value from which system segment numbers (SSNs) are taken. If *n* is not present, only segments given with SNAME are used.
BLK	Causes the procedure to be initially blocked.
ep	The entry point label.
icw	The initial instrumentation control word for the procedure (optional).
psw	The initial PSW setting.
gpl	The GPL name (defaults to unnamed GPL).
pst	The PST name (defaults to unnamed PST).
ql	The QL name (defaults to unnamed QL).
la	The LA name (defaults to unnamed LA).
snamex	The name segment (SNAMEx) loaded on CALL.

Notes:

1. *The number of segments may vary from zero to three.*
2. *If the SSTDEF cross-reference (n) is not used, the segment names are provided in the sequence SD0,SD1,SD2,SD3.*
3. *If the SSTDEF cross-reference is used, the order is SD3,SD2,SD1,SD0.*

WRITE — Gives this procedure write access on segment 1, 2, 3, or 4. The builder creates a flag in the most significant bit of each SSN field in the PT. It is redundant if write access is specified in the SSTDEF for the named segment.

6. IOPDEF - Generate PP Program Table Entry

The IOPDEF MASM procedure generates a port processor (PP) program table entry.

Format

```
IOPDEF   'name'  'pao','[pai]'   'proc'  'qlist' ;
         'seg0','seg1','seg2','seg3'
```

Parameters

name	The name of this PP program.
pao	The program address for output.
pai	The program address for input (optional).
proc	The name of the CP procedure scheduled when this PP stops due to a specification error (see 4.1).
qlist	The name of the queue list (optional). If this parameter is a number rather than a name, it is used as the length of an empty queue list.
segx	The segment visibility needed (optional).

3.7. LNKDEF - Define Link Area

The LNKDEF MASM procedure defines a link area (LA).

Format

[*lnknm*] LNKDEF,*m*

Parameters

lnknm	The name of this link area (optional).
m	The number of entries to reserve.

.8. MARKDEF/PROCDEF/PRIVDEF - Generate Procedure Module

These MASM procedures generate a procedure module with the PSW set up for a user or for privileged software, depending on the MASM procedure used (MARKDEF, PROCDEF, or PRIVDEF).

MARKDEF sets the marked procedure flag bit in the dictionary entry. Use MARKDEF with the new DIC$FENT service to allow runtime searches for flagged procedures.

Procedure Entry Points

The first PROCDEF MASM procedure in an assembly generates a full module definition. Subsequent PROCDEF MASM procedures generate new procedures sharing the GPL, PST, QL, and LA (GPQL) of the first (paired) procedure but with their own SSNs and entry point (EP).

Shared Lists

The GPL, PST, QL, and LA parameters are not normally provided. This results in the unnamed lists being associated with the procedure definition.

Each list may be specified by name in one of two ways:

- Specific list
- Cross-reference list

A specific list can specify a named list within the assembly. This allows you to use multiple lists within an assembly. In this case, you provide the list name without quotation marks.

A cross-reference list can specify a list that is actually generated in another assembly. In this case, you provide the name of the owner procedure, in the following format:

```
*'name'
```

The list resolution is processed at build time. See the ATXREF MASM procedure that is used in conjunction with this facility.

Format

```
pname    PROCDEF,n,BLK 'ep',,,,gpl,pst,ql,la ;
             'snamea',WRITE  'snameb',WRITE ...
```

```
pname    PRIVDEF,n,BLK 'ep',icw,psw,,gpl,pst,ql,la ;
              'snamea',WRITE  'snameb',WRITE ...

pname    MARKDEF,n,BLK 'ep',icw,psw,,gpl,pst,ql,la ;
              'snamea',WRITE  'snameb',WRITE ...
```

Parameters

pname — The procedure name (maximum eight characters). Because this is the name that other users must specify in their GPLDEF, it is the name used to call the procedure.

n — The SSTDEF cross-reference value from which SSNs will be taken. If *n* is not present, only segments given with SNAME are used.

BLK — Indicates that the procedure is initially blocked.

ep — The entry point label.

icw — The initial instrumentation control word for the procedure.

psw — The initial PSW setting.

gpl — The GPL name (defaults to unnamed GPL).

pst — The PST name (defaults to unnamed PST).

ql — The QL name (defaults to unnamed QL).

la — The LA name (defaults to unnamed LA).

snamex — The name segment (SNAMEx) loaded on CALL.

Notes:

1. *The number of segments may vary from zero to four.*
2. *If the SSTDEF cross-reference (n) is not used, the segment names are provided in the sequence SD0,SD1,SD2,SD3.*
3. *If the SSTDEF cross-reference is used, the order is SD3,SD2,SD1,SD0.*

WRITE — Gives this procedure write access on segment 1, 2, 3, or 4. The builder creates a flag in the most significant bit of each SSN field in the PT. It is redundant if write access is specified in the SSTDEF for the named segment

3.9. PSTDEF - Define Procedure Segment Table

The PSTDEF MASM procedure defines a procedure segment table (PST). It specifies the names of all segments that can be loaded (LSEG) by the procedure.

Format

```
[pstnm]   PSTDEF[,m ]'sname',WRITE[,k];
          'sname',WRITE[,k];
          ...
```

Parameters

pstnm — The name of this PST (optional).

m — The number of blank entries to reserve at the start of the table (optional).

sname — The name of the segment that may be loaded (using LSEG) by a procedure in this assembly element.

WRITE — Permits write access to the segment. The builder creates a flag from this field that is stored in the most significant bit of the SSN.

Caution: Any writes to a transient segment will be lost unless the segment is held in memory. See the CPA$HOLD and CPA$FREES services in Section 7 and the descriptions of resident and transient segments in Section 9.

k — The number of slots to reserve before inserting the next SSN (optional).

3.10. QUEDEF - Define Queue List

The QUEDEF MASM procedure defines a queue list. It specifies the names of all queues used by the procedure.

Format

```
[qlnm]   QUEDEF[,m ]'qname',g,p,a[,k];
         'qname',g,p,a[,k];
         ...etc...
```

Parameters

qlnm	The name of this queue list (optional).
m	The number of blank entries to reserve at the start of the table (optional).
qname	The name of the queue used.
g,p,a	Get, put, or arm. Field position is significant; separate unused fields with commas. For example: „a
k	The number of slots to reserve before inserting the next entry (optional).

.11. QUEUE - Define a Queue

The QUEUE MASM procedure defines a queue.

Format

qname `QUEUE,addr type size mode thrshld,'procedure',qx R1`

Parameters

qname The queue name.

addr The hardware ID (for example, RID, SID).

type

MCT	(message control table queue)
LIT	(literal queue)
SAI	(software attention item queue; uses the SQL1 for the run)
SPACE	(space queue)
LIST	(linked MCT queue)
SAI CPA	(software attention item queue; uses the CPA SQL1)

size The queue size.

mode The number of back entries allowed.

thrshld The queue threshold (maximum 255).

procedure The name of SAI PN scheduled.

qx The queue priority (0 to 3). If a queue name is given instead, the named target queue is placed in either the CPA SQL1 or the SQL1 of the run by the builder/loader and is used as a cascade queue.

R1 The 16-bit parameter in R1 on initial dispatch.

Note: See the INFO file on the DCP/OS level 5R1 release tape for additional information on cascading queues.

A queue defined as SAICPA becomes an event queue and is placed in the CPA architectural SQL1.

A queue defined as SAI becomes an event queue and is placed in the non–architectural SQL1 of the run.

Worker queues specifying a Qx with a SAICPA queue name have the following SAI:

Run/PN	User R1 Parameter
16 bits	16 bits

Worker queues specifying a Qx with a SAI queue name have the following SAI:

User R1 Parameter	Real Address of Event SAI Queue
8 bits	24 bits

Up to 251 event queues can be defined on the SQL1 for each run.

Up to 251 event queues can be defined for the one CPA SQL1 in the system.

3.12. SSTDEF - Define CPA System Segments

The SSTDEF MASM procedure defines one or more CPA system segments. One or more segment modules are created, depending on the size of the code or data. If the segment size exceeds the 4K byte boundary, another segment is automatically generated. These extra segments are named as follows:

segnam$n

where

segnam The original segment name.
n The sequential number, starting at 1.

Format

```
snamex   SSTDEF,n  'memtype'  mode-flags  offset,max,min  lc,lc,...
```

Parameters

snamex The segment name (maximum six characters).

n A cross-reference to PROCDEF. The associated procedure uses all SST entries generated without having to name each one.

memtype The acceptable parameters are

'AA$ZRES'	for resident segments
'AA$ZTRN'	for transient segments

mode-flags At least one of the following modes must be specified:

CP	communications processor (CP) executable
IOP	port processor (IOP) executable
READ	segment may be read from
SUBSEG	segment may be subsegmented (default)
WRITE	segment may be written to
CONTIG	segment should be contiguous

offset The base virtual address; for example, SDR2.

max	An optional parameter specifying the maximum number of SDRs this segment can use. A warning is given if the size is greater.
min	An optional parameter specifying the minimum number of SDRs this segment can use. A warning is given if the size is less.
lc	The location counter to include in this segment.

Section 4
Contingency Handling

This section describes the following:

- Contingency handling for PP state item
- Contingency handling for CP specification exception
- Contingency handling for IPM state change
- Contingency handling for throttle level change
- Contingency handling for II console command

Contingency handling allows a designated procedure called a contingency handler to examine the event type and the task which experienced the event. It also provides for taking other appropriate action, such as cleaning up the affected environment and rearming the queues.

The user program may directly dispatch a specific procedure when certain exception conditions are detected. The process of invoking user contingency handlers for all cases is detailed in the following subsections. In general, the contingency handling procedures are simply dispatched with a parameter that specifies the identity of the entity in question. The contingency handler can then invoke a DCP/OS service to retrieve detailed contingency information and take appropriate action.

4.1. PP State Items

A PP state item is posted whenever a PP program errs or halts. A CP contingency procedure may be designated for each PP program on the IOPDEF statement in the PP program source. This statement defines the PP program environment.

At run time, if a state item is posted from any port running that PP program, the CP contingency procedure is dispatched with the dispatch parameter (R1) set to the port number.

The contingency procedure then takes the required action. This could simply be to kill the PP (via PP$ELIM) and restart it. Usually, the handler will invoke the DCP/OS service PP$GETSI to retrieve the state item. The cause of the PP stop may then be determined. If the state item was caused by the PP issuing a HALT instruction, it may well be that an operator had commanded the program to stop the PP. In this case it is likely that the handler will be programmed to free the PP and return. If the state item was caused by a PP program error, the handler may be programmed to restart the port.

These decisions are made when designing the contingency handler.

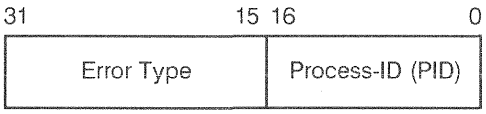
.2. CP Specification Exceptions

The DCP/OS traps all CP errors with CPA forced calls to PN1 or PN4. If the program process errs and registers a contingency handler to manage the error, then the process in error is suspended and the process-ID is queued on a literal queue. This contingency literal queue is specified when the contingency registration service (PC$CREG) is called.

If the contingency handling process itself crashes, the program is aborted.

When such a CP contingency occurs, a literal specifying the contingency type and process-ID is placed on the designated queue. If the contingency queue is full, the program is aborted, since this is regarded as an error in the contingency handler itself.

The CP contingency literal is 32 bits wide. It uses the 32 bits as follows:



Bit 31 of the error type can take one of the following values:

- 1 DCP/OS error code
- 0 CPA error code

The process-ID (PID) enables the contingency handler to identify the crashed process on calls to PC$CKILL, PC$CSTAT, and so forth.

The contingency handling services provide the user program with tools to construct a sophisticated, nonstop environment.

The simplest method of recovering from errors in a CPA environment is to clean up a crashed process and to ensure that all queues serviced by the initial procedure are rearmed. The contingency handler kills the errant process (PC$CKILL) and reschedules the initial procedure. Some data may be lost but the program survives what would otherwise be fatal errors.

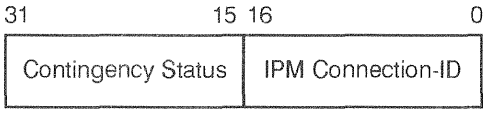
4.3. IPM State Change

The inter-program message (IPM) services provide a facility to transfer messages between separate, cooperating programs. An IPM connection implies the existence of a receiver and a transmitter. If the program at either end of the connection aborts or closes down its end of the connection, the result is an IPM state change. An IPM connect request, issued by a transmitter, also results in a state change at the receiver end.

An IPM contingency queue signals this change of state to the paired end of the connection. The caller designates the contingency queue when registering a receiver or setting up an IPM connection. It is an optional facility. If a contingency queue is not specified, the program using the IPM must maintain its own contingency checking (with timers and calls to get the IPM connection status, and so forth).

When such an IPM contingency occurs, a literal specifying the IPM connection-ID is placed on the designated queue. If the contingency queue is full, the contingency is ignored and it is up to the IPM user to request the connection status at some future time.

The IPM contingency literal is 32 bits wide. It uses the 32 bits as follows:



The contingency status can take one of the following values:

- IPC$CONN (Transmitter issued a connect - IPM$CONN)
- IPC$DISC (Transmitter issued a disconnect - IPM$DISC)
- IPC$FREE (Receiver issued a free - IPM$FREE)

In an IPC$DISC or IPC$FREE contingency, terminate the IPM connection and retry.

In an IPC$CONN contingency, open an IPM connection in the reverse direction. Establish a two-way pipe as follows:

- Retrieve the connection status using IPM$STAT.
- Get the transmitter's IPM name from IP$REPLY. field
- Issue an IPM connect request using IPM$CONN.

4. Throttle Level Change

A DCP/OS system specifies a number of buffers that satisfy normal operation. This number reflects the real and usual demand for buffers by the system including all programs that are run. When the system's demand for buffers goes beyond this number, all programs that have registered a procedure to receive status changes (using PC$SREG) will receive a throttle level change indication. Any program can access segments SCT and SYSINF to ascertain the current memory conditions using the memory management variables and parameters contained in these segments.

Based upon the status change indication, and the additional memory management information (if needed), the registered program should modify its own behavior in order to protect itself and to help prevent the exhaustion of the potential buffer pool. The program should view this indication with some urgency since the demand for buffers by some programs is at that point very high.

When in throttle, the DCP/OS reduces demand for buffers by preventing the following activities:

- New program loads except for privileged runs
- New transient segment loads (into buffers) except for DCP/OS transients, privileged runs, and programs registered for status change notification (this is logically transparent)
- Instrumentation

When in throttle, the DCP/OS breaks banks into 128-byte buffers or 4096 byte segments, as required. Adjustable memory management parameters exist for various frequency and sizing considerations.

If the entire potential buffer pool is exhausted, the DCP/OS will begin to kill programs as determined by the run priority. Lower priority jobs are killed first.

The run-priority is a number from 1 to 26 according to the priority letter entered on the @RUN line where A corresponds to 26, B corresponds to 25, and so on.

When the system's demand for buffers returns to normal and the throttle condition subsides, all programs that have registered using PC$SREG receive a throttle level change indication.

See Section 9 for more information on throttling and on memory management.

4.5. II Contingencies

When an operator enters

II *run-name*

on the DCP/OS console, the contingency procedure for that run (registered in PC$IREG) is dispatched with the value of ST$II in R1.

Section 5
Debug Facilities

This section provides the following information:

- Describes the tasks debug can perform
- Explains how to enter debug mode
- Lists the debug commands
- Describes the function and format of each debug command

5.1. Using Debug Facilities

Debug is the tool for debugging programs under DCP/OS. It uses full-screen displays and supports the functions shown in Table 5-1. The debug commands are fully described in 5.6.

Table 5-1. Debug Functions

Description	Command
Set the address display to byte mode	A
Set and display breakpoint parameters	B
Catch system service call (SVC) to be trapped	C
Dump debug session	D
End debug mode	E
Flip register display	F
Resume a trapped process	G
List available commands	H
Inspect Communications Processor Architecture (CPA) tables	I

continued

Table 5-1. Debug Functions (cont.)

Description	Command
Display the current call return stack	K
Set the display length of the inspect area	L
Modify the current display area	M
Step through the next one or more instructions	N
Step, but treat SCALL and CALL instructions as one instruction	O
Set and display profile parameters	P
Query dictionary	Q
Modify registers	R
Display segments	S
Switch the status display to trap	T
Inspect the virtual environment of a trapped process	V
Set the inspect mode to word	W
Kill the current trap	X
Set breakpoint parameters to zero	Z
Display the next page of memory	+
Display the previous page of memory	-

.2. Entering Debug Mode

You can enter debug mode in one of these ways:

- Using the console mode command DEB. See the *DCP Series, DCP/OS Operations Reference Manual.*
- Using the $ option when invoking a program in demand mode. See the @CRASH utility in the *DCP Series, DCP/OS Operations Reference Manual.*
- Using the demand mode bypass command @@DEB. See the *DCP Series, DCP/OS Operations Reference Manual.*

- Automatically, on encountering a program error, a breakpoint, or a system service call (SVC) catch

  Note: *In demand mode, you must be privileged to enter debug mode if you are not debugging a program. See the @PRIV command in the* DCP Series, DCP/OS Operations Reference Manual.

Regardless of what method you choose, you are presented with a full-screen display, comprising an inspect area and a status display area (see Figure 5-1). After you enter debug mode, you may begin entering any of the debug commands listed in Table 5-1.

```
*DEBUG*  Wed  16 Nov 88 11:29:48

Commands:     A=address      B=breakpoint  C=catch SVC   D=dump       E=end
F=flip        G=go           H=help        I=insp tab    J=n/a        K=stack
L=length      M=mod memory   N=single step O=one step    P=profile    Q=query
R=mod regs    S=segment      T=disp trap   U=n/a         V=virtual    W=word
X=kill trap   Y=n/a          Z=zero brkpt  +=next page   -=prev page

For help in the use of a particular command enter:      H <command>

Status:   Amod=WORD  Imod=REAL     Adr=000000 #Tr=00 U=Y
```

Figure 5-1 . Debug Screen

3. Command Format

The command format and a description of each command are in the following subsections. If you enter the command correctly, a display appears. If you enter the command incorrectly, you may hear a beep (depending upon the type of terminal you use). The command line remains on the screen so you may reenter the command correctly.

All numbers are in hexadecimal except the display length, breakpoint count, and the number of instruction steps.

4. Setting Program Traps

Starting Up

You usually initiate debug sessions by executing the user program with the $ option in demand mode. This option forces a pseudotrap at the entry point to the user program so you may set breakpoints and patches before executing the program.

Debugging

After entering debug using the $ option, you must cause your program to trap at a required point and then examine the virtual memory environment before proceeding.

There are two methods of causing program traps:

- Using the debug B command (breakpoint) to set breakpoint instructions
- Using the debug C command (catch service call) to trap user-selectable service system calls (SVCs)

A troublesome area may be swiftly pinpointed if a program contains many SVCs. By dynamically redefining the CATCH options and modifying code and data, you can patch several problems in a short time.

If you have reached the problem area, and the solution is not obvious, you may have to set a breakpoint at a specific address in user code and then examine the conditions when the program hits the breakpoint. While on a breakpoint, you may want to use the NEXT command to observe registers and memory as each instruction is executed.

5.4a. Profiles and Logging

You can dump portions of a running Telcon system or a program without terminating Telcon to analyze problems that occur during online traffic. The debug P command (profile) executes nearly any combination of debug commands at breakpoints you specify, and displays them. If you choose to log the results to disk, you can run the program unattended and use the utility program @SYS to view the log file using the debug option (@SYS,DQ).

The breakpoint and profile combinations can be simple and concise or varied and complex depending on the type and extent of the data you are gathering.

You can instruct profile commands to select the breakpoints on which specified commands execute. You can specifiy up to 8 profiles (numbered 1 through 8). The profile numbers you use change and zero out profiles. You can, however, omit the number when you add a new profile, because the new profile is assigned the next available profile number.

Use L to specify the logging option. You can attach the L to the profile number as follows:

Example

```
P 1L
```

Or, if you do not specify a profile number, you can specify the logging option directly after the profile command as follows:

Example

```
P L
```

You specify BRKPTs within parentheses. If you omit the BRKPT, the profile (P) command assigns the BRKPTs. Use a semicolon (;) to separate profile-specified commands.

Use a percent sign (%) before each comment.

The following example enters a new profile that logs its output on all current BRKPTs:

Example

```
B  1,PFRED,02a
B  2,PFRED,60a
P  L,V 5800;  V 8000; K   % On all BPs, Log mem & stk
```

The following example enters a new profile that takes a program dump when BRKPT 4 is reached.

*Note: Memory dumps are written to the specified file and not to the SYS$*SYSLOG file.*

Example

```
B 4, PFRED,614
P (4),D SYS$*SYSDUMP,P
```

Section 5.6.14a shows the format and additional examples of the P command.

This comparison between a normal session and a logging session illustrates how the profile (P) command fits into the debugging scheme:

Normal session	Logging session
set some breakpoints	set some breakpoints set some profiles (P command with logging specified)
do a G(o) command	do a G(o) command
BRKPT occurs, is displayed	BRKPT occurs, is logged
do some commands (V's.., S's.., K's.., N's.., I's.., etc...)	P commands string of specified commands are automatically done, resulting displays logged
do another G(o) command	resumes upon expiration of P commands string of commands, a G(o) at string-end ok too
more of the same...	leave it running, walk away
@@DEB, Z command	Z command on a BRKPT conditioned profile

A third type of session may be a combination in which only part of the profile settings specify logging. As BRKPTs are reached, logging occurs only where specified.

In a normal session, you can preset commands on a profile to save reentering them as each breakpoint is reached.

When you need a logging session with complex BRKPT/profile settings, you can test a normal session (without logging) first, and then you can add the logging (L) to the profiles and enter a go (G) command.

5.5. Example Using Debug

The following example assumes that you invoked the @CRASH utility using the D option, which specifies that you enter debug automatically when a trap occurs. A Communication Processor Architecture (CPA) specification error occurred and you automatically entered debug mode.

The screen status provides the following information:

Loc — The approximate address where the error occurred in the program (varies with DCP type)

PN — The name of the procedure where the program crashed

Op — The instruction operation code

Err — The error code of the trap, which is either:

x/xx	CPA-detected error
xxxx	DCP/OS-detected error

(When you use @FAC x-xx or @FAC xxxx to display error code, the error code slash changes to a hyphen.)

The following four steps show how to use this information in the problem-solving process.

Step 1 A trap occurred in LINSVS at 1A7 on an 1F instruction with a 4/01 error. This error (@FAC 4-01) indicates a blocked procedure. See the DCP Series Implementation Reference Manual, Volume 1, Volume 2, Rev. 1, and Volume 3 (UP-12728). The three items are on the call/return stack.

Enter V 1A0 to check the code. You find it is a CALL (1F) instruction at 1A5 with GPLx=2.

Enter V 1A0

Response

```
                              CALL (1F) instruction

 *DEBUG*  Wed  16 Nov 88 11:29:48     Run=LES       Program=TEL7R2


 0001A0:   B801  160A  0005  041A  046F  1F00  0002  04F6       .........o......
 0001A8:   120A  0005  C80D  9D06  1F20  029E  B803  8280       ......... ......
 0001B0:   0276  B804  160A  0005  041A  0429  0002  1F00       .v.........).o..
 0001B8:   0002  04F6  120A  0005  C80D  9D06  0002  029E       ..............  ..
 0001C0:   B803  8280  0276  B26B  0007  C8D1  0002  BBF9       .....v.k.....W..
 0001C8:   0008  9103  B24B  0005  0600  3800  0002  226B       .....k....8.c."k
 0001D0:   9115  071F  2004  C014  4022  0600  0002  C300       .... ...@".......
 0001D8:   0000  D220  B000  0412  0600  8107  C300  0000       ... ............

 Trap:     Loc=01A7 PN=LINSVS          CC=cvPz Op=1F #Stk=03 Err=4/01 [XS=1B0000]

 S 0-07: AA06FA0D 00000000 00000000 00000000 EF1D109E EB1B210C 00000000 00000000
 S 8-15: 00000000 00000000 00000000 8C05CC9F 00000000 00000000 00000000 00000000
 S16-23: 8C04F29F 00000000 00000000 00000000 8E09651F 00000000 00000000 00000000
 S24-31: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

 R 0-07:   0001  5800  0000  8000  0252  0000  0001  800A
 R 8-15:   8000  580A  5800  A000  0000  0001  0000  0001

 Status:   Amod=WORD  Imod=VIRTUAL  Adr=0001A0 #Tr=01 U=Y Seg=LNSVSS   SSN=01C9

 Enter command (H=help)-
```

Step 2 Enter K to display the stack and confirm the call/return addressing with a code listing.

Enter K

Response

```
*DEBUG*  Wed  16 Nov 88 11:49:06     Run=LES      Program=TEL7R2

1B0200:   20D2  005C  0000  0000  0180  0006  0000  0000        ..\.............
1B0210:   000D  D85D  0000  0000  0180  012D  8000  0000        ...]........-....
1B0220:   01A7  D15D  0000  0800  0180  012D  0000  0000        ...]........-....



Trap:     Loc=01A7 PN=LINSVS          CC=cvPz Op=1F #Stk=03 Err=4/01 [XS=1B0000]

S 0-07: AA06FA0D 00000000 00000000 00000000 EF1D109E EB1B210C 00000000 00000000
S 8-15: 00000000 00000000 00000000 8C05CC9F 00000000 00000000 00000000 00000000
S16-23: 8C04F29F 00000000 00000000 00000000 8E09651F 00000000 00000000 00000000
S24-31: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

R 0-07:   0001  5800  0000  8000  0252  0000  0001  800A
R 8-15:   8000  580A  5800  A000  0000  0001  0000  0001

Status:   Amod=BYTE  Imod=REAL     Adr=1B0200 #Tr=01 U=Y

Enter command (H=help)-
```

Step 3 Enter I GPL, LINSVS to display the gated procedure list. Entry number 2 is 803F, which means procedure 03F, but access is blocked.

Enter I GPL,LINSVS

Response

```
               blocked procedure 03F

 *DEBUG*  Wed  16 Nov 88 11:32:22     Run=LES      Program=TEL7R2

 1D3720:   000B  00A6  803F  0121  0055  0075  0013  0000       .....?.!.U.u....



 Trap:     Loc=01A7 PN=LINSVS          CC=cv Pz #Stk=03 Err=4/01 [XS=1B0000]

 S 0-07: AA06FA0D 00000000 00000000 00000000 EF1D109E EB1B210C 00000000 00000000
 S 8-15: 00000000 00000000 00000000 8C05CC9F 00000000 00000000 00000000 00000000
 S16-23: 8C04F29F 00000000 00000000 00000000 8E09651F 00000000 00000000 00000000
 S24-31: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

 R 0-07:   0001  5800  0000  8000  0252  0000  0001  800A
 R 8-15:   8000  580A  5800  A000  0000  0001  0000  0001

 Status:   Amod=BYTE  Imod=REAL     Adr=1D3720 #Tr=01 U=Y

 Enter command (H=help)-
```

Step 4 Enter Q PT,03F to find the name of procedure X'3F', which turns out to be CFACCS. This is the procedure you attempted to call. Note that the defined flag is not set.

Enter Q PT,03F

Response

```
                              defined flag              name of procedure

*DEBUG*  Wed  16 Nov 88 11:35:49     Run=LES      Program=TEL7R2

09FC14:   4346  4143  4353  2020  0002  0000  0000  003F       CFACCS ........?



Trap:     Loc=01A7 PN=LINSVS          CC=cvPz Op=1F #Stk=03 Err=4/01 [XS=1B0000]

S 0-07: AA06FA0D 00000000 00000000 00000000 EF1D109E EB1B210C 00000000 00000000
S 8-15: 00000000 00000000 00000000 8C05CC9F 00000000 00000000 00000000 00000000
S16-23: 8C04F29F 00000000 00000000 00000000 8E09651F 00000000 00000000 00000000
S24-31: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

R 0-07:   0001  5800  0000  8000  0252  0000  0001  800A
R 8-15:   8000  580A  5800  A000  0000  0001  0000  0001

Status:   Amod=BYTE  Imod=REAL     Adr=09FC14 #Tr=01 U=Y

Enter command (H=help)-
```

Analysis

The CFACCS entry in the GPL of LINSVS was marked INITIALLY BLOCKED because the CFACCS procedure was not included at build time.

Solution

Rebuild your program with an MLF that contains the procedure CFACCS.

5.6. Debug Commands

The debug commands are described next in alphabetical order. The descriptions include function, format, and parameters.

1. A—Address

The A command sets the address display to byte mode.

Additional byte mode addresses may be entered without the A command, since debug remembers the current addressing mode. The number of bytes displayed is governed by the current length setting (see L command), which defaults to the maximum of 128 bytes when a program is started.

Format

A [*address*]

Parameters

address A real-memory hexadecimal address, from one to six digits

5.6.2. B—Breakpoint

The B command sets, suspends, or displays breakpoints. Breakpoints are specified by procedure or segment, and are optionally qualified by countdown value, register content, or virtual memory content.

You may specify up to eight breakpoints at a time. When a valid PN or segment and address are entered, the breakpoint instruction is set. You may alter any parameter at any time for the respective breakpoint number.

When setting a continuous trace (a special form of breakpoint - see variant 3 of format), you must omit the PN, segment, and address parameters and set the register condition and the memory condition. The count parameter is optional. If you subsequently enter the PN or segment and address, the continuous trace becomes a normal breakpoint. Continuous trace can be used with only one breakpoint number at a time.

The breakpoint status is displayed on the bottom half of the trap screen, which can be redisplayed using the B command. The following is an example of a breakpoint status screen display:

```
 Brkpt #    PN        Segment      Adrs       Count    Reg/Int Cond    Mem/Date Condition

    1     FRED        SFRED       0012       8/150    R15 = 34CF    2003I4   ='DISKDRIV'
    2-    DUMMYPN     0122        0000         -       R5 = 0003     B020     # A0D0FFFF
    3                 *SSSS3      0024         -          -                    -
    4     0156        0101        0198         -       R0 <0000S                -
 -> 5     DISK        SDISK       000E         -          -          1000     = 00F2
    6p    FRED        SFRED       002A         -      Int = 0015    900731   / 07:30
    7       -           -           -          -       R8 = 0E0E                -
    8       ?           ?           ?       003/004    CC = cvPz     &     R8 >8000S
```

The display columns are defined as follows:

Brkpt #

The breakpoint table number ranging from 1 to 8. Up to 8 breakpoints may be defined.

An arrow (->) points to the current breakpoint (if any).

A minus sign (-) after the breakpoint number means this breakpoint is currently suspended.

The letter p following the breakpoint number indicates that a profile command has referenced the breakpoint.

PN

The procedure number (PN) specified on the B command when setting a breakpoint.

Segment

The segment specified on the B command when setting a breakpoint or the segment implied by specifying a PN on the B command. If no PN is specified on the B command, the segment is preceded by an asterisk.

Adrs

The address where the breakpoint is set, either segment-relative or procedure-relative, as entered on the B command.

Note: A question mark (?) in the PN, Segment, and Adrs field means a continuous trace is active.

Count

An optional field that specifies that a trap is to be activated only after the specified number of times the breakpoint is hit. The second number (150) shows the count that was entered on the B command. The first number shows the current count (8). These two numbers will be equal when the trap is activated. The first number resets to zero after the trap.

Reg/Int Cond

This field displays the conditional register setting for a register conditional trap or a time interval conditional trap, as entered on the B command when specifying a breakpoint.

Note CC = cvPz in breakpoint 8. Here the condition code of the program status word (PSW) is checked for four conditions to be true. These conditions are as follows:

c = no carry
v = no overflow
P = positive
z = nonzero

Breakpoint 4 shows the constant 0000S. The S (signed) suffix here indicates that the comparison is arithmetic. Otherwise, all comparisons are logical.

Memory/Date Condition

This field indicates the memory condition, or a date and time condition, for a virtual address trap, as entered on the B command when specifying a breakpoint.

The ampersand (&) on breakpoint 8 indicates that both the register condition and memory condition must be true for the breakpoint to be activated. Without the ampersand, only one of the conditions need be true.

Breakpoint 1 shows a virtual byte address indicated by I4 (register pair R4 and R5). In this case, 2003 (the contents of R4) is the effective word base address. R5 contains the byte offset beyond the base.

When a breakpoint occurs, the banner line is displayed as follows:

```
Brkpt 1: Loc=11B0 PN=PMNFG      CC=cvPz Op=1F #Stk=02
```

Format

Format 1	B	(Display breakpoints)
Format 2	B	[*n*],*name,adr*[,*count,R op value,*[&]*adrR op value*] (Set breakpoints)
Format 3	B	[*n*],*name,adr*[,*count,INT=min,DT=date/time*] (Set breakpoints with date, time, and interval)
Format 4	B	[*n*],[,,*count,R op value,*[&]*adrR op value*] (Continuous trace)
Format 5	B	[*n*],[,,,*INT=min,DT=date/time*] (Set date, time and interval)
Format 6	B	*n*,- (Suspend breakpoint)
Format 7	B	*n*, + (Activate breakpoint)

Format 8 B
(Suspend all breakpoints)

Format 9 B +
(Activate all breakpoints)

Parameters

n Breakpoint number (1 through 8)

name Procedure or segment name/number

*Note: Use an asterisk (*) prefix to specify a segment (for example, *SDISK or *96). The name/number without an asterisk is searched primarily as a procedure and, secondly, as a segment.*

adr Virtual address in PN or segment

count Optional count value (decimal 1 through 999). A counter is incremented on each breakpoint. When the counter reaches the count value, a trap occurs and the debug screen is displayed. The counter resets to zero after the breakpoint.

In the same manner, the counter is incremented and checked against the count each time continuous trace conditions are satisfied.

R Optional condition register

Rn = single register (R0 - R15)
CC = condition code (in PSW)

op Relational operator

= equal to value
not equal to value
> greater than value
< less than value

value Value to be compared with register contents

If Rn is specified, the value may be

'aa' = one or two ASCII characters

hhhh[S] = from one to four hex digits, right-justified and zero-filled.

optional S (signed) indicates the comparison is to be arithmetic.

If CC is specified, the value may be from one to four of the following alpha letters. Uppercase and lowercase letters represent mutually exclusive conditions. Therefore, choose only one case for each letter.

C = carry	c = no carry
V = overflow	v = no overflow
P = positive	p = not positive
Z = zero	z = nonzero

If the register content on each breakpoint check matches the condition specified, a breakpoint occurs and the debug screen is displayed.

& Optionally specifies that both the register condition and the memory condition must be true to display the breakpoint screen. If this parameter is omitted, either condition will cause the breakpoint display.

adrR Optional virtual address whose contents are compared against a specified value. Valid forms of this entry are

adr = virtual address.
R = virtual address is contents of R
adrR = virtual address is adr + contents of R
adr/byte = explicit virtual address with byte offset

where R is specified as follows:

For word addressing
Rn = virtual address (R1 - R15)

For byte addressing (Im)
paired even/odd registers
(R2/R3,R4/R5...R14/R15)
Im = virtual word address
Im + 1 = byte offset

For byte addressing (adr/byte)
adr = virtual word address
byte = byte offset

op — Relational operator

= equal to value
not equal to value
> greater than value
< less than value

value — Value to be compared with memory contents

'aaaaaaaa' = from one to eight ASCII characters
hhhhhhhh = from one to eight hex digits

If the virtual address content on each breakpoint check matches the condition specified above, a breakpoint occurs and the debug screen is displayed.

INT — Specifies to set time interval (INT = min)
Example: INT = 20

min — Optional time interval (decimal 1 through 1440 minutes). When a breakpoint occurs, the debug screen displays when the interval expires. The interval is reset after each expiration. If you specify the interval without an associated breakpoint segment and address, the help screen displays with zero traps.

DT — Sets date and time (DT = date/time).
Example: DT = 891130/0700

date/time — Optional date or time (YYMMDD/HHMM). When the associated breakpoint occurs, the debug screen displays only if the current date and time is greater than or equal to the specified date/time. If you specify the date/time without an associated breakpoint segment and address, the help screen displays with zero traps. If you specify date/time and INT parameter after the first breakpoint is reached, subsequent displays will be under the control of the specified interval.

Notes:

1. *Format 4 of the B command is for a continuous trace. A continuous trace is activated when the segment and address parameters are omitted and a conditional parameter is specified. A trap occurs when the condition is satisfied. The instruction, segment, and address are then displayed. This feature enables you to pinpoint where memory is being corrupted. Since each instruction is breakpointed, performance is affected.*

   The continuous trace mode is terminated by pressing MSG WAIT and entering @@DEB.

2. *The effective virtual address specified in a memory condition may be a 16-bit word address or a word address with byte offset. The word/byte offset mode is known as Im addressing.*

- Word addressing (adrRn, adr, or Rn):

  The effective word address specifies a single 16-bit word in the virtual address range 0-FFFF (where n = 1,2,...15).

- Im byte addressing (adrIm, Im, or adr/byte):

  The effective word address is a similar combination of adr and Im (m=2,4,...14). A byte offset from this effective word address is specified in Im+1 or as a constant. In this mode, a breakpoint condition works on an arbitrary string of characters or hex digits starting at even or odd byte addresses.

Example

The following are examples of setting breakpoint:

```
B ,SDISK,20,10,R1=5000,5800/A='DISK'
B ,,,,R13>7FF,&200R7# FFF
B ,PDISK,80A,,,I10=FFFFFFFF
B ,SFRED,2A,,INT=15,DT=900731/0730
```

This is the resulting display that appears on the bottom half of the trap screen:

```
Brkpt | PN    | Segment | Adrs | Count   | Reg/Int Cond | Mem/Date Condition
  1   |       | *SDISK  | 0020 | 000/010 | R1 = 5000    | 5800/00A ='DISK'
  2   |   ?   |    ?    |  ?   |         | R13>07FF     |  &0200R7 #FFF
  3   | PDISK |  SDISK  | 080A |         |              |      I10 =FFFFFFFF
  4   |       | *SFRED  | 002A |         | INT=0015     | 900731   /07:30
  5   |       |         |      |         |              |
  6   |       |         |      |         |              |
  7   |       |         |      |         |              |
  8   |       |         |      |         |              |
```

5.3. C—Catch Service Call

The C command specifies system service calls (SVCs) to be trapped on entry and exit. This enables you to view the SVC request parameters and the results of the SVC.

Additionally, the C ERR command filters only SVC error returns for display. By default, all procedures in your program are monitored, but use of the PN proc parameter further limits SVC traps to the specified procedure.

You may specify multiple SVC traps in a single command line. Trap selections are cleared with the OFF parameter, which clears all SVC trap flags. The ALL parameter invokes a trap on every SVC your program specifies. You may use the C command to redefine the trap options any time a program is active. If you do not have an active program, you may hear a warning beep (depending upon the type of terminal you are using) if you try to enter the C command.

Format

C [ERR,] [PN,*proc*,] [*param*,*param*,...]

Parameters

ERR — Catch SVC error returns only

proc — The procedure name or number (defaults to ALL)

param — SVC-type, OFF, or ALL

SVC-type is one of the following:

CPA	Trap CPA services
DIC	Trap dictionary services
ELT	Trap E$ SDF services
FILE	Trap FILE services
INST	Trap instrumentation services
IPM	Trap IPM services
LM	Trap line module services
PC	Trap dispatcher services
PP	Trap PP services
RUN	Trap run services

Example	
CPA	Catches all CPA SVCs on entry and exit in all procedures
C PP,LM,PN,PPORT	Catches only PP and LM type SVCs in procedure PPORT
C ERR,ALL	Catches all SVC error returns

5.4. D—Dump Debug Session

The D command invokes a dump without terminating a session. The amount of memory dumped and the file to which the dump is taken are specified by the @CRASH utility program (executed before the current program) or by the parameters used with the D command.

Format

D [*filename*][,*options*]

Parameter

filename The optional filename to which you direct the dump

options

F	performs a full dump (all memory)
P	dumps user program and DCP/OS memory

Note: If no option is specified, the default is P.

5.6.5. E—End Debug Mode

The E command exits debug if no traps are active, and returns you to system control. If any traps are active, a warning beep may sound (depending upon the type of terminal you are using). All traps must be inactive before you exit (see the X command).

Format

E

.6. F—Flip

The F command switches the display from segment descriptor registers (SDRs) to process control registers (PCRs) and system control registers (SCRs) or vice versa.

The current display is swapped to the other display for this and subsequent trap displays. This command is valid only if a trap is active.

Format

F

5.6.7. G—Go (Resume Trap)

The G command resumes a trapped process. The trap may simply be resumed at the current location. You may also resume the process at a different location by giving the address parameter and optionally trap at that location.

DCP types vary the precision of the address of a specification exception condition. The location is often given as the address following the instruction in error or trap. You should carefully note the location before using G without an address parameter. This does not apply to SVC traps invoked by the C command, since the DCP/OS always handles these correctly.

If no trap is active, you may hear a warning beep (depending upon the type of terminal you are using).

Format

G [*virtual address*] [,*N*]

Parameters

virtual address	Indicates a hexadecimal word address in the current virtual visibility
N	Forces into step mode at specified address

Example

G	Go (resume trap) at the current address
G 912	Go (resume trap) at virtual address 912
G 912,N	Go (resume trap) at virtual address 912 and trap into step mode

.8. H—Help

The H command without a parameter displays an abbreviated list of available commands in the lower half of the screen. This is the default display when no trap is active.

Use the T command to display the registers of an active trap.

Format

H [*command*]

Parameters

command	The debug command name. If you use this parameter, the format of the optional command is displayed.

Example

H C	Displays help information for the C command.

5.6.9. I—Inspect Table

The I command inspects Communications Processor Architecture (CPA) tables. The display length is set to the entry length of the table type specified. Always prefix your numbers with a zero.

Format

I *table-name* [*index*]

Parameters

table-name

EXEC	Exec process stack table
GPL	Gated procedure list
ICT	Interface control table
LA	Link area
PPIT	PP instrumentation table
PPPT	PP program table
PPQL	PP queue list
PST	Procedure segment table
PT	Procedure table
PUT	Procedure use table
Q	Queue
QL	Queue list
QT	Queue table
SST	System segment table

index As follows:

TABLE NAME	INDEX
PT, GPL, PST, QL, LA, and PUT	Procedure name or number to which table belongs
SST	Segment name or number
PPPT	Program name or number
ICT, PPQL, PPIT	PP number
QT, Q	Queue name or number
EXEC	Run number

;.10. K—Display Call/Return Stack

The K command displays the current call/return stack in the inspect area of the screen.

Format

K

Example

```
*DEBUG*  Wed   16 Nov 88  14:38:48    Run=LESA       Program=DISK

198200:  20D4  005C  0000  0000  0100  0006  0000  0000
198210:  0851  D05D  0400  0807  0100  000A  0000  0000
198220:  0044  D35D  0000  0000  0100  000D  0000  0000


SVC-Exit:  SVC=FILE[8000] called from 0014 in PN=PMENUS  [XS=198000]

S 0-07: AA06F482  AA0F7691  8A0BB085  EB02611C  00000000  8A0F0607  00000000  8C044F9F
S 8-15: 00280008  00290009  00000000  8C05AD1F  00000000  CC074980  00000000  00000000
S16-23: 00000000  00000000  00000000  00000000  AA03FC06  00350015  00000000  00000000
S24-31: 00000000  00000000  00000000  003B001B  003C001C  CB005A0B  CB025E84  AA0A0F8D

R 0-07: 001E  5000  5000  0000  0000  5000  0000  0000
R 8-15: 0000  0000  8000  0004  8000  0000  0000  0000

Status: Amod=BYTE  Imod=REAL  Adr=198200  #Tr=01  U=Y
CATCH SVC: FILE,CPA,RUN,PC,PP,LM,DIC,IPM,ELT,INST
```

5.6.11. L—Set Display Length

The L command sets the display length of the inspect area (top half of the screen) to the specified (decimal) number of bytes/words, depending upon the current display mode.

Format

L *length*

Parameters

length The display length for the inspect area (1 through 128); larger (up to 4096) when P and L are used together

Examples

L Reset length to 128, the maximum

L 16 Set display length to 16

P L,V 5800;L 900
Set display length to 900 when profile command specifies logging (the first L specifies logging; the second L specifies length)

;.12. M—Modify

The M command modifies the current display area and register contents according to the current (modified) contents on the screen or the optional parameter values from the command line.

To modify memory and change the contents on the screen when no parameters are specified, enter M and press XMIT to transmit. You may also use function key 4 (F4) instead of entering the M command.

When optional parameter values are specified, memory is modified starting at the first location of the display area. It uses as many of the parameter values as can fit on the command line. Registers are not modified in this case.

In both cases, with or without parameters, only the displayed memory is modified by the M command. A warning beep may sound (depending on the type of terminal you are using) if a nonprivileged user attempts to modify either real memory or DCP/OS segments.

Note: It is advisable to set the display length to the number of bytes required (L command) when modifying a volatile area of memory. This prevents you from inadvertently modifying other locations.

If a segment is modified, the segment use count automatically increments by 1, which freezes the segment in memory.

Format

M [*value1/value2.../valuen*] or function key 4 (F4)

Parameters

value 1 to 4 hexadecimal digits separated by a comma, a space, or a slash. Values are right-justified and zero-filled before you modify memory.

5.6.13. N—Next

The N command sets the debug mode into single-step action, which executes one instruction. Service calls (SVCs) are treated as only one instruction. After the first N command is entered (following a trap), you can simply transmit to continue single-stepping. This is faster than reentering the N command.

The N command is a powerful feature with several variants. In its simplest form, N single-steps through the current instruction and traps, causing a new debug display. If another command is entered (such as V or S to display memory), then single-step mode may be resumed by entering N again.

Another variation of the N command is to specify the number of instructions to be stepped through. The default is 1. In this variant, it is also possible to specify whether or not the debug page is to be displayed on each instruction or only at the completion of the number of instructions specified. This is done by using the D parameter.

All user mode instructions may be single-stepped. This includes CALL, RTN, SCALL, SRTN, jumps, and conditional jumps. All forced calls, including the SVC instruction, are effectively treated as single instructions. This means that when you single-step through code, any SVC or segment load (using PN4) appears transparently as a single instruction.

Format

N [*number*] [,*D*]

N I, inst

N V, va

Parameters

number Specifies the number of instructions (1-9999) to execute before trapping. For example, N 100 executes 100 instructions.

Note: A breakpoint encountered along this path preempts the specified number of instructions from being completed.

D Displays screen on each single-stepped instruction. Thus, registers and memory displays can be viewed during execution of the specified number of instructions.

I, inst Traps at the next instruction that matches the value of inst. For example, N I, 06 traps at the next operations code 06.

V, va Traps at the next virtual address that matches the value of va. For example, N V, 081 traps at the next virtual address containing the first three digits 081.

In step mode, the following banner line is displayed:

```
Step:  Loc=12CC  PN=PMNFG   CC=cvPz  Op=B8  #Stk=03
```

5.6.14. O—One Step

The O command functions the same as the N command, except that the CALLs and SCALLs are treated as one instruction, similar to the SVC. As a result, by using the O command, you can avoid stepping through uninteresting subroutines and procedures.

Format

O [*number*][, *D*]

O I, inst

O V, va

5.6.14a. P—Profile

The P command sets, clears, or displays profiles. You can specify one or more debug commands with a P command to display profiles of memory and registers.

Formats

P

Use this format to display profiles.

P *prof#*

Use this format to change a profile.

P *[prof#][L] [(bp#s)], cmd1; cmd2; ... cmdn*

Use this format to add a profile.

P *Z*

Use this format to zero all profiles.

P *Zprof#*

Use this format to zero a specified profile.

Parameters

prof# is the profile number that you want to change or clear. The profile command string redisplays on the prompt line for reediting when you use Format 2. When you use Format 3 and omit the profile number, the next available profile number is used.

L — is the log screen display parameter. When you use this parameter, the debug screen displays are written to disk.

bp#s — are the BRKPTs on which you execute profile commands. The default is all BRKPTs.

cmd — is any debug command and its respective parameters (except E); in addition, only one P command can be specified for the cmd parameter and it must be the last (or rightmost). You can substitute the space and a semicolon (;) for the XMT key. Separate each command and its parameter with a semicolon (;) and leave a space between each command and its parameters.

Note: *If you omit the command, only the breakpoint screen is logged.*

Examples

```
P L
P 21(1)
```

Z — specifies that you want to clear profiles.

Debug Screen Display

When you enter a P command, the profile setting status is displayed on the top half of the screen. The following are two examples of command entries and screen displays follow.

Example 1

```
B  1,FRED,2a,,INTV=15,DT=890717/0730

P  L (1),A 24000; L 300              % log data(300 bytes) each 15 min

B  2,,,,,DT=890717/1030

P  L (2),D SYS$*DUMP; @@X            %  Dump pgm and kill run after 3 hours


*DEBUG*   Tue Jul 89 15:58:28        Run= WTH       Program=DISK

 Prof# ( Brkpts )    P r o f i l e *** C o m m a n d *** S t r i n g s
   1L |(1       )  A 24000; L 300    % log data(300 wds) & stack each 15 min
   2L |( 2      )  D SYS$*DUMP; @@X  % Dump pgm and kill run after 3 hours
   3  |(        )
   4  |(        )
   5  |(        )
   6  |(        )
   7  |(        )
   8  |(        )
********************************************************************************
********************************************************************************
 Brkpt    PN        Segment   Adrs   Count  Reg/Int Cond  Mem/Date Condition
   1p    | FRED    | *SFRED  | 002a |      | INTV= 15    |   890717 / 07:30
   2p    |         |         |      |      |             |   890717 / 10:30
   3     |         |         |      |      |             |
   4     |         |         |      |      |             |
   5     |         |         |      |      |             |
   6     |         |         |      |      |             |
   7     |         |         |      |      |             |
   8     |         |         |      |      |             |

Enter command,  1 Traps (H=help)-
```

In the first example, for BRKPT 1, at the specified time, a BRKPT is set in segment FRED at address 002a. When the BRKPT is reached for the first time, the commands of profile 1 are executed and the output is logged to disk. Thereafter, the first BRKPT is reached after each 15-minute interval and causes logging. At the specified time on BRKPT 2, Profile 2 executes a dump command, followed by @@X to terminate the run.

Note: *The L following prof# number (top half of screen) indicates logging for this profile. The p following the BRKPT number (bottom half of screen) indicates at least 1 profile for this BRKPT.*

Example 2

```
B  1,  SFRED,10                          (At initialization time)

B  2,  SFRED,3a                          (During main program loop)

B  3,  SFRED,12f                         (At termination time)


P  L (1), S SFRED 40; M 1f20/07a0;              % SCALL out to patch area

P  L (1), 07a0; M 0720/1020                    % Execute clobbered instruction

P  L (1), 0610/0050/2710/281c/1d20             % Do patch & return

P  L (2), L 10; V 205c                         % Log 10 wds data per BRKPT 2

P  L (3), S SFRED,40;M 0720/1020;Z;Z P         % End: remove patch, zero all
```

```
*DEBUG*   Tue Jul 89 15:58:28        Run= WTH       Program=DISK

 Prof# ( Brkpts )    P r o f i l e *** C o m m a n d *** S t r i n g s
|  1L  (1        ) S SFRED 40; M 1f20/07a0;     % SCALL out to patch area     |
|  2L  (1        ) 07a0; M 0720/1020            % Execute clobbered instr.    |
|  3L  (1        ) 0610/0050/2710/281c/1d20     % Do patch & return           |
|  4L  ( 2       ) L 10; V 205c                 % Log 10 wds data per BRKPT 2 |
|  5L  (  3      ) S SFRED,40;M 0720/1020;Z;Z P % End: remove patch, zero all|
|  6   (         )                                                            |
|  7   (         )                                                            |
|  8   (         )                                                            |
*******************************************************************************
*******************************************************************************
 Brkpt    PN        Segment   Adrs   Count  Reg/Int Cond  Mem/Date Condition
|  1p    |        | SFRED   | 0010 |      |              |                   |
|  2p    |        | SFRED   | 003a |      |              |                   |
|  3p    |        | SFRED   | 012f |      |              |                   |
|  4     |        |         |      |      |              |                   |
|  5     |        |         |      |      |              |                   |
|  6     |        |         |      |      |              |                   |
|  7     |        |         |      |      |              |                   |
|  8     |        |         |      |      |              |                   |

Enter command,  1 Traps (H=help)-
```

The second example shows how to specify multiple profile strings for the same BRKPT (BRKPT 1). Comments are shown here, but could be omitted, allowing the entire profile command string section to be used only for commands.

5.6.15. Q—Query Dictionary

The Q command queries the dictionary and, if it finds the entry, displays it in the display area.

Two forms of this command are available. The first form has a single parameter, which is the name of the entry to be searched for. The second form requests a search for a specific entry of a given CPA type.

If a named entity cannot be found, check that the correct program is in debug and that the dictionary is toggled on (U option).

Format 1

Q *name*

Format 2

Q *type,number*

Parameters

name Any 8-character name in the dictionary

type A table type:

PT	Procedure table
QT	Queue table
SST	System segment table
PPPT	Port processor program table

number Relative number within type

Example

Q MYSEG Searches for entry named MYSEG

Q SST,0123 Searches for entry of segment 123

This page is intentionally left blank.

5.6.16. R—Register Modify

The R command either reads the current screen display of registers or, if present, the optional parameter values from the command line. In either case, you can modify the registers of the trapped process. To modify registers, you change the registers on the screen, tab forward to the command line, enter R, and press XMIT (transmit). Or you can specify the register change-value as a parameter.

The R command is relevant only when a trap is active. If no trap is active, you may hear a warning beep (depending upon the type of terminal you are using).

Format

R [*r,nnnn*]

Parameters

r = register numbers (0 to 15)

nnnn = 1 to 4 hexadecimal digits. This value is right-justified and zero-filled before modifying the register.

.17. S—Display Segment

The S command displays the requested segment in ASCII and hexadecimal and resets the display length to 128 bytes.

Note: *If the virtual address goes out of range as you page forward through a segment, the inspect area (upper half) of the screen is blanked. You can restore the display by entering a new, valid virtual address or paging back (P) into range.*

If the specified address is greater than a full segment length (X'800' words), the segment number is automatically rounded up and the address is rounded down within the range of a full segment. In these cases, verify the segment name.

Format

S *segment-ID* [*address*]

Parameters

segment-ID	Segment name or number
address	Relative word address (defaults to zero)

Example

S 0100	Displays segment 100 from address zero
S SDISK 04B8	Displays segment SDISK from address 4B8

5.6.18. T—Display Trap

The T command switches the status display area from HELP (if current) to TRAP. The trap area displays the SDRs and registers that are pertinent at the time of the trap. If no trap is active, you may hear a warning beep (depending upon the type of terminal you are using).

Format

T

.19. V—Inspect in Virtual Mode

The V command inspects the virtual environment of a trapped process or of an active port processor (PP).

Note: *If the virtual address goes out of range as you page forward through virtual memory, the inspect area (upper half) of the screen is blanked. You can restore the screen display by entering a new, valid virtual address or paging back into range.*

In CP mode, the V command is relevant only when a trap is active. However, you can enter debug to set up the display before invoking a trap, since the mode is remembered over debug entries. For example, you may expect to find a message at virtual address X'5800' at many points in a program. The display mode may be set to 'V 5800' at any time in anticipation of a trap. The message is automatically displayed at the appropriate time.

Format

V [*address*]	CP mode
V [*address*][*/port*]	PP mode

Parameter

address	Virtual address	0000-FFFF words for CP
/port	Port number	0000-3FFF bytes for PP

Example

V 1800	Displays from virtual word address 1800 in the CP context
V 1000/6	Displays from virtual byte address 1000 of PP #6

5.6.20. W—Set Inspect Mode to Word

The W command sets the inspect mode to word and displays from a specified word address in real memory.

Format

W [*address*]

21. X—Kill Trap

The X command kills the current trap by releasing all segments, allocatable space, and the task stack. If present, the next queued-up trap is displayed. Otherwise, control is returned to the system manager. (Compare with the E command.) This command does not kill the whole program, even if it is single-tasking. After returning to run mode, the program may be killed by using the @@X T command.

If no trap is present, you may hear a warning beep (depending upon the type of terminal you are using).

Format

X

5.6.22. Z—Zero Breakpoint

The Z command clears any or all breakpoints and redisplays the current breakpoint settings.

Format

Z [*number,CRM*]

Parameters

number The breakpoint number to clear. If the number is omitted, all breakpoint settings are cleared.

CRM The breakpoint count, register, and memory parameters may be cleared without clearing the entire breakpoint. Any combination of the following:

C = clear the count
R = clear the register condition
M = clear the memory condition

.23. +(Plus)—Next Page

The + (plus) command displays the next page of memory when you are in step mode. The length of the display is not changed. Any default inspect length or user set display length is used when paging through memory. Use the XMIT key to display the next page when you are not in step mode.

Format

+

5.6.24. -(Minus)—Previous Page

The - (minus) command displays the previous page of memory. The display length is not altered. After the first - (minus) command is entered, each time you press XMIT the program will go back one more page.

Format

-

Section 6
MASM Utility Procedures

This section describes the following general MASM procedures:

- Basic CP Utility MASM Procedure (AAWRENCH)
- Extended Utility MASM Procedure (AEXTPROC)
- Structured MASM Procedures (AASTPROC)

These procedures are in the library file, DCPOSEQU, on the release tape.

6.1. AAWRENCH—Basic CP Utility MASM Procedures

The basic central processor (CP) utility, AAWRENCH, provides MASM procedures for the CPA CP instructions and CPA module definitions. It also offers a series of instructions and table definition enhancements. The CPA module definitions are discussed in Section 3. The CPA instructions are defined in the DCP Series Implementation Reference Manual, Volume 1, Volume 2 Rev. 1, and Volume 3 (UP-12728).

Table 6-1 defines the symbols used for the procedure descriptions.

Table 6-1. Key to Symbols Used

Symbol	Definition
BQF	Byte-equated field: symbol defined by BEQUF procedure.
EQ	Simple equated value with no relocation.
EQF	Field defined as an EQUF or simple equate (no relocation).
EQR	Equated value that may have relocation (not EQUF).

continued

Table 6-1. Key to Symbols Used (cont.)

Symbol	Definition
Im	Pair of index registers (Im, Im+1 : Im even) containing Im mode address.
N	Integer.
Q	Field defined as a BEQUF (possibly with some added relocation information).
R	Register.
RA	Source/destination register.
RM	Index register containing word address or offset.
RW	Work register.
RWE	Even work register.
RWO	Odd work register.
S	String.
TV	Truth value : true<>0 false=0.
W	An EQF, possibly with some added relocation information.
X	Any acceptable MASM value.
[...]	Contents optional.
<...>	Parameter written as contents <BLOCK> => parameter is written as BLOCK.
{...}	Repeat contents (with blank separator) any number (>0) of times.
n*{...}	Repeat contents at most n times, but at least once.
{{...}}	Repeat contents (with comma separator) any number (>0) of times.
n*{{...}}	Repeat contents at most n times, but at least once.
*	Following parameter flagged with *. This is a shorthand way of writing <*>.
v <- func(params)	The function (func) takes parameters (params) and returns type (v).
/	Use preceding or following options, but not both.

6.1.1. Location Counter Handling

The following MASM procedures in Table 6-2 can alter the location counter when invoked. Use them carefully since setting the location counter to a specified place does not guarantee that the actual code or data generated is similarly aligned. The use of segments constructed from multiple location counters can also lead to addresses having different properties from those expected.

Table 6-2. Location Counter Handling

MASM Proc	Format	Definition
EVEN	EVEN . No parameters	Rounds the current location counter to the even word boundary.
ODD	ODD . No parameters	Rounds the current location counter to the odd word boundary.
ZM2K	ZM2K . No parameters	Rounds the current location counter to the next SDR.
ZM64	ZM64 . No parameters	Rounds the current location counter to the next granule.

6.1.2. Constant Manipulation MASM Procedures

The group of MASM procedures in Table 6-3 generates either a nibble instruction or the equivalent 16-bit instruction. For example, LOADC generates either a load nibble (LN) or a load constant (LK) instruction (or a load from R_m (LR) instruction, under certain circumstances). For more information on LN, LK, or LR, see the DCP Series Implementation Reference Manual, Volume 1, Volume 2 Rev. 1, and Volume 3 (UP-12728).

Table 6-3. Constant Manipulation MASM Procedures

MASM Proc	Format	Definition
ADDC	ADDC RA,EQR [,RM]	Adds the constant into a register.
ASHFL	ASHFL RA,EQR [,RM]	Arithmetic shift left register by constant.
ASHFLD	ASHFLD RA,EQR [,RM]	Arithmetic shift left register pair by constant.
ASHFR	ASHFR RA,EQR [,RM]	Arithmetic shift right register by constant.
ASHFRD	ASHFRD RA,EQR [,RM]	Arithmetic shift right register pair by constant.
COMC	COMC RA,EQR [,RM]	Compares (signed) the constant with a register.
COMUC	COMUC RA,EQR [,RM]	Compares (unsigned) the constant with a register.
CSHFL	CSHFL RA,EQR [,RM]	Circular shift left register by constant.
CSHFLD	CSHFLD RA,EQR [,RM]	Circular shift left register pair by constant.
CRSN	CRSN RA,EQ . EQ <16	Circular right shift nibble.
DIVC	DIVC RA,EQR [,RM]	Divides register pair by a constant.
LOADC	LOADC RA,EQR [,RM]	Loads the constant into a register.
LSHFR	LSHFR RA,EQR [,RM]	Logical shift right register by constant.
LSHFRD	LSHFRD RA,EQR [,RM]	Logical shift right register pair by constant.
MULTC	MULTC RA,EQR [,RM]	Multiplies register pair by a constant.
SBITS	SBITS RA,[*]EQF	Sets bits in a register.
SUBC	SUBC RA,EQR [,RM]	Subtracts the constant from a register.
SVC	SVC EQ	Issues an SVC instruction for the function specified.
TBITS	TBITS[,RW] RA,[*]EQF	Tests bits in a register.
ZBITS	ZBITS RA,[*]EQF	Clears bits in a register.

*The value is taken as a mask for the bits to be tested; otherwise, it is taken as an EQUF and thus defines a set of bits.

6.1.3. Field Manipulation MASM Procedures and Functions

A field is any value defined by an EQUF. Fields are used to hide table formats, allowing them to be varied without necessitating reprogramming, only recompiling. Fields defined by EQUF, which act on words, should be distinguished from the fields defined by BEQUF, which act on bytes.

Table 6-4 lists the MASM procedures used to manipulate fields.

Table 6-4. Field Manipulation MASM Procedures

MASM Proc	Format	Definition
ALL	N <- ALL({{EQF}})	Returns a value equal to the logical OR of the mask generated by the individual parameters. If a parameter is an EQUF, the mask is given by the function MASK. If it is an equate, the mask is the single bit equal to 1*/equate.
ALLBUT	N <- ALLBUT({EQF})	Returns the one's complement of the above (ALL) function.
BIT	N <- BIT(EQF)	Returns the leftmost (most significant) bit of the field.
BYTELIKE	TV <- BYTELIKE(W)	Returns true if the parameter passed is a byte (8 bits long and aligned with left-bit=15 or 7); otherwise, it returns false.
CMASK	N <- CMASK(EQF)	Returns the one's complement of mask.
EQUF	EQUF EQ [,EQ [,EQ]]	The EQUF directive defines word-oriented fields. Use it for DCP coding rather than the MASM directive $EQUF. For example: EQ,EQ,EQ = F,LB,LEN where F The field offset (which may be relocatable). LB The starting (leftmost) bit of the field. Defaults to 0.

continued

Table 6-4. Field Manipulation MASM Procedures (cont.)

MASM Proc	Format	Definition
		LEN The length of the field, 0 implies 16. Defaults to 0.
FIELDLIKE	TV <- FIELDLIKE(W)	Returns true if the field is neither wordlike nor bytelike; otherwise, it returns false.
LEN	N <- LEN(EQF)	Returns the field length in bits.
MASK	N <- MASK(EQF)	Returns a value with ones occupying the field position, and zeros elsewhere.
MAXVAL	N <- MAXVAL(EQF)	Returns the largest integer that can be contained within a field.
RBIT	N <- RBIT(EQF)	Returns the rightmost (least significant) bit of the field.
WORD/WD	N <- WORD(EQF)	Returns the word offset defined within an EQUF
	N <- WD(EQF)	(WD is a shorter name).
WORDLIKE	TV <- WORDLIKE(W)	Returns true if the parameter passed is a word (16 bits); otherwise, it returns false.

4. Value Manipulation

The functions in Table 6-5 provide simple facilities for manipulating and testing values.

Table 6-5. Value Manipulation MASM Procedures

MASM Proc	Format	Definition
MAXIMUM	N <- MAXIMUM({{EQ}})	Returns the maximum integer passed as a parameter.
MINIMUM	N <- MINIMUM({{EQ}})	Returns the minimum integer passed as a parameter.

continued

Table 6-5. Value Manipulation MASM Procedures (cont.)

MASM Proc	Format	Definition
RELOC$	TV <- RELOC$(X)	Returns true if the value passed is relocatable; otherwise, it returns false.
TABL	TV <- TABL(X)	Returns true if the value passed is a field; otherwise, it returns false.
TBEQUF	TV <- TBEQUF(X)	Returns true if the value passed is a byte field; otherwise, it returns false.
TWOS	EQ <- TWOS(EQ)	Returns a value converted to two's complement form, for example, TWOS(-1) -> 0FFFF, TWOS(-2) -> 0FFFE, etc. A negative number must be passed as a parameter; otherwise, TWOS adds one to the value.
TREG	TV <- TREG(X)	Returns true if the value passed is a register; otherwise, it returns false.
WORD$	EQR <- WORD$(W)	Returns an EQR with the relocation of the original parameter. This function removes the EQUF information from the parameter.
X	N <- X(EQ)	Returns a hexadecimal value.

6.1.5. Extended Instructions

The MASM procedures in Table 6-6 extend the DCP instruction set. Most of them use field definitions, performing the same operation as the equivalent machine instruction, but using the field rather than a constant. For example, LOAD R1,ABC,R12 loads R1 from the field defined by ABC, offset by R12. Compare this with the equivalent machine instruction L R1,ABC,R12, which loads R1 from the wordlike ABC, offset by R12. In most cases, the format of the extended instruction is identical to the machine instruction, except for the use of an optional work register.

All of the following MASM procedures can accept W-field specifications that contain relocation. They can thus be used with any type of field or label specification.

Table 6-6. Extended Instructions

MASM Proc	Format	Definition
ADD	ADD[,RW] RA,W[,RM]	Adds the contents of a field in memory to a register.
CLEAR	CLEAR[,[*]RW] W[,RM]	Clears (sets to zero) the contents of a field in memory. * Implies RW already contains the contents of the word in which the field is defined.
COM	COM[,RW] RA,W[,RM]	Compares (signed) a field in memory against a register.
COMU	COMU[,RW] RA,W[,RM]	Compares (unsigned) a field in memory against a register.
EXC	EXC RA,RM	Exchanges the contents of two registers.
EXTRACT	EXTRACT W[,RM]	Extracts a field from a register and right-justifies it in a register.
JTBL	JTBL[,[EQ][,EQ]] RA,[*]EQR[,RM]	Generates instructions to jump to the address given by the table entry.
LDK	LDK RA,EQ	Loads a 32-bit constant into a register pair.
LOAD	LOAD RA,W[,RM]	Loads a register from a field within memory.
MOVE	MOVE[,RW[,RW] W,RM] W,RM	Moves the contents of a field in memory to a field in memory.
RADD	RADD[,[*]RW] [*]RA,W[,RM]	Adds the contents of register RA + 1 into a field in memory and sets the register RA with the result. * RW contains offset to the field; * RA means RA can be changed.
RDEC	RDEC[,[*]RW] [*]RA,W[,RM]	Decrements the field in memory by one and stores the result in a register. * RW contains offset to the field; * RA may be changed and is used as a work register.
RINC	RINC[,[*]RW] [*]RA,W[,RM]	Increments the field in memory by one and stores the result in a register. * RW contains offset to the field; * RA may be changed and is used as a work register.

continued

Table 6-6. Extended Instructions (cont.)

MASM Proc	Format	Definition
RSUB	RSUB[,[*]RW] [*]RA,W[,RM]	Subtracts the contents of the register RA + 1 from a field in memory and sets the register RA with the result. * RW contains offset to the field; * RA means RA can be changed.
SET	SET[,[*] RW] W[,RM]	Sets the contents of a field in memory to ones. * Implies RW already holds the contents of the word in which the field is defined.
STORE	STORE[,RW] [*]RA,W[,RM]	Stores the value in a register into a field in memory. * Restores RA to its original value.
STOREC	STOREC[,RW] EQ,W[,RM]	Stores a constant into a field in memory.
SUB	SUB[,RW] RA,W[,RM]	Subtracts the contents of a field in memory from a register.
SZMIE	SZMIE [*]EQ,[*]RM	Clears multiple words in memory with optional advance of the index register (EQ+1 words are cleared). IF * set on EQ: at completion RM is unpredictable ELSE * not set on EQ IF * set on RM at completion RM points to the first nonzeroed word ELSE * not set on RM at completion RM unchanged (normal) ENDIF ENDIF
TEST	TEST[,[*] RW] W[,RM]	Tests the contents of a field in memory against 0.

* Implies RW already contains the contents of the word in which the field is defined.

6. Table and Instruction Generation

The MASM procedures in Table 6-7 generate code or data.

The GEN MASM procedure generates a variable number of constants and provides conversion for two's complement negatives. One or two operands may be supplied. With one operand, that operand is the value of the word. With two operands (separated by a comma), each operand is the value of the respective byte of the word. Where negative constants might appear, use this MASM procedure instead of the + operator.

The GEN MASM procedure avoids the disadvantage of the RES MASM directive while providing capabilities not found in the RES statement.

Format

`GEN[,[*]rpt] val1[,val2]`

Parameters

rpt — A one-field repeat parameter immediately after the call name indicates the nonnegative (0,1,2,...) number of words to generate with the same value.

Default: 1

**rpt* — An asterisk (*) before the repeat parameter signifies that every word must be listed (unless unlist is set). Otherwise, only the first and last words are listed (unless unlist is set).

val1 — This value (positive or negative) goes in the left half of the word. If *val2* is not present, *val1* is the value of the whole word.

val2 — This value (positive or negative) goes in the right half of the word. If *val2* is not present, *val1* may also be a multiple character string. If *val1* is flagged, example *'ABCDEFGHIJKL', then only the first 8 characters are displayed.

Example

```
0000    0001    GEN       1
0001    FFFF    GEN      -1
0002    0102    GEN     1,2
0003    01FE    GEN    1,-2
0004    FE02    GEN,5  -2,2
0008    FE02
0009    0101    GEN,*4  1,1
000A    0101
000B    0101
000C    0101
000D    4845    GEN 'HELLO.'
000E    4C4C
000F    4F2E
```

Table 6-7. Table and Instruction Generation (AAWRENCH)

MASM Proc	Format	Definition
GEN	GEN[,[*]RPT] VAL1[,VAL2]	Generates a variable number of constants and converts two's complement negatives.
GENTAB	GENTAB[EQ[,EQ]] {[*]S[,EQF[,EQ]]/EQ[,EQF]}	Generates a table in which the values, which cannot have any relocation, are positioned according to field definitions. There is no maximum table size.
GINL	GINL 4*{{S}}	Generates an instruction with simultaneous display. It looks like the SHOW MASM procedure, but generates the specified instruction. Used with MASM procedures to save typing. For example: GINL 'LABEL','LOAD','R2,FIELD,R12','GET FIELD'
JMPTBL/ BYTBL	JMPTBL[,[S] [,EQ] [,EQ] [,EQ]] {EQR[,EQ]}	Generates a table (word/byte oriented).
	BYTBL[,[S] [,EQ] [,EQ] [,EQ]] {EQR[,EQ]}	Generally used to build jump tables. The maximum table size is 255 words/bytes.

.7. Display Control

The MASM procedures and functions in Table 6-8 can be used to perform standard display operations. See also the GINL MASM procedure in Table 6-7.

Table 6-8. Display Control

MASM Proc	Format	Definition
CVALAS	S <- CVALAS(R/EQF/W/S/BQF)	Converts a MASM value into a string that can be assembled.
DISL	DISL { {{ S / [*EQ] }} }	Displays one or more lines constructed from mixed numeric (decimal or binary) and string values.
NOLIST	NOLIST S	Switches on (S='ON') or off (S='OFF') the display of MASM procedure expansions. It is initially set to on, but it clarifies a listing if no MASM procedure expansions are produced.
OPSTR	S <- OPSTR (3*{{ [*] R/EQF/EQR }})	Generates a string that represents the operand field of an instruction. It is often used in conjunction with the SHOW or GINL MASM procedures. See also OPSTR$.
OPSTR$	S <- OPSTR$ (3*{{[*] R/EQF/W/S/BQF }})	Similar to OPSTR except that the string produced can also be assembled if used within a microstring. The format of the string produced is often less obvious than that of OPSTR, but it has the advantage of losing no relocation or control information in the conversion. It can be used well with GINL.

continued

Table 6-8. Display Control (cont.)

MASM Proc	Format	Definition
PROCLIST	PROCLIST S / 5*{{EQ}} S EQ	Sets display control parameters such as tab positions, etc. In particular, the comment character generated in front of all lines produced by the SHOW MASM procedure can be changed from ' ' to '+', making it easier to distinguish between real comments and generated comments and lines.
SCREAM	SCREAM { {{ S / [*]EQ }} }	Displays the error message consisting of the prefix ***** ERROR , connected with one or more lines of mixed numeric (decimal or binary) and string values Note: An F-flag is generated for each use of SCREAM.
SHOW	SHOW 4*{{S}}	Displays an instruction from within a proc.
TAB	S <- TAB(S,EQ)	Pads out a string to the specified tab position.
WARNING	WARNING { {{ S / [*EQ] }} }	Displays a warning message consisting of the prefix "***** WARNING", connected with one or more lines of mixed numeric (decimal or binary) and string values.

.8. AAWRENCH Procedure Summary

Groups of AAWRENCH procedures were dealt with in 6.1 through 6.1.7. Table 6-9 summarizes the MASM procedures found in AAWRENCH.

Table 6-9. Summary of AAWRENCH MASM Procedures

MASM Proc	Format	Type
ADD	ADD[,RW] RA,W[,RM]	Extended instructions
ADDC	ADDC RA,EQR [,RM]	Constant manipulation
ALL	N <- ALL({{EQF}})	Field manipulation
ALLBUT	N <- ALLBUT({EQF})	Field manipulation
ASHFL	ASHFL RA,EQR [,RM]	Constant manipulation
ASHFLD	ASHFLD RA,EQR [,RM]	Constant manipulation
ASHFR	ASHFR RA,EQR [,RM]	Constant manipulation
ASHFRD	ASHFRD RA,EQR [,RM]	Constant manipulation
BIT	N <- BIT(EQF)	Field manipulation
BYTELIKE	TV <- BYTELIKE(W)	Field manipulation
BYTBL	BYTBL[,[S] [,EQ] [,EQ] [,EQ]] {EQR[,EQ]}	Table and instruction generation
CLEAR	CLEAR[,[*]RW] W[,RM]	Extended instructions
CMASK	N <- CMASK(EQF)	Field manipulation
COM	COM[,RW] RA,W[,RM]	Extended instructions
COMC	COMC RA,EQR [,RM]	Constant manipulation
COMU	COMU[,RW] RA,W[,RM]	Extended instructions
COMUC	COMUC RA,EQR [,RM]	Constant manipulation
CSHFL	CSHFL RA,EQR [,RM]	Constant manipulation
CSHFLD	CSHFLD RA,EQR [,RM]	Constant manipulation
CRSN	CRSN RA,EQ . EQ <16	Constant manipulation
CVALAS	S <- CVALAS(R/EQF/W/S/BQF)	Display control
DISL	DISL { {{ S / [*EQ] }} }	Display control

continued

Table 6-9. Summary of AAWRENCH MASM Procedures (cont.)

MASM Proc	Format	Type
DIVC	DIVC RA,EQR [,RM]	Constant manipulation
EQUF	EQUF EQ [,EQ [,EQ]]	Field manipulation
EVEN	EVEN . No parameters	Location counter handling
EXC	EXC RA,RM	Extended instructions
EXTRACT	EXTRACT W[,RM]	Extended instructions
FIELDLIKE	TV <- FIELDLIKE(W)	Field manipulation
GEN	GEN[,[*]RPT] VAL1[,VAL2]	Table and instruction generation
GENTAB	GENTAB[EQ[,EQ]] {[*]S[,EQF[,EQ]]/EQ[,EQF]}	Table and instruction generation
GINL	GINL 4*{{S}}	Table and instruction generation
JMPTBL	JMPTBL[,[S] [,EQ] [,EQ] [,EQ]] {EQR[,EQ]}	Table and instruction generation
JTBL	JTBL[,[EQ][,EQ]] RA,[*]EQR[,RM]	Extended instructions
LDK	LDK RA,EQ	Extended instructions
LEN	N <- LEN(EQF)	Field manipulation
LOAD	LOAD RA,W[,RM]	Extended instructions
LOADC	LOADC RA,EQR [,RM]	Constant manipulation
LSHFR	LSHFR RA,EQR [,RM]	Constant manipulation
LSHFRD	LSHFRD RA,EQR [,RM]	Constant manipulation
MASK	N <- MASK(EQF)	Field manipulation
MAXIMUM	N <- MAXIMUM({{EQ}})	Value manipulation
MAXVAL	N <- MAXVAL(EQF)	Field manipulation
MINIMUM	N <- MINIMUM({{EQ}})	Value manipulation
MOVE	MOVE[,RW[,RW]] W,RM W,RM	Extended instructions
MULTC	MULTC RA,EQR [,RM]	Constant manipulation
NOLIST	NOLIST S	Display control
ODD	ODD . No parameters	Location counter handling

continued

Table 6-9. Summary of AAWRENCH MASM Procedures (cont.)

MASM Proc	Format	Type
OPSTR	S <- OPSTR (3*{{[*] R/EQF/EQR }})	Display control
OPSTR$	S <- OPSTR$ (3*{{[*] R/EQF/W/S/BQF }})	Display control
PROCLIST	PROCLIST S / 5*{{EQ}} S EQ	Display control
RADD	RADD[,[*]RW] [*]RA,W[,RM]	Extended instructions
RBIT	N <- RBIT(EQF)	Field manipulation
RDEC	RDEC[,[*]RW] [*]RA,W[,RM]	Extended instructions
RELOC$	TV <- RELOC$(X)	Value manipulation
RINC	RINC[,[*]RW] [*]RA,W[,RM]	Extended instructions
RSUB	RSUB[,[*]RW] [*]RA,W[,RM]	Extended instructions
SBITS	SBITS RA,[*]EQF	Constant manipulation
SCREAM	SCREAM { {{ S / [*]EQ }} }	Display control
SET	SET[,[*] RW] W[,RM]	Extended instructions
SHOW	SHOW 4*{{S}}	Display control
STORE	STORE[,RW] [*]RA,W[,RM]	Extended instructions
STOREC	STOREC[,RW] EQ,W[,RM]	Extended instructions
SUB	SUB[,RW] RA,W[,RM]	Extended instructions
SUBC	SUBC RA,EQR [,RM]	Constant manipulation
SVC	SVC EQ	Constant manipulation
SZMIE	SZMIE [*]EQ,[*]RM	Extended instructions
TAB	S <- TAB(S,EQ)	Display control
TABL	TV <- TABL(X)	Value manipulation
TBEQUF	TV <- TBEQUF(X)	Value manipulation
TBITS	TBITS[,RW] RA,[*]EQF	Constant manipulation
TEST	TEST[,[*] RW] W[,RM]	Extended instructions
TREG	TV <- TREG(X)	Value manipulation
TWOS	EQ <- TWOS(EQ)	Value manipulation
WARNING	WARNING { {{ S / [*EQ] }} }	Display control

continued

Table 6-9. Summary of AAWRENCH MASM Procedures (cont.)

MASM Proc	Format	Type
WORD$	EQR <- WORD$(W)	Value manipulation
WORD/WD	N <- WORD(EQF)	Field manipulation
	N <- WD(EQF)	(WD is a shorter name.)
WORDLIKE	TV <- WORDLIKE(W)	Field manipulation
X	N <- X(EQ)	Value manipulation
ZBITS	ZBITS RA,[*]EQF	Constant manipulation
ZM2K	ZM2K . No parameters	Location counter handling
ZM64	ZM64 . No parameters	Location counter handling

6.2. AEXTPROC—Extended Utility MASM Procedures

The AEXTPROC utility element contains a number of additional MASM procedures you may find useful. See Table 6-1 for symbols used in this section.

6.2.1. Byte Field Manipulation

Fields may be defined as being part of bytes (as is the case with EQUF) rather than words. A field can be defined as any contiguous sequence of bits of minimum length 1, and of maximum length 16. Such a maximum permits the definition of wordlike fields or of other extended fields such as sequence counts.

However, to limit the complexity of manipulating arbitrary 16-bit fields, a field cannot extend over more than two contiguous bytes. Thus a 16-bit BEQUF can start only at the most significant bit of one byte (bit 7) and finish at the least significant bit (0) of the immediately succeeding byte. A 9-bit field may start anywhere within a byte, and extend through into the next byte.

Figure 6-1 illustrates the use of BEQUF with fixed protocol headers:

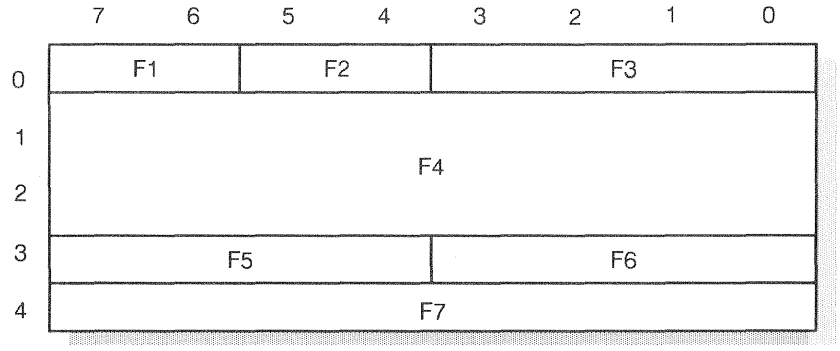


Figure 6-1. Fixed Protocol Header

The following defines the Figure 6-1 header:

```
F1   BEQUF 0,7,2
F2   BEQUF 0,5,2
F3   BEQUF 0,3,4
F4   BEQUF 1,7,16
F5   BEQUF 3,7,4
F6   BEQUF 3,3,4
F7   BEQUF 4,7,8
```

The format can be seen as EQUF; the main differences are that the offsets are byte-oriented and that the field lengths can appear to be incompatible with the left bit used.

To make use of these definitions, a register pair must be initialized to point to the first byte of the header. Once this is done, the LBF, SBF, and SMBF MASM procedures may be freely used.

Format

BEQUF *B*[,*LB*[,*LEN*]]

Parameters

B	byte offset
LB	leftmost bit of the field (default is 7)
LEN	length of the field in bits (default is 8)

If LB>7 or LEN>LB+9, an error is flagged and the default definitions are used.

The BEQUF defaults are such that if no LB or LEN specifications are given, the BEQUF generated is equivalent to a single byte at the offset specified.

Example

The F7 definition described earlier could have been written as follows:

F7 BEQUF 4

Function

In general, the format of the BEQUF functions is identical to the corresponding EQUF function.

Table 6-10. Byte Field Manipulation (BEQUF Functions)

MASM Proc	Format	Definition
BBIT	N<-BBIT(value)	Returns the leftmost bit (MSB) for the given value.
BBYTE	N<-BBYTE(value)	Returns the byte offset for the given value. The value must be a BEQUF or a simple numeric equate (without relocation).
BMASK	N<-BMASK(value)	Returns a 16-bit mask corresponding to the BEQUF passed as a value. If the BEQUF is simple (that is, does not extend over a byte boundary), the high byte of the mask is zero and the low byte is as might be expected. For a multiple-byte BEQUF, the mask is that resulting from the two bytes being considered as a word. The masked bits generally are clustered around the center of the word. For example, BMASK(F1) returns 0C0. BMASK(F3) returns 0F, BMASK(F4) returns 0FFFF.

continued

Table 6-10. Byte Field Manipulation (BEQUF Functions) (cont.)

MASM Proc	Format	Definition
BLEN	N<-BLEN(value)	Returns the length of the field.
BRBIT	N<-BRBIT(value)	Returns the rightmost bit (LSB) for the given value.
CBMASK	N<-CBMASK(value)	Returns the 16-bit one's complement of BMASK.
TBEQUF	TV<-TBEQUF(value)	Returns true(1) if the value is a BEQUF; otherwise, false(0).

.2. Load Operators

The LBF procedure is a single operation which loads a register from a field defined by BEQUF header.

Format

LBF[,*RW*] [*]*RA*,[*]*Q*,[*]*RM*

Parameter

RW	A work register (default is R0).
RA	A result register.
Q	BEQUF or a simple numeric equate.
RM	A pair of registers (RM,RM+1 : RM even) containing an Im-byte address.

The value contained in the field defined by Q and RM is loaded into the register RA. A work register may be necessary.

Condition	Result
Q is flagged	Any offset information within the BEQUF is ignored; that is, the offset is assumed to be zero. This can be used in conjunction with flagging RM to achieve increased efficiency of code generation.
Q is not flagged	The offset is added to RM+1 to enable access of the required byte.
RA is flagged	It is preserved across the MASM procedure call.
RM is flagged	The RM pair is not reset to its initial value; that is, any offset added into RM+1 is not subtracted on completion of the operation. This can result in substantial code savings. Use with care because redefinition of the fields could result in incorrectly generated code.
RM is not flagged	The RM pair is always set back to its original value.
RA is not flagged	The condition codes are set to correspond to the value loaded into RA. Note that in the general case, only Z or NZ are meaningful tests to perform, though 16-bit fields also correctly set the P/NP bit.
RA is flagged	The condition codes resulting from the load are not meaningful. This can result in saving a superfluous LR instruction.

Other conditions:

- If RW is used, it may not equal RA, RM, or RM+1.
- RA may not equal RM or RM+1.

3. Store Operators

Two store operators are currently defined as listed in Table 6-11. The parameter descriptions follow the table.

Table 6-11. Store Operators

MASM Proc	Format	Definition
SBF	SBF[,RW] [*] RA,*Q,[*]RM	Store byte field The SBF takes the contents of a register and stores it into the specified field.
SMBF	SMBF[,RW] RA,[*]RM £[Q,[*]W[,RX] £]	store multiple byte field The SMBF can build an entire protocol header and store it into a specified field. This is noticeably more efficient and should be used in all possible cases.

Parameters

RA For SBF, RA is a register from which the contents are stored. For SMBF, it is an intermediate storage register that must be defined.

RM A pair of registers (RM,RM+1 : RM even) containing an Im-byte address.

RW A work register (default is R0).

RX An optional index register.

Q BEQUF or simple numeric constant.

W EQUF or equate with or without relocation.

SBF conditions	Result
Q is flagged	Any offset information within the BEQUF is ignored; that is, the offset is assumed to be zero. This can be used in conjunction with flagging RM to achieve increased efficiency of code generation.
Q is not flagged	The offset is added to RM+1 to enable the required byte to be accessed.
RA is flagged	RA is preserved across the MASM procedure call.
RM is flagged	The RM pair is not reset to its initial value; that is, any offset added into RM+1 is not subtracted on completion of the operation. The same comments apply as for the LBF proc.
RM is not flagged	The RM pair is always set back to its original value.
Q is a simple numeric constant	The entire byte at offset given by Q**OFFFF is assumed to be written.

For SMBF, the combination *Q,W,RX* is repeated for every field set up. SMBF merges all such set-up requests and then optimizes the creation and storing away of values. The flag conditions and the constraints imposed on register choices are described as follows:

SMBF conditions	Result
W is flagged	The value to be inserted in the field is taken to be a constant; otherwise, it is assumed to define a fieldlike zone in memory.
RM is flagged	The RM pair is not reset to its initial value; that is, the offset added into RM+1 is not subtracted on completion of the operation.

Other conditions:

- RW may not equal RA, RM, or RM+1.
- RA may not equal RM or RM+1.
- RX may not equal RA or RM+1.

.4. Stack Handling MASM Procedures

Stack handling facilitates the writing of recursive or reentrant code and provides a convenient means for saving and restoring registers as you go. All stack operations require the use of a stack register. This register is defined as an equate to provide maximum flexibility.

The register can be redefined within the program, if necessary. To redefine the stack register, include AAWRENCH then insert the following where *n* is the required register (0..15):

STK$$ EQUR *n* .

The default register is R14.

The SETSTACK MASM procedure sets up a stack area and initializes the stack pointer. The PUSH$ and POP$ MASM procedures save and restore registers, respectively. The REMSTACK MASM procedure deallocates the stack. The CHKSTACK MASM procedure ensures that *n* words remain on the stack. Table 6-12 describes these procedures.

Table 6-12. Stack Handling MASM Procedures

MASM Proc	Format	Definition
CHKSTACK	CHKSTACK [stackbase],size	Generates in-line code to determine if a specified number or number of words remain on the stack. If not, a specification exception is generated.
PUSH$	PUSH$ registers	Pushes the values in the register or registers specified onto the stack.
POP$	POP$ registers	Pops to the top of the stack register values that have been pushed onto the stack.
RDSTK	RDSTK RA,W subroutine linkage pkg	Generates code that provides access to data pushed onto the stack. It does so without having to pop and then repush data. Note the CAUTION that follows Additional Discussion.

continued

Table 6-12. Stack Handling MASM Procedures (cont.)

MASM Proc	Format	Definition
REMSTACK	REMSTACK[,RW] [stackbase[,size]]	Deallocates a stack area allocated by SETSTACK. See Additional Discussion.
SETSTACK	SETSTACK[,RW] [*]stackbase[,[*]size[,<RSVD>]]	Initializes the stack pointer, STK$$. See Additional Discussion.

Parameters

stackbase — The address of the base of the stack. If not defined, the value used is that in the previous SETSTACK.

size — The number of words left on the stack.

registers

<registers>::=<register pair>/<register pair><blank><registers>

<register pair>::=<register>/<register><comma><register>

RA — The destination register.

RW — A work register.

W — Defines an offset, relative to the current stack top, where the data can be found.

<RSVD> — This optional parameter reserves words at the base of the stack area. Since this starts at constant address stackbase, it can be used as a dynamic table area. Checks are made to see that the area requested is sufficiently large to enclose the reserved area.

Additional Discussion

- When using PUSH$, the registers are saved from left to right; that is, the first mentioned are pushed the deepest.

  Example

```
PUSH$   R1,R4 R8              . Saves R1,R2,R3,R4, and R8
PUSH$   R13,R6                . Saves R13,R14,R15,R0,R1..R6
PUSH$   R4 R2 R9,R10          . Saves R4,R2,R9, and R10
```

- When using POP$, the order of restore is from right to left; that is, the last mentioned are popped first. This means that the same parameter list can be used for both PUSH$ and POP$.

  Example 1

```
        PUSH$      R1,R3     . Saves original R1 to R3
             .....
        POP$       R1,R3     . Restores R3 with original R3
                             . Restores R2 with original R2
                             . Restores R1 with original R1
```

  Example 2

```
        PUSH$      R1,R3     . Saves original R1 to R3
             .....
        PUSH$      R7        . Saves original R7
             .....
        POP$       R1,R4     . Restores R4 with original R7
                             . Restores R3 with original R3
                             . Restores R2 with original R2
                             . Restores R1 with original R1
```

Caution

The RDSTK MASM procedure is dangerous since it presupposes knowledge of what and how data is stored on the stack. Exercise great care when using this procedure. Remember that the stack pointer is not altered by this instruction.

- It is advisable to use the predefined variable, TOP$STACK, when using the RDSTK MASM procedure. The topmost (latest pushed) entry is found at TOP$STACK, the next highest at TOP$STACK-1, and the next highest at TOP$STACK-2, and so on (note the sign: -1, -2).
- When using REMSTACK, if stackbase is specified, size (default is 128) bytes are deallocated at stackbase. If the previous use of SETSTACK did not allocate any memory, this MASM procedure is a NOP. Otherwise, the memory previously allocated is deallocated. Three work registers are required for this, usually R1, R2, and R3, but they can be redefined by specifying RW. As for SETSTACK, RW must be odd and < 14. If defined, RW, RW+1, and RW+2 are used.
- When using SETSTACK, two types of initialization can take place, depending on whether the "*" option is taken. A SETSTACK stackbase sets the stack pointer to stackbase. This assumes the data area (read/write) is set up and visible.

  The alternative option, SETSTACK *stackbase, requests that a data area be allocated at address stackbase. In this case, two options are possible. Since allocation requires the use of three work registers, usually taken as R1, R2, and R3, they may be specified explicitly by defining RW. In this case, RW, RW+1, and RW+2 are used as work registers. RW must be odd and < 14. Similarly, the stack size can be explicitly defined. If this is not done, it is taken to be 128 bytes.

  Size is measured in bytes. If size is either 128 or 4096 bytes, non-subsegmented space is allocated. This can be overridden by prefixing size with an asterisk.

Example of SETSTACK

```
SETSTACK  *SDR9                 . Allocates granule at SDR 9
SETSTACK  STACK                 . STK$$ <- address(STACK)
SETSTACK,R13  *SDR29,256        . Allocates 256 bytes at SDR 29
```

5. Subroutine Linkage

The following MASM procedures provide a simple subroutine linkage package. MASM procedures are available to define entry and return points from subroutines, as well as the required CALL statements. The JLR instruction is used for calling and the link register is R13.

Note: *The existence of a stack is assumed.*

Table 6-13. Subroutine Linkage MASM Procedures

MASM Proc	Format	Definition
ENTRY	ENTRY[,S] [[RA,RM / RA]]	Defines the start of a subroutine. Only one entry point/routine may exist. The optional S parameter is used to suppress the automatic save of the link register. Registers may be saved on entry. These are restored by the RETURN MASM procedure.
GOSUB	GOSUB [*] EQR [,RM]	Passes control to a named subroutine. It is not required that the entry statement define the called routine; all that is necessary is that the same linkage is assumed.
RETURN	RETURN	Defines a point at which control is returned to the calling routine. Many such points may exist per routine. Any registers saved by the ENTRY MASM procedure are restored.

6.2.6. Table and Instruction Generation

The MASM procedures listed in Table 6-14 generate code or data.

Table 6-14. Table and Instruction Generation (AAEXTPROC)

MASM Proc	Format	Definition
GTABLE	GTABLE[,[EQ] [,EQ] [EQR [,EQF]]	Generates a table similar to GENTAB except that relocatable values can be used. There is no maximum table size.
LMREG	LMREG [,RW] [[RA]] [*EQR [,RM]/W[,RM]/RM]	Loads multiple registers from a variety ensuring the integrity of the load ordering.
STABLE	STABLE[,RW] [*]RA,RM [W[,RX]/*EQR[,RX]/RX]	Builds a table with or without clearing former contents from almost any type of source: registers, fields, constants. The code, though not optimal, is better than can be done by applying the LOAD/LOADC/ STORE MASM procedures. In particular, it can be used to build words containing several fields, with no intermediate stores being performed between successive loads.
TABLEDEF	TABLEDEF [[*]EQ/[*]S[,EQ[,EQ[,EQ]]]]	Generates a set of EQUFs that describe a table. With this MASM procedure, the individual fields within the table can be defined automatically.
TCLR	TCLR W[,RM]	Clears a field as set by TSET.
TSET	TSET[,<L> / <S>] [*]W[,RM]	Tests and sets a field with wait and retry if the value is already set. Two types of waits may exist: a short wait and a long wait of several milliseconds that requires process suspension.

7. AEXTPROC Procedure Summary

Table 6-15 summarizes the MASM procedures found in AEXTPROC.

Table 6-15. Summary of AEXTPROC MASM Procedures

MASM Proc	Format	Type
BBIT	N<-BBIT(value)	Byte field manipulation
BBYTE	N<-BBYTE(value)	Byte field manipulation
BEQUF	BEQUF B[,LB[,LEN]]	Byte field manipulation
BLEN	N<-BLEN(value)	Byte field manipulation
BMASK	N<-BMASK(value)byte field manipulation	
BRBIT	N<-BRBIT(value)	Byte field manipulation
CBMASK	N<-CBMASK(value)	Byte field manipulation
CHKSTACK	CHKSTACK [stackbase],size	Stack handling
ENTRY	ENTRY[,S] [[RA,RM / RA]]	Subroutine linkage
GOSUB	GOSUB [*] EQR [,RM]	Subroutine linkage
GTABLE	GTABLE[,[EQ]] [,EQ] [EQR [,EQF]]	Table and instruction generation
LBF	LBF[,RW] [*]RA,[*]Q,[*]RM	Load operator
LMREG	LMREG [,RW] [[RA]] [*EQR [,RM]/W[,RM]/RM]	Table and instruction generation
POP$	POP$ registers	Stack handling
PUSH$	PUSH$ registers	Stack handling
RDSTK	RDSTK RA,W subroutine linkage pkg	Stack handling
REMSTACK	REMSTACK[,RW] [stackbase[,size]]	Stack handling
RETURN	RETURN	Subroutine linkage
SBF	SBF[,RW] [*]RA,*Q,[*]RM	Store operator
SETSTACK	SETSTACK[,RW]	Stack handling

continued

Table 6-15. Summary of AEXTPROC MASM Procedures (cont.)

MASM Proc	Format	Type
SMBF	SMBF[,RW] RA,[*]RM [Q,[*]W[,RX]]	Store operator
STABLE	STABLE[,RW] [*]RA,RM [W[,RX]/*EQR[,RX]/RX]	Table and instruction generation
TABLEDEF	TABLEDEF [*] S [[*]EQ/[*]S[,EQ[,EQ[,EQ]]]]	Table and instruction generation
TBEQUF	TV<-TBEQUF(value)	Byte field manipulation
TCLR	TCLR W[,RM]	Table and instruction generation
TSET	TSET[,<L>/<S>] [*]W[,RM]	Table and instruction generation

3. AASTRPRC—Structured MASM Procedures

The following describes program-structuring MASM procedures that assist implementation and improve clarity of programs by using higher level control statements in conjunction with assembler coding.

Do not confuse these procedures with a compiler. Very little optimization or syntax checking is performed. You should inspect the generated output for register conflicts.

For clarity, wherever the name of a construct is to be understood as the AASTRPRC keyword, it is printed in uppercase.

The following structure types are available:

- IF (with ELSIF in fix groups and ORIF and ANDIF continuation tests)
- FOR
- LOOP
- CASE

1. Initialization Before Structure Statement

The omnibus element containing the MASM procedures is called AASTRPRC. It must be included at the head of each program.

```
$INCLUDE 'AASTRPRC'
```

The MASM procedures require an initialization call before the first structure statement is used.

Format

STRPRC *<list>*, *<base>*, *<jump-type>*

Parameters

<list>	If this parameter is NOLIST, the generated code is displayed as comments following the statements. Advisable for final released versions but not for test runs.

<base>	If specified in conjunction with the *jump-type*, *<base>* is used as the index register on jump instructions.
<jump-type>	If specified as LONG, all jump instructions generate long rather than local jumps.

6.3.2. Jump Instructions

Unless the LONG parameter is placed on the STRPRC call, the structure MASM procedures normally generate local jump instructions. If the destination is out of range, then the usual E flag is out of range. The usual E flag and comment are printed.

To force a long format jump to be generated by a particular MASM procedure, code LONG following the MASM procedure keyword. For example:

IF *<condition>*

would be changed to:

IF,LONG *<condition>*

If the range of a structure is beyond the extent of a local jump, the structure is probably too extensive for clear understanding and further use of subroutines is desirable.

6.3.3. Register Usage

The IF and UNTIL constructions may require one or more work registers. These constructions always use R0 if they require one, and R0 and R1 if they require two.

4. IF Structure

The IF structure permits selection of one from a series of alternatives according to the results of condition tests. The permitted format of the condition tests is described in 6.3.8.

The following is an allowable IF structure:

<IF-conditions>
<Instruction-sequence>
<ELSIF-conditions>
<Instruction-sequence>
ELSE
<Instruction-sequence>
ENDIF

The *<IF-conditions>* take the following form:

IF *<condition>*
<Instruction-sequence> repeated as required
ANDIF| ORIF *<condition>*

The meaning is best understood by regarding the ANDIF and ORIF keywords as logical operators connecting the individual Boolean conditions, with ANDIF associating (according to conventional operator precedences) preferentially to ORIF. The interpolated instruction sequences are best limited to the setup required for an immediately succeeding test; otherwise, the whole conditional sequence can become unintelligible.

An *<ELSIF-conditions>* follows the same rules as the *<IF-conditions>*, but is introduced with the keyword ELSIF instead of IF.

The formal rules for transfer of control within the IF structure follow:

- IF, ANDIF, and ELSIF - If the test fails, control passes to the next ORIF in this conditional group, if one exists; else to the ELSIF introducing the next alternative of the structure, if one exists; else to the ELSE introducing the default alternative for the structure, if one exists; else to the ENDIF that terminates the structure. If the test passes, control is passed to the next statement in sequence.

- ORIF - If the test passes, control is passed to the set of code lines representing the alternative for this conditional group; this is the statement after the last ANDIF or ORIF of the group. If the test fails, follow the same procedure as IF, ANDIF, and ELSIF.

  A conditional expression at the same nested level may not be coded between the ELSE and the ENDIF. Any number of instructions may be coded between the statements.

6.3.5. FOR Structure

A FOR and an ENDFOR pair of statements may bound instruction sequences that repeat a known number of times. This structure may be nested, but each FOR statement must have a matching ENDFOR for correct code generation. The following is an allowable structure:

FOR <*register*> EQ <*start*> TO <*end*>
    <*Instruction-sequence*>
    ENDFOR

where

<*register*>	The register used in the loop control.
<*start*> and <*end*>	Constants, registers, or fields defined by <EQUF>. The values determine the number of times the sequence is repeated.

The register is used as a work register, not as a loop index. During the first execution of the sequence, the register is equal to the difference between <*end*> and <*start*> less one. Each iteration causes the index value to be decremented until the last pass, when it reaches zero and control falls to the statement following the ENDFOR.

The FOR structure may be prematurely terminated before the specified number of iterations has completed a call on FORCEOUT.

    FORCEOUT

The current iteration is the last one, and control drops through to the next statement at the next ENDFOR. FORCEOUT always acts on the innermost active FOR structure.

Note: *FORCEOUT does not generate a jump out of the structure.*

6. LOOP Structure

Instruction sequences that repeat until a certain condition is met may be bound by a LOOP/ENDLOOP pair of statements containing an *<UNTIL-conditions>* that specifies a condition to end the repetition. The format of the condition tests specified in the *<UNTIL-conditions>* is defined in 6.3.8.

The allowable structure is:

LOOP
<Instruction-sequence>
<UNTIL-conditions>
< Instruction-sequence>
ENDLOOP

The *<UNTIL-conditions>* follow the same rules as *<IF-conditions>*, but have their own set of keywords: UNTIL, ANDUNTIL, and ORUNTIL. The destination for any test or group of tests that passes is the end of the loop.

6.3.7. CASE Structure

The CASE structure permits one selection from a set of equally weighted alternatives by the value of a variable. The following is the allowed structure:

CASENTRY .....
Optional code given control if the selection value is not defined on any of the CASE alternatives.

CASE *v*..... Code to execute if selection value is one of those defined on this CASE statement.

.
.
.

ENDCASE

The individual MASM procedures are:

CASENTRY Defines the start of the structure and the selection variable. The format of the MASM procedure call is:

CASENTRY <*r n*>[,<*source*>]
[LIMIT[,*minimum*,*maximum*]]

where

<*r n*> The work register, which cannot be R0.

<*source*> An optional specification that defines the source of the selection information. Any expression legal in the corresponding fields of the LOAD MASM procedure can be used. If this parameter is omitted, the selection information is assumed to be already in the working register.

LIMIT — An optional keyword that specifies that a range check is required on the selection value to ensure that it is within the range defined by the CASE alternatives.

If omitted, no range check is generated.

If specified, it is advisable to also specify the minimum and maximum values explicitly so that code for tests can be improved: in their absence, a CK followed by an LJ test is generated for each limit.

CASE — This MASM procedure defines the start of a CASE alternative and the set of selection values that gives it control. The format of the CASE MASM procedure call is

CASE <*value*>[,<*value*>]...

where

<*value*> — A selector for the case alternative. The values must be fixed-point binary, in the range 0 through +32767 (16 bits, positive), and must be defined at the pass 1 encounter.

ENDCASE — Defines the end of the case structure. No parameters are required on the MASM procedure call line.

8. Conditional Tests

The IF and LOOP structures are controlled by Boolean tests, referred to earlier as conditions. The two types of conditions are CPA condition code test and comparisons.

CPA Condition Code Tests

If the condition specified is a single parameter from the following list, a test on the condition code is generated. This is useful, for example, when a test of the attribute of a variable that has just been loaded to a register is desired.

The allowed abbreviations are:

EQ	Equal
NE	Not equal
LT	Less than
LE	Less than or equal
GE	Greater than or equal
GT	Greater than
CR	Carry
OV	Overflow

Comparisons

If more than one parameter appears in a *<condition>*, then it must be an arithmetic expression giving a Boolean result:

<value> *<relational-operator>* *<value>*

where

<value> May be one of the following:
- A register
- A bit in a register (test EQ or NE 0 only)
- A constant
- A field name defined by an <EQUF>

<relational-operator>

May be one of the following:

EQ	Equal
NE	Not equal
LT	Less than
LE	Less than or equal
GE	Greater than or equal
GT	Greater than
DEQ	Double-length equal
DNE	Double-length not equal
DLT	Double-length less than
DLE	Double-length less than or equal
DGE	Double-length greater than or equal
DGT	Double-length greater than

If the relational operator is flagged by an asterisk (*GT, for example), the values compare unsigned.

Double-length comparisons have the following additional restrictions:

- A register operand must be even.
- A storage operand must be on an even address boundary (or be indexed so that this is potentially possible).
- Double-length words may not be compared to a partial-word field.
- Double-length comparisons may not be made with constant operands.

The *<IF-conditions>* has only one parameter and it is not one of the condition code testing keywords, the keywords NE 0 are assumed.

For a test on a bit in a register to be specified, the register name is qualified by the bit position.

Example

```
IF    R9,1 NE 0
```

This example tests if bit 1 of R9 is set. Bit 0 is the least significant bit.

Section 7
System Service Calls (SVCs)

This section describes the following:

- The service mechanism
- Run services
- Communications Processor Architecture (CPA) services
- Dictionary services
- Dispatch services
- File manager services
- Instrumentation services
- Inter-program message (IPM) services
- Line module services
- Port processor (PP) services
- System data format (SDF) record services
- Telcon tailoring services
- Program product services

System services are provided by the operating system at run time to programs in each of the above categories. To maintain the protection afforded by CPA, only DCP/OS procedures may be run in privileged mode. If programs need to manipulate CPA structures to create segments and extra queues, a system service call (SVC) instruction is used to invoke a privileged DCP/OS service. The following subsections specify the function and entry/exit conditions for every system service call.

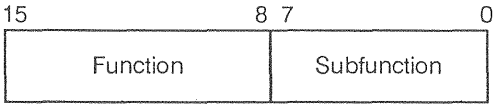
7.1. The SVC Mechanism

The SVC instruction may be used to request a service either from the operating system or from a separate program that is active and has registered the requested service.

The service is invoked as follows:

SVC <service code>

where the service code is a 16-bit code defined as follows:



Valid function codes for external services are (hex) 01-7F. Function codes 80-8F are reserved for operating system services. The subfunction may or may not be significant, depending on the function. The entire service code is automatically passed to the service procedure in R0 (that is, the caller does not have to explicitly load R0).

The following subsections specify the service codes and entry and exit conditions for all available operating system services.

Any procedure in a user program may be registered as a service through the operating system service CPA$ASVC. A registered service may be deregistered, either to block further calls to the service or to abort the service for all current users, through the CPA$RSVC service. A user program may get the current status of a specific service by using the CPA$GSVC service, described in this section.

A program calling a service procedure can be terminated by the operator while in the midst of the service. While the operating system automatically cleans up system tables used by the terminating program, such as the service program's lock table, it cannot reinitialize data areas local to the service program. Plan for this contingency by designing the service program so that it will survive external termination of the calling program.

All service and function codes (SVC arguments) are found in the definition element AASERVDEF. See Appendix C. All error codes can be found in the *DCP Series DCP/OS Operations Reference Manual* (7831 5702).

.1a. Extended SVC Use

The SVC instruction calls DCP/OS services, such as PRINT$ or CPA$CQUE. It can also call a procedure running in another program if the procedure is registered as a service procedure.

To register a procedure in a user program as a service procedure, so that it can be called and executed by an outside program, define a service code for the procedure. This is an 8-bit code in the range 01 to 7F. You can choose any code as long as it is not being used by another service procedure. For example, you can define the service code:

```
SVC$CODE   EQU   02400 . The service code is 024.
```

Note that you should define the service code in the upper byte of a full word, since this is how it is called. The lower byte can contain an 8-bit parameter.

As an option, you can register for notification whenever another program on the DCP is terminated. This service is provided so that the service provider knows when to clean up the environment of a task. The PC$PKILL service is called to register for notification.

Register the service procedure by invoking the CPA$ASVC service. The required parameters are:

R1 = GPLx of the service procedure
(Do not forget to add the service procedure to the GPL of the procedure that registers it.)
R2-R3 = (optional) 4-character name for the service (or 0)
R4 = service code in the format described above (upper byte)

When you want to use the service you need to:

1. Set up any parameters in registers and SDRs.
2. Call the service procedure using the SVC instruction, as follows:

```
SVC   SVC$CODE . Execute service
```

Cautions

- A Service Procedure must be able to handle multiple callers at the same time.

  There is no limit on the number of tasks that can call a registered service procedure and no restrictions on the runs that may call it. Any data or code that cannot handle multiple, simultaneous tasks should be protected by locks or a queued interface.

- Be prepared for the possibility of the calling program being killed in the middle of the service.

  Another user can terminate a program or run while using your service. DCP/OS clears LOCKs and attempts minor cleanup, but it cannot clear test-and-set bits or remove leftover queued items. The service procedure must be designed to handle these and similar contingencies. If you register for notification by DCP/OS whenever another program is killed and keep track of which runs are using your service, you can clean up any loose ends when a calling program aborts.

- Your extended service may not complete successfully.

  If the program providing the service is killed, DCP/OS returns an error code to all current users of the service. DCP/OS cannot guarantee that the environment has not been modified (for instance, SDRs can be present that were not present prior to the SVC call). You should be prepared for this. Ideally, the author of the service will document SDRs, registers used and any other effects that might cause problems for you.

- Do not register your service procedure if you do not want anyone to call it.

  The CPA$RSVC service revokes the registration of a currently registered service (if it is called by the same run that registered it).

- Define error codes to let the callers of your service know when something goes wrong. The usual convention is to return a negative value in R0 when an error occurs.

Pascal-coded SVC Procedures

Write service procedures in Pascal if they are to be called by other Pascal programs only. Do not try to call a Pascal service from a MASM procedure. The Pascal runtime system uses registers and segment descriptor registers (SDRs) that are invisible to the programmer and there are almost certain to be fatal conflicts when the two program environments are mixed.

A Pascal service must not be a TASK or PROGRAM, since either of these calls will attempt to reinitialize the internal stack and destroy the caller's stack. Do not use global variables in the service procedure and avoid getting space from the heap (using NEW, for example) unless you release it before the end of the procedure.

This page is intentionally left blank.

'.2. Run Services

Services are provided for runtime interaction with DCP/OS resources (for example, workstation dialog). The SVC mechanism invokes every run service. The function code definitions are found in AASERVDEF. (*See* Appendix C.)

The run service calls destroy user registers R0 through R5. The run service calls preserve all other user registers, except as stated in the individual service specifications.

Example

```
LOADC     R1,AV$MCT          . Point to message MCT
SVC       PRINT$             . Go display the message
```

'.2.1. COM$—Console Output Message

This service broadcasts a message to all console mode workstations. The message must be provided in a message control table (MCT) in standard DCP/OS message format.

Call	Entry	Exit
SVC COM$	R1 = virtual address of MCT	R0 = 8xxx error R0 = 0 successful

'.2.1a COMS$—Console Output Message to Status Line

This service broadcasts a message to all console mode workstations. The message is written to the console status line (currently defined as line 24). The previous contents of this line are erased. If PRNT is active, this message is not sent to the hardcopy printer. The message must be provided in a message control table (MCT) in standard DCP/OS message format, should not exceed 70 characters, and should not contain any screen control characters.

Call	Entry	Exit
SVC COMS$	R1 = virtual address of MCT R2 = 0 do not prefix Runid to message ≠ 0 prefix callers Runid to message	R0 = 8xxx error R0 = 0 successful

.2.2 COMW$—Console Output Message and Wait for Response

This service broadcasts a message to all console mode workstations and suspends the calling run until a response is received. The message must be provided in a message control table (MCT) in standard DCP/OS message format and must not exceed 60 characters.. The response is returned in the same MCT and is also in standard DCP/OS message format.

Call	Entry	Exit
SVC COMW$	R1 = virtual address of MCT	R0 = 8xxx error R0 = 0 successful R1 = virtual address of MCT

'.2.3 CSF$—Issue Control Statement

This service issues certain messages to the system manager on behalf of the calling program. The message simulates workstation input and must be in standard system message format. Only demand mode and demand mode bypass commands are allowed.

Call	Entry	Exit
SVC CSF$	R1 = virtual address of MCT	R0 = 8xxx error R0 = 0 successful

'.2.4 DATE$—Get Current System Date

This service returns the current date.

Call	Entry	Exit
SVC DATE$	None	R0 = 8xxx error R0 = 0 successful R2 = year (1900 = 0) R3 = month (1-12) R4 = day (1-31)

7.2.5. ERR$—Terminate Program with Error Code

This service terminates the calling program with a user-specified error code and frees up all in-use resources.

Call	Entry	Exit
SVC ERR$	R1 = user defined error code	None (The program is dead)

7.2.6. EXIT$—Terminate Program

This service terminates the calling program and frees up all in-use resources.

Call	Entry	Exit
SVC EXIT$	None	None (The program is dead)

7.2.7. FACMSG$—Expand File Manager Error Code

This service fills in a text message at the specified address corresponding to the supplied file manager error code. At least 40 bytes must be available at the specified position (R10,11) for the message to be inserted.

Call	Entry	Exit
SVC FACMSG$	R1 = error code R10 = virtual address of message R11 = byte offset to start of message	R0 = 0 successful R11 = updated to end of insert Destroys R0-R7

2.8. GETC$—Get Run Control Word

This service returns the current run control word.

Call	Entry	Exit
SVC GETC$	None	R0 = 0 (always successful) R1 = run control word

2.9. INFO$—Get Run Information

This service retrieves information from the run control table entry pertinent to the caller's run ID.

Call	Entry	Exit
SVC INFO$	None	R0 = 8xxx error R0 = 0 successful R1 = run flags (RC$FLAG) R2-R4 = run name R6 = run ordinal number R7 = run sequence number R8-R10 = project-ID (RC$QUAL) R11-R13= entered run name (RC$NAMO) see below

Entered run name is the run name entered on the @RUN card. If, however, a run by this same name already exists, an altered name is generated by DCP/OS. This altered name is found in R2-R4.

2.10. LINFO$—Get Load Information

This service retrieves information concerning the boot time activities such as the load path and load device.

Call	Entry	Exit
SVC LINFO$	None	R0 = 8xxx error R0 = 0 successful R2 = DCP serial number R3 = DCP type R4 = boot keys R5 = load path R6 = loaded system number R7 = load device R8 = OS level/OS version R9 = OS update

7.2.11. LOG$—Log Message

This service logs the message in the system log file (SYS$*SYSLOG). The caller run-id, as well as the current date and time, are prefixed to the message.

The virtual address of the message must not be in the first four SDRs of the caller's visibility because these SDRs are architecturally cleared upon invoking the service.

Call	Entry	Exit
SVC LOG$	R1 = virtual address of memory R2 = word count of message (can be up to 118 words) R3 (upper) = log type LOG$USER = user defined format LOG$STAT = system statistics LOG$JCL = JCL log message R3 (lower) = user defined format type or log subtype	R0 = 8xxx error R0 = 0 successful Destroys R0-R5

7.2.12. PRINT$—Send Message to Current Output Stream

This service queues a message (in MCT/buffer standard message format) to the current output stream.

If an illegal message type is present (in MH$MTYP), then this field is set to MH$MCO.

Call	Entry	Exit
SVC PRINT$	R1 = virtual address of MCT/buffer MH$MTYP = message type MH$MCO = scrolled output MH$MCOT = transparent output MH$DBO = message byte offset MH$DBC = message byte length	R0 = 8xxx error R0 = 0 successful

.13. PRTCN$—Set Workstation Print Control

This service sets the input/output (I/O) characteristics according to the calling parameter mask.

Call	Entry	Exit
SVC PRTCN$	R1 = flags to set (bits 15-8) = flags to clear (bits 7-0) bit 15/7 = escape output bit 10/2 = data mode bit 9/1 = escape input	R0 = 8xxx error R0 = 0 successful

.14. RAQUAL$—Read Assumed Qualifier

This service reads the assumed qualifier from the control table. The assumed qualifier is set or cleared by the @QUAL command. For information on the @QUAL command, see the DCP/OS Operations Reference Manual (UP-11541).

Call	Entry	Exit
SVC RAQUAL$	None	R2-R4 = assumed qualifier

7.2.15. RDQUAL$—Read Dump File Qualifier

This service reads the dump file qualifier from the control table.

Call	Entry	Exit
SVC RDQUAL$	None	R2-R4 = dump file qualifier

7.2.16. READ$—Read Image

This service reads a message from the current input stream. On the first call to READ$, the returned message is the INFOR$ packet corresponding to the command line. On subsequent calls to READ$, the messages are simple data strings. The returned message is in standard message format. If the message type is MH$MEOF, then the program should clean up and terminate with EXIT$.

Call	Entry	Exit
SVC READ$	R1 = virtual address for MCT/buffer	R0 = 8xxx error R0 = 0 successful MH$MTYP = message type MH$MCI = input MH$MCIT = transparent input MH$MCIF = function key MH$MADD = add stream data MH$MEOF = end-of-file indicator MH$DBO = message byte offset MH$DBC = message byte length

7.2.17. READT$—Read Image (Transparent)

This service reads an image in the same manner as READ$ except that no solicit (or implied scroll) is output.

Call	Entry	Exit
SVC READT$	R1 = virtual address for MCT/buffer	R0 = 8xxx error R0 = 0 successful

2.18. READW$—Read Image (from Workstation)

This service reads an image in the same manner as READ$ except that it bypasses any active add stream and always solicits input from the workstation. If a program running in batch mode issues this function, it is processed as a READ$.

Call	Entry	Exit
SVC READW$	R1 = virtual address for MCT/buffer	R0 = 8xxx error R0 = 0 successful

2.19. RINFO$—Get Run Information for a Specified Run

This service retrieves information from the run control table entry pertinent to the specified run-ID.

Call	Entry	Exit
SVC INFO$	R1 = run-ID	R0 = 8xxx error R0 = 0 successful R1 = run flags (RC$FLAG) RF2-R4 = run name R6 = run ordinal number R7 = run sequence number R8-R10 = project-ID (RC$QUAL) R11-R13 = entered run name (RC$NAMO)

The entered run name is the run name entered on the @RUN card. If, however, a run by this same name already exists, an altered name is generated by DCP/OS. This altered name is found in R2 through R4.

'.2.20. RQUAL$—Read Project-ID (Default Qualifier)

This service reads the project ID (default qualifier) from the RC$QUAL field of the run control table.

Call	Entry	Exit
SVC RQUAL$	None	R2-R4 = project-ID

'.2.21. SETC$—Set Run Control Word

This service sets the current run control word to the specified value.

Call	Entry	Exit
SVC SETC$	R1 = new value for RCW	R0 = 8xxx error R0 = 0 successful

'.2.21a TDATE$ —Return Date and Time

This service returns the current real-time clock (RTC) value and the date.

Call	Entry	Exit
SVC TDATE$	None	R0-R1 = RTC value R2 = Year (19xx) R3 = (upper) Month = (lower) Day

.2.22. TREAD$—Type and Read Image

This service displays a message to the workstation in the same manner as PRINT$ and then reads an image in the same manner as READ$. The TREAD$ service uses the same virtual address for both input and output messages.

Call	Entry	Exit
SVC TREAD$	R1 = virtual address for MCT/buffer (in and out)	R0 = 8xxx error R0 = 0 successful

.3. CPA Structure Services

The DCP/OS provides services to manipulate entries in the queue list (QL), gated procedure list (GPL), and procedure segment table (PST). It also provides services to create queue lists, link areas, queues, and so on.

The SVC mechanism invokes every CPA service. The function code values are found in AASERVDEF. (See Appendix C.)

The CPA service preserves all user registers, except as stated in the individual service specifications.

Example

```
LOAD      R1,myseg          . Pick up segment number
LOADC     R2,pstent         . Get PST index
SVC       CPA$APST          . Go add segment to PST
```

.3.1. CPA$AAQL—Add Queue to Alternate Queue List (PP)

This service adds a queue to a queue list of a PP alternate environment.

Call	Entry	Exit
SVC CPA$AAQL	R1 = queue number (QN) R2 = port number R3 = get,put,arm (bits 15,14,13) PP QLx (bits 12-0) R4 = AEL index	R0 = 8xxx error R0 = 0 successful

.3.2. CPA$AGPL—Add PN to GPL

This service adds a procedure to the GPL list of the calling procedure. The specified procedure must be a user procedure number (PN).

Call	Entry	Exit
SVC CPA$AGPL	R1 = PN to add R2 = GPLx	R0 = 8xxx error R0 = 0 successful

7.3.3. CPA$APPQL—Add Queue to QL (PP)

This service adds a queue to the queue list of a PP. The use count of the added queue is incremented. If this new entry overwrites an old entry, the use count of the previous queue is decremented.

Call	Entry	Exit
SVC CPA$APPQL	R1 = QN R2 = port number R3 = bits 15,14,13 - get,put,arm = bits 12-0 - QLx	R0 = 8xxx error R0 = 0 successful

7.3.4. CPA$APST—Add SSN to PST

This service creates a PST entry, giving access to the specified segment. If the segment being added is dynamic, its use count is incremented. If the new entry overwrites an old entry and the old segment is dynamic, its use count is decremented.

Call	Entry	Exit
SVC CPA$APST	R1 = SSN R2 = PSTx	R0 = 8xxx error R0 = 0 successful

7.3.5. CPA$AQL—Add Queue to QL (PN)

This service adds a queue to the queue list of the calling procedure. The use count of the queue is incremented. If the new entry overwrites an old entry, the use count of the previous queue is decremented.

Call	Entry	Exit
SVC CPA$AQL	R1 = QN R2 = bits 15,14,13 - get,put,arm = bits 12-0 - QLx	R0 = 8xxx error R0 = queue type successful

.6. CPA$ASPT—Add Segment to Procedure Table Entry

This service adds an SSN to one of the four initial visibility segment descriptors (SDs) in a procedure table (PT) entry. The current PT entry must be zero.

Call	Entry	Exit
SVC CPA$ASPT	R1 = SSN R2 = PN (0 = caller) R3 = SD number (0 to 3)	R0 = 8xxx error R0 = 0 successful

Note: *If the specified PN is the caller, the added segment will be visible upon return from the service.*

.7. CPA$ASVC—Add Extended SVC Procedure

This service allows a program to register a procedure as an extended service (that is, one that may be called by another program). A record is added to the service vector table (SVT) and marked valid.

The service name is optional. It is used for display and as an optional parameter by the debug C command (see 5.6.3). Any name submitted must consist of an alphabetic character followed by up to three alphanumeric or special characters ("$", "-", "@"). Lowercase alphabetic characters will be converted automatically to uppercase. If the name is less than 4 characters the trailing characters must be blanks.

By convention, the service code is passed to the service procedure in R0, and error codes are passed back to the caller in R0.

The service code is a 16-bit parameter defined as follows:

Upper 8 bits = 01-7F to uniquely represent this program/procedure combination

Lower 8 bits = ignored by the OS. These bits will be passed to the service procedure in R0 and may be used as a parameter.

Call	Entry	Exit
SVC CPA$ASVC	R1 = (14-0) GPLx (15) Force PN5 event if set R2-R3 = 4 ASCII characters with the service name or 0 if no name is desired R4 = (15-8) Service code (upper 8 bits) (7-0) Ignored	R0 = 8xxx if error R0 = 0 if successful

7.3.8. CPA$CAE—Create Alternate Environment Entry

This service creates an entry in the CPA alternate environment list (AEL) to register an alternate environment for the specified port. The caller selects whether to register an alternate SD, an alternate queue list, or both.

If you register an alternate SD, you must supply its virtual address in R3 on a 4K byte boundary. If the address is of an allocated buffer, the buffer is removed from the caller's visibility. If the address is of a system segment, the use count is incremented.

If you register an alternate queue list, R4 must contain the length of the required queue list. This service creates the queue list.

The alternate environment index passed in R2 determines the offset of the alternate environment entry in the AEL of the port. This service allows an alternate SD or alternate queue list to be registered even though the other may be present.

Call	Entry	Exit
SVC CPA$CAE	R1 = port number R2 = AEL index (0-15) R3 = virtual address (or zero) R4 = queue list size (or zero)	R0 = 8xxx error R0 = 0 successful

3.9. CPA$CAET—Create Alternate Environment Table

This service creates an entry in the CPA alternate environment table (AET) for the specified port and reserves a fixed number of zeroed entries in the AEL for subsequent assignment as alternate environments. The port must not be active when this service is invoked.

Call	Entry	Exit
SVC CPA$CAET	R1 = port number R2 = number of entries in AEL (1 to 16)	R0 = 8xxx error R0 = 0 successful

.10. CPA$CKSD—Check Segment Descriptor

This service returns a bit map corresponding to the space allocated under the given virtual address. The space may be contiguous or subsegmented. Each bit in the returned map represents a granule (128 bytes).

Call	Entry	Exit
SVC CPA$CKSD	R1 = virtual address	R0 = 0 not present = 1 present, contiguous = 2 present, subsegmented R2,R3 = 32 bit map

.11. CPA$CLA—Create Link Area

This service creates a link area of a given size for the specified procedure. If the specified procedure is zero, the caller is assumed.

Call	Entry	Exit
SVC CPA$CLA	R1 = PN (0 = caller) R2 = link area size	R0 = 8xxx error R0 = 0 successful

7.3.12. CPA$CLONQ—Clone a Queue

This service creates the requested queue and returns the queue number (QN). The created queue is an exact copy of the queue specified by R1.

Call	Entry	Exit
SVC CPA$CLONQ	R1 = QLx of queue to clone	R0 = 8xxx error R0 = QN successful R1 = (15-8) queue type = (7-0) queue mode = (7-0) queue size (upper) [link-list-q] R2 = (15-8) queue threshold = (7-0) queue size = (15-0) threshold [link-list-q] R3 = (15-8) Qx = (7-0) access type = (7-0) queue size (lower) [link-list-q] R4 = queue device address R5 = queue lower SAI (R1 parameter) R6 = queue upper SAI (PN to schedule)

7.3.13. CPA$CPPQL—Create PP Queue List

This service creates the specified size of queue list and attaches it to the specified PP (interface control table, or ICT entry).

Call	Entry	Exit
SVC CPA$CPPQL	R1 = port number R2 = number of entries (QLn)	R0 = 8xxx error R0 = 0 successful

7.3.14. CPA$CQUE—Create a Queue

This service creates the requested queue and returns the assigned QN. The queue mode must be zero for a link list queue.

Call	Entry	Exit
SVC CPA$CQUE	R1 = (15-8) queue type = (7-0) queue mode = (7-0) queue size (upper) [link-list-q] R2 = (15-8) queue threshold = (7-0) queue size = (15-0) queue threshold [link-list q] R3 = (15-8) Qx = (7-0) undefined = (7-0) queue size (lower) [link-list-q] R4 = queue device address R5 = queue lower SAI (R1 parameter) R6 = queue upper SAI (PN to schedule)	R0 = 8xxx error R0 = QN successful

.15. CPA$CSEG—Create a Segment

This service creates a new system segment from allocatable space passed by the requestor. The space definition is changed from allocatable to transient, and a system segment number (SSN) is defined for the segment. If successful, the space is removed from the caller's visibility.

The created segment is a forced memory-resident segment even though the architectural space type is transient.

The caller may optionally specify that a previously reserved SSN (see CPA$RSSN) be used when creating the segment, by setting bit 9 of R2. In this case, the SSN is passed into R4.

Call	Entry	Exit
SVC CPA$CSEG	R1 = segment virtual address R2 = segment mode bits (bits 13 to 10) bit 13 = 1 CP execute bit 12 = 1 PP execute bit 11 = 1 Data read bit 10 = 1 Data write bit 9 = 1 reserved SSN R4 = SSN (optional)	R0 = 0 error R0 = SSN successful

7.3.16. CPA$CVIS—Check SDR Visibility

This service returns the SD mode bits and length pertaining to the specified virtual address.

Call	Entry	Exit
SVC CPA$CVIS	R1 = virtual address	R0 = (15-8) mode bits = (4-0) length bits

7.3.17. CPA$DAE—Delete Alternate Environment Entry

This service deletes an entry (or part of an entry) in the AEL for the specified port. The caller elects whether to delete an alternate SD, an alternate queue list, or both.

If an alternate SD is deleted and the SD is an allocated buffer, that buffer is deallocated. If the SD describes a system segment, the use count is decremented.

If an alternate queue list is deleted, the use counts of all queues in the queue list are decremented.

The alternate environment index passed in R2 determines the offset of the AEL entry for the specified port.

Call	Entry	Exit
SVC CPA$DAE	R1 = port number R2 = AEL index (0-15) R3 = function (1 = delete SD) (2 = delete QL) (3 = delete both)	R0 = 8xxx error R0 = 0 successful

3.18. CPA$DAET—Delete Alternate Environment Table

This service deletes an entry in the AET. Delete the corresponding entries in the AEL before invoking this service.

Call	Entry	Exit
SVC CPA$DAET	R1 = port number	R0 = 8xxx error R0 = 0 successful

3.18a CPA$DASVC — Disable SVC Automation

This service disables microcoded automation of the next SVC instruction (after this SVC completes). Disabling remains in effect only for a single SVC call unless the run is in CATCH mode through the debugger. The Telcon module TELSERV calls this service if the DCP microcode includes the automated SVC instruction. If you use a tailored version of Telcon (level 8R1 or lower) and do not want to eliminate TELSERV, this call is necessary.

Call	Entry	Exit
SVC CPA$DASVC	R0 = 0 if automated SVCs were not already disabled	R0 = 1 if automated SVCs were already disabled

3.19. CPA$DPPQL—Delete PP Queue List

This service deletes the specified queue list. The use counts for all dynamic queues on the specified queue list are decremented.

Call	Entry	Exit
SVC CPA$DPPQL	R1 = port number	R0 = 8xxx error R0 = 0 successful

'.3.20. CPA$DQUE—Delete Dynamic Queue

This service deletes the dynamic queue if its use count is zero. Also, unless the queue is a literal queue, there must be no forward or back items on the queue.

Call	Entry	Exit
SVC CPA$DQUE	R1 = QN	R0 = 8xxx error R0 = 0 successful

7.3.21. CPA$DSEG—Delete Dynamic System Segment

This service deletes a dynamically created system segment. The segment must be dynamic and the use count must be zero. The space associated with the segment is returned to the buffer pool.

Call	Entry	Exit
SVC CPA$DSEG	R1 = SSN	R0 = 8xxx error R0 = 0 successful

7.3.22. CPA$ESSN—Extend Dynamic System Segment

This service adds zeroed buffer space to the indicated system segment. The specified address is the current user virtual address within the segment where space is added. The address must be on a 128-byte boundary. The specified length must be a multiple of 128 bytes and must be totally within the same segment descriptor register (SDR). The segment must be a dynamic segment and must be subsegmented.

Call	Entry	Exit
SVC CPA$ESSN	R1 = byte length R2,R3 = Im byte address	R0 = 8xxx error R0 = 0 successful

7.3.23. CPA$FFGPL—Find Free GPL Entry

This service searches the caller's GPL sequentially from the specified entry, looking for a zero entry. The GPLx of the first free entry is returned in R0.

Call	Entry	Exit
SVC CPA$FFGPL	R1 = GPLx to start search	R0 = 8xxx error R0 = GPLx successful

.24. CPA$FFGPLR—Find Free GPL Entry within Range

This service searches the caller's GPL sequentially from the specified entry, looking for a zero entry. The GPLx of the first free entry within the given range is returned in R0.

Call	Entry	Exit
SVC CPA$FFGPLR	R1 = GPLx to start search R2 = number of entries to search	R0 = 8xxx error R0 = GPLx successful

.25. CPA$FFLA—Find Free LA Entry

This service searches the caller's link area (LA) sequentially from the specified entry, looking for a zero entry. The LAx of the first free entry is returned in R0.

Call	Entry	Exit
SVC CPA$FFLA	R1 = LAx to start search	R0 = 8xxx error R0 = LAx successful

.26. CPA$FFLAR—Find Free LA Entry within Range

This service searches the caller's LA sequentially from the specified entry, looking for a zero entry. The LAx of the first free entry within the given range is returned in R0.

Call	Entry	Exit
SVC CPA$FFLAR	R1 = LAx to start search R2 = number of entries to search	R0 = 8xxx error R0 = LAx successful

7.3.27. CPA$FFPST—Find Free PST Entry

This service searches the caller's PST sequentially from the specified entry, looking for a zero entry. The PSTx of the first free entry is returned in R0.

Call	Entry	Exit
SVC CPA$FFPST	R1 = PSTx to start search	R0 = 8xxx error R0 = PSTx successful

7.3.28. CPA$FFPSTR—Find Free PST Entry within Range

This service searches the caller's PST sequentially from the specified entry, looking for a zero entry. The PSTx of the first free entry within the given range is returned in R0.

Call	Entry	Exit
SVC CPA$FFPSTR	R1 = PSTx to start search R2 = number of entries to search	R0 = 8xxx error R0 = PSTx successful

7.3.29. CPA$FFQL—Find Free QL Entry

This service searches the caller's QL sequentially from the specified entry, looking for a zero entry. The QLx of the first free entry is returned in R0.

Call	Entry	Exit
SVC CPA$FFQL	R1 = QLx to start search	R0 = 8xxx error R0 = QLx successful

3.30. CPA$FFQLR—Find Free QL Entry within Range

This service searches the caller's QL sequentially from the specified entry, looking for a zero entry. The QLx of the first free entry within the given range is returned in R0.

Call	Entry	Exit
SVC CPA$FFQLR	R1 = QLx to start search R2 = number of entries to search	R0 = 8xxx error R0 = QLx successful

3.31. CPA$FREES—Free Segment in Memory

This service frees the transient segment held in memory (which is caused by the service CPA$HOLDS). When the use count goes to zero, the segment could be overlaid or rolled out.

Call	Entry	Exit
SVC CPA$FREES	R1 = virtual address of segment	R0 = 8xxx error R0 = 0 successful

3.32. CPA$GETSD—Get SDR Contents

This service returns the 32-bit SDR pertaining to the specified virtual address.

Call	Entry	Exit
SVC CPA$GETSD	R1 = virtual address	R2 = SD upper R3 = SD lower

Note: The length field is always 0IF for subsegmented space (see the DCP Series Implementation Reference Manual, Volume 1, Volume 2 Rev. 1, and Volume 3, (UP-12728). Use CPA$CKSD to obtain a bit map if the space is segmented.

7.3.33. CPA$GPN—Get PN from GPL Entry

This service returns the PN of the procedure in the specified GPL entry.

Call	Entry	Exit
SVC CPA$GPN	R1 = GPLx	R0 = 8xxx error R0 = 0 the specified GPL entry is unused or out of range R0 = PN successful

7.3.34. CPA$GQN—Get QN from QL Entry

This service returns the queue number (QN) of the queue in the specified QL entry. If the specified queue list entry is empty, the returned QN is zero.

Call	Entry	Exit
SVC CPA$GQN	R1 = QLx	R0 = 8xxx error R0 = 0 the specified QL entry is unused or out of range R0 = QN successful

.35. CPA$GSSN—Get SSN from PST Entry

This service returns the SSN of the segment in the specified PST entry. If the specified PST entry is empty, the returned SSN is zero.

Call	Entry	Exit
SVC CPA$GSSN	R1 = PSTx	R0 = 8xxx error R0 = 0 the specified SST entry is unused or out of range R0 = SSN successful

.36. CPA$GSVC—Get SVC

This service returns information on the indicated service. If the specified service code is zero, the service name is used for the search. If the service name is also zero, information is returned on the default service, service code 0.

Call	Entry	Exit
SVC CPA$GSVC	R1 = (15-8) Service code (upper 8 bits) or zero (7-0) Ignored R2-R3 = Service name or zero	R0 = 8xxx if error R0 = 0 if successful R1 = Service code R2-R3 = Service name R4 = Run number of service program R5 = PN of service procedure R6 = Use count of service R7 = Flags

7.3.37. CPA$HOLDS—Hold Segment in Memory

This service increases the specified segment's use count by one. This prevents a use count of zero, which would allow a transient segment to be overlaid or rolled out and updated last.

Call	Entry	Exit
SVC CPA$HOLDS	R1 = virtual address of segment	R0 = 8xxx error R0 = 0 successful

7.3.38. CPA$LPST—Make PST a Loadable Segment

This service creates a system segment containing the PST. Therefore, the caller can enter SSNs directly in the PST without invoking the CPA$APST service.

This service enters the SSN of the new segment containing the PST into the PST at the PSTx specified in R2.

Call	Entry	Exit
SVC CPA$LPST	R1 = PN owning PST (0 = caller) R2 = PSTx	R0 = 8xxx error Note: R0 = EC$CPA+ER$LPST if PST is already a loadable segment R0 = 0 successful

3.39. CPA$PINFO—Get PN Information

This service returns the lengths of the related lists for the specified procedure. These are QLn, LAn, PSTn, GPLn. If the specified procedure is zero, then the caller PN is assumed.

Call	Entry	Exit
SVC CPA$PINFO	R1 = PN (0 = caller)	R0 = 8xxx error R0 = 0 successful R0 = 1 successful but procedure blocked R2 = QLn R3 = LAn R4 = PSTn R5 = GPLn

3.40. CPA$QL1X—Get System Queue List 1 Index

This service returns the queue's index (QX) location of the specified queue, SQL1 (system queue list 1), in the system. The caller's queue list index (QLx) specifies which queue.

Call	Entry	Exit
SVC CPA$QLX1	R1 = QLx	R0 = 8xxx error R0 = QX successful

3.41. CPA$QMDE—Modify Queue Mode

This service sets the mode of the specified queue to zero so back items can be recovered.

Call	Entry	Exit
SVC CPA$QMDE	R1 = QLx	R0 = 8xxx error R0 = 0 successful

7.3.42. CPA$QMOD—Modify Queue

This service modifies portions of the specified queue. The caller must have arm access to the queue.

If the QX is not in the range 0 to 3, it must be in the user range of cascade queues (in SQL1). This allows the caller to specify a full 32-bit subarchitectural interface item when using cascaded queues from CPA SQL1. Otherwise the user must specify only 8 bits from SQL1 queues.

If the QX is in the range 0 to 3, the PN must be in the user PT range.

The R8and R9 parameters are used in the reverse order of that in the CPA$QSTAT service. (See 7.3.4.3.

Call	Entry	Exit
SVC CPA$QMOD	R1 = QLx R6 = (15-8) Qx; (7-0) not used R7 = device address R8 = SAI lower (parameter) R9 = SAI upper (PN)	R0 = 8xxx error R0 = 0 successful

7.3.42a CPA$QSMD – Set Queue Mode

Set the mode field for the specified queue to the indicated value.

Call	Entry	Exit
SVC CPA$QSMD	R1 = QLx R2 = Mode value	R0 = 0 if successful = error code if not successful

3.43. CPA$QSTAT—Get Queue Status

This service returns the current header fields from the specified queue.

Call	Entry	Exit
SVC CPA$QSTAT	R1 = QLx	See the following.

Exit conditions for MCT, literal, or space queue:

```
R0 = 8xxx     error
R0 = 0        error, no queue
R0 = QN       of queue
R2 = (15-8)   reserved
   = (7-0)    back items
R3 = (15-8)   forward
   = (7-0)    current
R4 = (15-8) queue type
   = (7-0)  queue mode
R5 = (15-8) queue threshold
   = (7-0)  queue size
R6 = (15-8) Qx
   = (7-0) N(r)
R7 = queue device address
R8 = queue upper SAI (PN to schedule)
R9 = queue lower SAI (R1 parameter)
```

Exit conditions for link-list queue:

```
R0 = QN of queue
R2 = 0 (unprocessed back items)
R3 = forward
R4 = (15-8) queue type
   = (7-0) QX
R5 = 0 queue size
R6 = reserved
R7 = queue device address
R8 = queue upper SAI (PN to schedule)
R9 = queue lower SAI (R1 parameter)
R10 = reserved
R11 = threshold
```

7.3.44. CPA$RAQL—Remove Queue from Alternate Queue List (PP)

This service removes a queue from the queue list of a PP alternate environment. The QN of the removed queue is returned in R0.

Call	Entry	Exit
SVC CPA$RAQL	R1 = port number R2 = AEL index R3 = QLx	R0 = 8xxx error R0 = QN successful

7.3.45. CPA$RGPL—Remove GPL Entry

This service clears the specified GPL entry.

Call	Entry	Exit
SVC CPA$RGPL	R1 = GPLx	R0 = 8xxx error R0 = 0 successful

7.3.46. CPA$RPPQL—Remove Queue from Queue List (PP)

This service deletes the queue list entry of the specified PP. The use count for a dynamic queue entry is decremented. The removed QN is returned in R0 if successful. If there was no entry in the queue list, R0 is set to zero.

Call	Entry	Exit
SVC CPA$RPPQL	R1 = port number R2 = QLx	R0 = 8xxx error R0 = QN successful

7.3.47. CPA$RPST—Remove SSN from PST Entry

This service clears the specified PST entry.

Call	Entry	Exit
SVC CPA$RPST	R1 = PSTx	R0 = 8xxx error R0 = SSN of entry successful

48. CPA$RQL—Remove Queue from QL (PN)

This service clears the specified queue list entry. The use count for a dynamic queue entry in the specified queue list is decremented. The removed QN is returned in R0 if successful. If there was no entry in the queue list, R0 is set to zero.

Call	Entry	Exit
SVC CPA$RQL	R1 = QLx	R0 = 8xxx error R0 = QN successful

49. CPA$RSPT—Remove Segment from PT Entry

This service removes a segment from the specified procedure table entry. This effectively removes this segment from one of the first four SDs of the procedure. The SSN provided in R1 must match the current entry in the PT.

Call	Entry	Exit
SVC CPA$RSPT	R1 = SSN R2 = PN (0 = caller) R3 = SD number (0 to 3)	R0 = 8xxx error R0 = 0 successful

Note: *If the specified PN is the caller, the removed segment will not be visible upon return from the service.*

50. CPA$RSSN—Reserve Consecutive SSN Entries

This service reserves the requested number of system segment table (SST) entries. These entries are only reserved. No segments are created. These entries may be used by specifying a reserved SSN in a subsequent service call to create a segment (CPA$CSEG).

Call	Entry	Exit
SVC CPA$RSSN	R1 = number of consecutive SSNs	R0 = 8xxx error R0 = 0 successful R1 = number of SSNs R2 = first SSN reserved

7.3.51. CPA$RSVC—Remove Extended SVC Procedure

This service allows the program that registered a service procedure to revoke its registration. If other programs on the DCP are currently using this service, they can either be allowed to complete or be aborted, depending on the option invoked. A blocked service is one where current users are allowed to complete but no new calls are allowed. A blocked service may be aborted by reinvoking this service with the abort bit set. A different service using the same service code may not be registered unless the abort option was used to revoke the registration of the previous service, though a blocked service may be unblocked through the CPA$ASVC service as long as the calling program, requested procedure and service name are not changed.

Note that the use count of an aborted service is cleared, but the use count of a blocked service is not affected.

Call	Entry	Exit
SVC CPA$RSVC	R1 = (15-8) Service code (upper 8 bits) (7-0) Ignored R2 = 0 if current users of the service are allowed to finish but all other users are blocked R2 = 8xxx if current users are forced to abort	R0 = 8xxx if error R0 = 0 if successful

52. CPA$USSN—Unreserve Consecutive SSN Entries

This service unreserves the requested number of system segment table (SST) entries. These entries are only unreserved, no segments are deleted.

Call	Entry	Exit
SVC CPA$USSN	R1 = number of consecutive SSNs (1 to 15) R2 = first reserved SSN	R0 = 8xxx error R0 = 0 successful

53. CPA$XLA—Extend Link Area

This service expands the link area of the specified procedure. If the PN is specified as zero, then the calling procedure is assumed. All other procedures currently sharing the same link area are updated to share the new, expanded link area.

Call	Entry	Exit
SVC CPA$XLA	R1 = PN (0 = caller) R2 = Extra link area entries required	R0 = 8xxx error R0 = 0 successful R1 = PN

54. CPA$XPST—Extend PST

This service expands the PST of the specified procedure. If the PN is specified as zero, the calling procedure is assumed. All other procedures currently sharing the same PST are updated to share the new, expanded PST.

Call	Entry	Exit
SVC CPA$XPST	R1 = PN (0 = caller) R2 = extra PST entries required	R0 = 8xxx error R0 = 0 successful R1 = PN

7.3.55. CPA$XQL—Extend Queue List

This service expands the queue list of the specified procedure. If the PN is specified as zero, the calling procedure is assumed. All the procedures currently sharing the same queue list are updated to share the new, expanded queue list.

Call	Entry	Exit
SVC CPA$XQL	R1 = PN (0 = caller) R2 = extra queue list entries required	R0 = 8xxx error R0 = 0 successful R1 = PN

.4. Dictionary Services

Dictionary Services are called to gain access to the dictionary of a program (either the caller's dictionary or the dictionary for a specified runid) and to return specific information from this dictionary to the caller. Every active program on the DCP has a dictionary to keep track of named entities such as segments (defined via SSTDEF), procedures (INITDEF/MARKDEF/PRIVDEF/PROCDEF), or queues (QUEUE).

Where Dictionaries Come From

A program dictionary is constructed as the absolute (ABS) element and built using the program builder (BUILD or DCPBUILD). The builder appends the dictionary to the end of the ABS element. When the program is loaded by DCP/OS (or when DCP/OS is initialized), the dictionary is put into loadable segments that are referenced by the procedure segment table (PST) of the two dictionary service procedures. The dictionary packet is described in Appendix A.

Accessing a Program Dictionary

When a program is executed, the loader in DCP/OS automatically builds a segment containing the dictionary for this program. This segment is accessed using the dictionary services (DIC$).

4.1. DIC$FENT – Find a Dictionary Entry by Type/Flags

This service finds the first dictionary entry, starting at a specified dictionary entry number, of the specified type with the indicated flag set. Return the entry number if the item is found, and the item number according to the item type (for example, the PN for a procedure or the SSN for a segment).

Call	Entry	Exit
SVC DIC$FENT	R1 = Starting entry number for search R2 = (15-8) Flags that must be set (0 if none) R2 = (7-0) Type of entry (PN, SSN, etc.) R3 = Runid (See appendix A or DIC$FINM for defined types.)	R0 = 0 if item was found R0 = error code otherwise R2 = Item number (second word of DC$ADRS field) R4 = Entry number of the found item

7.4.2. DIC$FIAD—Search Dictionary by Address

This service searches the program dictionary for a match with the entity type and address defined in the request packet. If found, the whole dictionary entry is copied into the caller's packet. A type of zero can be specified to indicate that only a match on the address is required.

Note: The interface packet must not be in the first four SDRs of the caller's visibility, because these SDRs are architecturally cleared on invoking the service.

Call	Entry	Exit
SVC DIC$FIAD	R10 = virtual address of packet DC$ADRS = required address (2 words) DC$TYPE = required type (see DIC$FINM for types)	R0 = 8xxx error R0 = 0 successful R1 = dictionary index number

4.3. DIC$FINM—Search Dictionary by Name

This service searches the program dictionary for a match with the entity type and name defined in the request packet. If found, the whole dictionary entry is copied into the caller's packet. A type of zero can be specified to indicate that only a match of name is required.

Note: The interface packet must not be in the first four SDRs of the caller's visibility, because these SDRs are architecturally cleared on invoking the service.

Call	Entry	Exit
SVC DIC$FINM	R10 = virtual address of packet DC$NAME = required name (8 bytes) DC$TYPE = required type Types: DC$TPSEG = segment DC$TPPN = procedure DC$TPPPP = PP program DC$TPQ = queue 0 = match any type	R0 = 8xxx error R0 = 0 successful R1 = dictionary index number

7.4.4. DIC$GENT – Read a Dictionary Entry

This service, when given the entry number of a dictionary entry, copies the dictionary information into the packet supplied by the caller. The supplied packet should contain a copy of the dictionary entry.

Call	Entry	Exit
SVC DIC$GENT	R1 = Entry number R10 = Virtual address of packet	R0 = 0 if item was copied successfully R0 = Error code otherwise R1 = Unchanged

7.4.4a DIC$GENTC – Read a Dictionary Entry for Runid

The service, when given the entry number of a dictionary entry for the specified runid, copies the dictionary information into the packet supplied by the caller.

Call	Entry	Exit
SVC DIC$GENTC	R1 = entry no. R2 = Runid R10 = Virtual address of packet	R0 = 0 if successful = error code if not successful

.5. SDF Record Services

The DCP/OS provides services that allow the user program to read from and write to record structured files.

The SVC mechanism can invoke every SDF service. The function code definitions are found in AAFILDEF in Appendix C.

The user parameter FR$USER is unchanged over every call.

Example

```
LOADC     R1,AV$MCT          . Point to message MCT
SVC       E$SDFII            . Go initialize input
```

5.1. E$GET—Get Record

This service reads a variable length record from an opened file. All parameters are passed in registers. This allows the user program to issue file read requests without having to specify a file request packet (FRP).

The data must be in symbolic element type format. Each variable length record is preceded by a 16-bit byte count.

Call	Entry	Exit
SVC E$GET	R1,R2 = FCB-ID R3 = block number R4 = word offset in block R5,R6 = virtual address of user buffer (Im - must be word boundary.)	R0 = number bytes read (X'FFFF' = EOF) R1 = virtual address of user buffer (word) R3 = next block number R4 = word offset in next block Destroys R2,R5-R9

.5.2. E$READ—Read Record

This service reads a record from an opened file. The call specifies the record length and no virtual space is required. This allows the user program to issue file read requests without having to specify an FRP.

Call	Entry	Exit
SVC E$READ	R1,R2 = FCB-ID R3 = block number R4 = word offset in block R5,R6 = virtual address of user buffer (Im - must be word boundary.) R7 = record length (words)	R0 = 0 successful R1 = virtual address of user buffer (word) R3 = next block number R4 = word offset in next block Destroys R2,R5-R9

7.5.3. E$SDFI—Read SDF Record

This service reads a variable length record into user space at the specified virtual address. The caller must not modify the FRP and FRP buffer.

Call	Entry	Exit
SVC E$SDFI	R1 = virtual address of MCT of FRP packet R2,R3 = virtual address of record buffer (word/byte) FR$FCB = (used by SDFI) FR$BN = (used by SDFI) FR$BW = (used by SDFI)	Destroys R0-R5 R0 = (FR$CC) file manager code R4 = (FR$WC) byte length of record (NB can be zero) set to zero if R0 is nonzero

7.5.4. E$SDFIC—Close SDF Input

This service deallocates any data under SD1 of the FRP.

Call	Entry	Exit
SVC E$SDFIC	R1 = virtual address of MCT of FRP packet	R0 = (FR$CC) file manager completion code Destroys R1 - R5

5.5. E$SDFII—Initialize SDF Input

This service initializes the FRP of an opened file for sequential record input. The file must have been opened with a suspension type interface, not queued. The SDF record input services apply to a file or element. If element input is required, it is the caller's responsibility to position the FR$BN field to the beginning of the element by calling FR$ESRCH (7.6.12).

Call	Entry	Exit
SVC ERSDFII	R1 = virtual address of MCT of FRP packet FR$FCB = file manager identifier FR$BN = initial input block number	R0 = (FR$CC) completion code FR$BUF = initialized for E$SDFI use FR$BW = initialized for E$SDFI use FR$WC = initialized for E$SDFI use Destroys R1 - R5

5.6. E$SDFO—Write SDF Record

This service packs a variable length record into the output buffer at location SD1 of the FRP and writes out full buffers to disk, as necessary. The caller must not modify the FRP and the buffer.

Call	Entry	Exit
SVC E$SDFO	R1 = virtual address of MCT of FRP packet R2,R3 = virtual address of data record (word/byte) R4 = data length in bytes (maximum 254) FR$FCB = (used by SDFO) FR$BN = (used by SDFO) FR$BUF = (used by SDFO)	Destroys R0-R5 R0 = (FR$CC) file manager completion code FR$BN = corrupted FR$WC = corrupted

'.5.7. E$SDFOC—Close SDF Output

This service stores an end-of-file sentinel (X'FFFF') and writes out the remaining data buffer to disk.

Call	Entry	Exit
SVC E$SDFOC	R1 = virtual address of MCT of FRP packet FR$FCB = file manager identifier	Destroys R0-R5 R0 = (FR$CC) file manager completion code FR$WC = Corrupted FR$BN = next block number after end of element FR$ELEN = total number of blocks in element

'.5.8. E$SDFOI—Initialize SDF Output

This service initializes the FRP of an opened file for sequential record output. The file must have been opened with a suspend interface, not queued. The SDF record output services apply to a file or element. If element output is required, it is the caller's responsibility to position the FR$BN field to the beginning of the element.

Call	Entry	Exit
SVC E$SDFOI	R1 = virtual address of MCT of FRP packet FR$FCB = file manager identifier FR$BN = initial output block number	R0 = (FR$CC) file manager completion code FR$BUF = corrupted FR$ELBN = set to FR$BN for later element insertion Destroys R1 - R5

.6. File Manager Services

The file manager provides a number of services to allow the user to create and access files on mass storage. For example, a user program might use file manager services for the following sequence of operations:

- Catalog a file.
- Open the file.
- Write to the file.
- Close the file.

The file manager can be invoked with the SVC instruction. The user can interact with the file manager in the following two ways:

- The user's task is suspended until the file manager has completed the function (suspension interface). This is the usual method when the user is not concerned about the program being suspended for several milliseconds.
- The user gains control immediately after passing the request to the file manager. The requests are returned to the user through a user-specified queue when the operation has been completed.

The file manager receives instructions from the user via the file request packet (FRP). The FRP is a contiguous data packet that contains the parameters for the requested function. Details on the various FRP fields are given in the following subsections and in Appendix A. Of universal interest, though, are the following FRP fields:

FR$FNC	Contains the function code (see Appendix A)
FR$CC	Contains the completion code (see Appendix A)
FR$QN	Is used to distinguish between a suspension interface (FR$QN = 0) and a queued interface (FR$QN = queue number)
FR$USER	Contains user-defined data and is maintained by the file manager

In order to invoke the file manager, the user must first decide whether to use the suspension interface or a queued interface.

Note: Choosing a suspension interface or a queued interface for an OPEN FILE function also affects all operations on the opened file, such as READ, WRITE, or CLOSE. Changing FR$QN in the FRP for these subsequent operations will have no effect.

Suspension Interface

There are five contiguous segment descriptor registers (SDRs) that must be set aside for file manager usage:

SDR n	FRP MCT or not present. This SDR will be deallocated and a new MCT will be allocated by the file manager. It must not contain critical data.
SDR n+1	Input/output data area (read/write functions only) or not present. The file manager uses this virtual space as work space. It corrupts any user data present, except in the case of the read/write functions.
SDR n+2	Input/output data area (read/write functions only) or not present. The file manager uses this virtual space as work space. It corrupts any user data present, except in the case of the read/write functions.
SDR n+3	FRP (must be on SDR boundary)
SDR n+4	not present. This virtual space is used by the file manager.

Once the FRP has been initialized with FR$QN = 0 and the SDRs have been set up, the file manager can be invoked as follows:

```
LOADC   R1,FRPMCT       . Virtual address of SDR n

SVC     FILE$           . Call the file manager
```

The user task is suspended until the file manager completes the validation and execution of the function. The results of the function are as follows:

R0	Contains the results of the function validation (R0 = 0 if the function was valid)
FR$CC	Contains the results of the function itself (FR$CC = 0 if the function was successful)

Queued Interface

In order to request a queued response from the file manager, the user must pass the file manager a nonzero queue number in the FR$QN field of the FRP. The file manager validates the request and immediately returns control to the user. When the function has completed, the file manager queues the results back to the user on the specified queue.

Just as in the case of the suspension interface, five contiguous segment descriptor registers (SDRs) must be set aside for file manager usage. These SDRs are as follows:

SDR n	FRP MCT. This MCT will be queued to the specified queue when the function has completed.
SDR n+1	Input/output data area (read/write functions only) or not present. The file manager uses this virtual space as workspace. It corrupts any user data present, except in the case of the read/write functions.
SDR n+2	Input/output data area (read/write functions only) or not present. The file manager uses this virtual space as workspace. It corrupts any user data present, except in the case of the read/write functions.
SDR n+3	FRP. The FRP must lie on the SDR boundary.
SDR n+4	not present. This virtual space is used by the file manager.

Before the file manager returns control to the user, it validates the request and removes the five SDRs from the user's visibility. The results of the validation are returned in R0 which contains 0 if the function is valid.

It is the user's responsibility to process the specified return queue to get the results of the file manager function. The completion code is contained in the FRP in the FR$CC field. Other fields in the FRP or data in the Read/Write SDRs (n+1 and n+2) contain returned information pertaining to the function performed.

7.6.1. FR$ASG—Assign a Device

This service assigns the specified mass storage device exclusively to the calling program, if it is available.

If successful, the FR$FCB field returns an identifier for subsequently freeing the device.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$ASG (function code = assign) FR$USER = user-defined field FR$VID = device-id	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.2. FR$CAT—Catalog a File

This service creates a file on the specified disk/diskette device. The volume may be specified by name or generic type and defaults to any volume (except IFDC) on which space is available. The order of priority follows:

1. 8441 Winchester
2. 8409 Winchester
3. 8408 fixed cartridge
4. 8408 removable cartridge
5. 8441 diskette
6. 8406 FDDS
7. 8406 FDS

The file descriptor information is entered in an unused data set label (DSL) on the target volume.

The file control block (FCB) field is defined as the same offset as the file name; therefore, the file name is corrupted. If the FR$QUAL field is empty or invalid, the current project-ID will be substituted by the file manager.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$CAT (function code = catalog) FR$USER = user-defined field FR$QN = QN of user return queue (or 0) FR$VID = volume-id (or FR$DT = device type) PD$TYPIF = IFDC PD$TYPDR = cartridge, removable PD$TYPDF = cartridge, fixed PD$TYPDT = diskette PD$TYPWD = 8409 Winchester PD$TYPDD = double-sided diskette PD$TYPSD = 8441 diskette PD$TYPSW = 8441 Winchester FR$NAM = file name, 8 characters left-justified FR$QUAL = qualifier, 6 characters left-justified FR$NUV = file size in blocks (block = 256 bytes)	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

3. FR$CFRO—Clear File's Read-Only Flag

This service clears the file's read-only flag.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$CFRO (function code = clear file's read-only flag) FR$USER = user-defined field FR$FCB = file manager identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.4. FR$CLS—Close a File

This service closes the file previously opened on the specified file identifier (FR$FCB). The associated FCB entry is freed and the user return queue, if applicable, is removed from the file manager's queue list. The associated DSL is updated if the highest block written changed during the assignment.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$CLS (function code = close) FR$USER = user-defined field FR$FCB = file manager identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.5. FR$CLSI—Close Immediate

This service closes the specified file and forces a suspend-type interface, whether or not the file was opened with queued I/O.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$CLSI (function code = close imm) FR$USER = user-defined field FR$FCB = file manager identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.6. FR$DEL—Delete a File

This service deletes the requested file if the file is not currently in use. The DSL is marked available.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$DEL (function code = delete) FR$USER = user-defined field FR$QN = QN of user return queue (or 0) FR$NAM = file name, 8 char left-justified FR$QUAL = qualifier, 6 char left-justified	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7. FR$DOWN—Down a Device

This service downs a mass storage device if it is not in use. This service deletes all files cataloged on the disk volume from the system catalog.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$DOWN (function code = down) FR$USER = user-defined field FR$QN = QN of user return queue (or 0) FR$DT = device identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

8. FR$EDEL—Delete an Element from the TOC

This service marks an element as deleted in the table of contents (TOC) of an opened program file. The fields FR$ENAME and FR$ETYPE must be supplied in the FRP.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$EDEL (function code = delete element) FR$USER = user-defined field FR$FCB = file manager identifier FR$ENAME = element name FR$ETYPE = element type	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.9. FR$EINS—Insert an Element into the TOC

This service inserts an element into the TOC of an opened program file. The fields FR$ENAME, FR$ETYPE, FR$ESUBT, FR$ELBN, FR$ELEN, FR$EYEAR, FR$EMON, FR$EDAY, FR$EHOUR, FR$EMIN, and FR$ESEC must be supplied in the FRP. Any previous version of the same element name and type is automatically deleted.

This service should be called only after the new element is inserted in the file. In that way, if the calling program terminates before managing to call this service, the file is not corrupted. The correct sequence of service calls to insert an element is as follows:

FR$OPN	- Open the file (exclusively)
FR$ENXWL	- Get the next write location
FR$WRT	- Write out the new element
FR$EINS	- Insert new element information into TOC

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$EINS (function code = insert element) FR$USER = user-defined field FR$FCB = file manager identifier FR$ENAME = element name FR$ETYPE = element type FR$ESUBT = element subtype FR$ELBN = element start block number FR$ELEN = element size in blocks FR$EYEAR = creation year FR$EMON = creation month FR$EDAY = creation day FR$EHOUR = creation hour FR$EMIN = creation minute FR$ESEC = creation second	R0 = 0 request accepted R0 = 8xxx request not accepted

10. FR$ENXWL—Get Element Next Write Location

This service retrieves the next write location from the root TOC block of an opened program file. The next write location is returned in the FR$BN field of the FRP.

If the file was opened for output (FR$OPOUT), then the block number maintained by the system in the FCB is also set to 'NXWL'.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$ENXWL (function code = get write location) FR$USER = user-defined field FR$FCB = file manager identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.11. FR$ERS—Erase a File

This service effectively deletes all elements from a program file by writing out an empty TOC to block zero of the (opened) file.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$ERS (function code = erase file) FR$USER = user-defined field FR$FCB = file manager identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.12. FR$ESRCH—Search the TOC

This service searches the TOC of an opened program file for the specified element. The fields FR$BN, FR$ELBN, FR$ESUBT, FR$ELEN, FR$EYEAR, FR$EMON, FR$EDAY, FR$EHOUR, FR$EMIN, and FR$ESEC are returned if the search was successful. Otherwise, these fields are untouched.

If the file was opened for either input or output (FR$OPIN, FR$OPOUT), the current block number maintained by the system is also set to the beginning of the element.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$ESRCH (function code = search TOC) FR$USER = user-defined field FR$FCB = file manager identifier FR$ENAME = element name FR$ESUBT = element subtype (ignored if 0)	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC) FR$ETYPE = element type

13. FR$EUPWL—Update Next Write Location

This service updates the next write location in the root TOC block of an opened program file. This service is implicitly invoked by issuing an insert element (FR$EINS) request.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$EUPWL (function code = update write location) FR$USER = user-defined field FR$FCB = file manager identifier FR$BN = block number of next write location	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

14. FR$FDN—Down a File

his service marks the specified file as down, provided that it is not currently assigned to any application.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$FDN (function code = file down) FR$USER = user-defined field FR$NAM = file name to down	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

15. FR$FREE—Free a Device

This service frees the specified device from exclusive assignment.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$FREE (function code = free) FR$USER = user-defined field FR$FCB = file manager identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.16. FR$FUP—Up a File

his service marks the specified file on the specified volume as up, provided that no file by the same name is already up.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$FUP (function code = up file) FR$USER = user-defined field FR$VID = volume name (6 characters, left-justified and blank filled) FR$NAM = file name (8 characters, left-justified and blank filled) FR$QUAL = qualifier (6 characters, left-justified and blank filled)	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.17. FR$OPN—Open a File

This service opens the requested file, if found in the system catalog. The field FR$LOC specifies the location of the required file.

If FR$LOC is zero, a local file is assumed; otherwise, FR$LOC should contain the name of a configured host channel.

The open type field (FR$OT) can specify whether read-only, write-only, or read and write access is required. This field can also specify whether or not exclusive use is required. The open type (FR$OT) applies to local files only.

If the open type is specified as read (FR$OPIN) or write (FR$OPOUT), the system maintains the current block number and any value placed by the user program in the block number field (FR$BN) is ignored on read (FR$RD) and write (FR$WRT) functions.

If the open type is random (FR$OPRND), maintenance of the block number field (FR$BN) is the responsibility of the user program.

The field FR$QN specifies a user return queue or, if zero, indicates that suspended I/O is required. The calling process is suspended until the requested function is complete. The chosen interface is user selectable and all file manager function calls are affected on the particular file between the open and a subsequent close.

FR$FCB is a two-word field that uniquely identifies each open file.

The FCB field is defined as the same offset as the file name; therefore, the file name is corrupted.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$OPN (function code = open) FR$USER = user-defined field FR$QN = QN of user return queue (or 0) FR$QUAL = qualifier (6 characters for DCP files, 12 characters for host files left-justified and blank filled) FR$NAM = file name, 8 chars left-justified FR$OT = open type (FR$OPIN - input) (FR$OPOUT - output) (FR$OPEXT - extend) (FR$OPRND - random)	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC) If successful FR$FCB = file-ID FR$VID = volume-ID

The preceding open types may be combined with FR$OPX to request exclusive use of the file. The open type FR$OPUPD (update) is a convenient combination of FR$OPRND and FR$OPX, giving exclusive use for read and write operations with random access.

```
(FR$OPX - exclusive)
(FR$OPUPD - update)
```

The open type values must be stored in the open type field, as follows:

```
LOADC     R0,FR$OPRND*/12    . Generate open type
STORE     R0,FR$OT,R10       . Store in FRP
```

7.6.18. FR$OPNI—Open a File Immediate

This service opens a file and suspends the caller until the open function is complete. Please refer to FR$OPN (7.6.17) for the specification of entry and exit parameters.

7.6.19. FR$POS—Position a File

This service positions a file to the specified block number. The file must have been opened and is specified by file manager identifier (FR$FCB).

This service is not pertinent to files opened for random access since the calling program maintains the current block number.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$POS (function code = position) FR$USER = user-defined field FR$FCB = file manager identifier FR$BN = block number	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

20. FR$PREP—Format a Disk

This service issues a format request on behalf of the calling program. The specified device must be exclusively assigned to the caller.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$PREP (function code = format) FR$USER = user-defined field FR$FCB = file manager identifier FR$DEN = density (sector size) FR$IFD = flag for IFDC interface FR$MSP = zero (determine format density) = non-zero (device characteristics table index)	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

21. FR$QFE—Move Filename into Appropriate FRP Field

This service parses a single string containing the *qualifier*filename.element* and places them into the appropriate fields in the file request packet (FRP). The string is passed to the service in the FRP starting in the FR$QUAL field. The parsing is terminated by the first space found.

Call	Entry	Exit
SVC FR$QFE	R1 = virtual address of FRP MCT FR$QUAL = Q*F.ELT string	R0 = 0 request accepted R0 = 8xxx request not accepted FR$QUAL, FR$NAM, and FR$ENAME filled

.6.22. FR$RD—Read

This service reads the requested number of words, starting at the beginning of the specified logical block in the opened file. The block number is relative to the beginning of the file (block 0) and each block is 256 bytes long. If the file was opened for input (FR$OPIN), the block number is taken from the "current block number" field maintained in the FCB.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$RD (function code = read) FR$USER = user-defined field FR$FCB = file manager identifier FR$BN = block number (if open type random) FR$WC = word count (maximum 4k) FR$BUF = data offset	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

.6.23. FR$RENF—Rename a File

This service renames a currently opened file. The name is not changed on mass storage until the file is closed. The user must have exclusive access to the file and the file must not be flagged as read-only.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$RENF (function code = rename) FR$USER = user-defined field FR$FCB = file manager identifier FR$VID = file name, 8 chars left-justified FR$QUAL = new qualifier, 6 characters left-justified	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

6.24. FR$RESET—Reset HBW

This service resets the highest block written (HBW) of a currently open file. This service is only pertinent to data files (not program files). The HBW is not updated in the DSL until the file is closed.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$RESET (function code = reset) FR$USER = user-defined field FR$FCB = file manager identifier FR$BN = new HBW	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

.6.25. FR$SFRO—Set File to Read-Only

This service sets the open file to read-only. It takes effect when the file is closed.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$SFRO (function code = set file to read-only) FR$USER = user-defined field FR$FCB = file manager identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

.6.26. FR$STAT—Status of File

This service returns the status of the specified file. The file descriptor information is returned in the FRP in FCB format starting at offset FR$FCBDEF. Refer to Appendix A for more information on the FCB format.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$STAT (function code = status) FR$USER = user-defined field FR$QN = QN of user return queue (or 0) FR$NAM = file name, 8 chars left-justified FR$QUAL = qualifier, 6 chars left-justified	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.27. FR$STATO—Status of Open File

This service returns the status of a currently open file. The file descriptor information is returned in the FRP in FCB format starting at offset FR$FCBDEF. Refer to Appendix A for more information on the FCB format.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$STATO (function code = status) FR$USER = user-defined field FR$FCB = file manager identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.28. FR$UP—Up a Device

This service places a mass storage device in an up state and reads the volume label. It automatically registers each in-use DSL into the system catalog.

Service is reserved for system use only.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$UP (function code = up) FR$USER = user-defined field FR$QN = QN of user return queue (or 0) FR$DT = device identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

.29. FR$WRT—Write

This service writes the requested number of words, starting at the beginning of the specified logical block. The block number is relative to the beginning of the file (block 0) and each block is 256 bytes long.

If the file was opened for output (FR$OPOUT) or extend (FR$OPEXT), the block number is taken from the "current block number" field maintained in the FCB.

The data written must be contiguously addressable (virtual addressing) from FR$BUF for the specified word count. If the word count does not span an integral number of blocks, then the content of the remainder of the last block written is undetermined.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$WRT (function code = write) FR$USER = user-defined field FR$FCB = file manager identifier FR$BN = block number (if open type random) FR$WC = word count (maximum 4k) FR$BUF = data offset	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7.6.30. FR$WCT—Write Catalog

This service writes the catalog entry for a given open file to mass storage without having to close the file. This service is used to protect against lost information if the program fails while maintaining an open file.

Call	Entry	Exit
SVC FILE$	R1 = virtual address of FRP MCT FR$FNC = FR$WCT (function code = write catalog) FR$USER = user-defined field FR$FCB = file manager identifier	R0 = 0 request accepted R0 = 8xxx request not accepted (see FRP completion code in FR$CC)

7. Instrumentation Services

Instrumentation is a microcode-supported feature of CPA that enables the selective recording of software execution and event-related information, such as message tracing or procedure call tracing. The selection of data to gather is controlled by setting instrumentation control words for the CP and for each PP.

Instrumentation data is recorded into a buffer, either in wraparound fashion or sequentially. In wraparound mode, the same buffer is used continuously, so that all data is captured in memory only. In sequential mode, a full instrumentation buffer is queued to the system queue list 5 (SQL5) so that a CP procedure may process the information, typically by logging the buffer to mass storage.

The DCP/OS instrumentation services provide the necessary privileged interface to manipulate the instrumentation control words (ICWs) and to gain access to SQL5.

Instrumentation is a powerful feature, but you should be aware that it is costly (CP overhead). Give serious thought before using it in a production environment.

.1. I$FLUSH—Flush Instrumentation

This service eliminates all instrumentation structures and data for the specified run.

Call	Entry	Exit
SVC I$FLUSH	R1 = run number (0 = caller)	R0 = 8xxx error R0 = 0 successful

.2. I$PN—Set Procedure ICW

This service sets the CP instrumentation control word for a specific procedure.

Call	Entry	Exit
SVC I$PN	R1 = run number (0 = caller) R2 = PN R3 = ICW	R0 = 8xxx error R0 = 0 successful

7.7.3. I$PP—Set Port Processor ICW

This service sets the PP instrumentation control word for a specific PP.

Call	Entry	Exit
SVC I$PP	R1 = run number (0 = caller) R2 = port number R3 = ICW	R0 = 8xxx error R0 = 0 successful

7.7.4. I$PPBUF—Establish PP Instrumentation Buffer

This service establishes an indirect instrumentation buffer for a PP. Instrumentation buffers (IBs) may be shared by making successive service calls using the same IB pointer table index.

Call	Entry	Exit
SVC I$PPBUF	R1 = run number (0 = calling run) R2 = 0 - no wraparound = 1 - wraparound R3 = port number R4 = IB pointer table index	R0 = 8xxx error R0 = 0 successful

7.7.5. I$PPSETUP—Establish PP Instrumentation Only

This service builds all structures for PP instrumentation only (as distinct from I$SETUP (See 7.7.8), which builds CP and PP instrumentation structures).

Call	Entry	Exit
SVC I$PPSETUP	R1 = run number (or 0 if calling run)	R0 = 8xxx error R0 = 0 successful R1 = ER$PPIT (BIT 12) 1 = PP instrumentation already established (this is meaningful only if R0 = 0) R2 = IBp index allocated for a successful return and no status bits

.6. I$RUN—Set Run ICW

This service sets the CP instrumentation control word to the requested value for the specified run.

Call	Entry	Exit
SVC I$RUN	R1 = run number (0 = calling run) R2 = ICW	R0 = 8xxx error R0 = 0 successful

7. I$RUNBUF—Establish Run Instrumentation Buffer

This service establishes an instrumentation buffer (IB) for a run. It assumes I$SETUP is already called, hence an IB table pointer index and buffer are already established. I$RUNBUF is used when either the buffering mode (wrap or nonwrap) or the IB table pointer changes.

Call	Entry	Exit
SVC I$RUNBUFF	R1 = run number (or 0 if the calling run) R2 = 0 if no wraparound 1 if wraparound R3 = IB pointer table index (a value of 0FFFF causes the first free entry found in the IB pointer table index to be used)	R0 = 8xxx error R0 = 0 successful

7.7.8. I$SETUP—Setup Instrumentation

This service builds all structures required for running the instrumentation. All indirect instrumentation buffer pointers (IIBP) are initialized to the first entry in the instrumentation buffer pointer (IBP) file and a 4K wraparound buffer is allocated for that entry. The status is returned in R1.

Call	Entry	Exit
SVC I$SETUP	R1 = run number (0 = calling run)	R0 = 8xxx error R0 = 0 successful R1 = ER$PPIT (bit 14) 1 = Port Processor Instrumentation Table not established ER$PUT (bit 13) 1 = Procedure use table not established ER$IBP (bit 12) 1 = Instrumentation buffer pointer (IBp) not established

7.7.9. I$SQL5G—Get SQL5 Access

This service allows the calling program to get access to queues within system queue list 5 (the instrumentation buffer queues).

Call	Entry	Exit
SVC I$SQL5G	R1 = system QL identifier (0 = SQL5) R2 = (bits 12 - 0) caller's QLx (bit 15) get (bit 14) put (bit 13) arm R3 = SAI PN (procedure number)	R0 = 8xxx error R0 = 0 successful R1 = number of queues in SQL5

.10. I$SQL5R—Remove SQL5 Access

This service removes access to queues within system queue list 5 from the calling program.

Call	Entry	Exit
SVC I$SQL5R	None	R0 = 8xxx error

.11. I$STAPN—Return Procedure ICW

Returns a procedure ICW (from the procedure use table).

Call	Entry	Exit
SVC I$STAPN	R1 = run number (0 if the calling run) R2 = procedure number	R0 = 8xxx error R0 = 0 successful R1 = procedure ICW

7.7.12. I$STAPP—Return Port Processor ICW

Returns a port processor ICW (from the PP instrumentation table).

Call	Entry	Exit
SVC I$STAPP	R1 = run number (or 0 if calling run) R2 = port number (a value of OFFFF causes a logical sum of all PP ICWs for ports assigned to the calling run to be returned in R1)	R0 = 8xxx error R0 = 0 successful R1 = PP ICW

7.7.13. I$STARUN—Return Run ICW

This returns the specified run's ICW (from PCR 9).

Call	Entry	Exit
SVC I$STARUN	R1 = run number (or 0 if calling run)	R0 = 8xxx error R0 = 0 successful R1 = run ICW

8. IPM Services

The services included here provide an inter-program message (IPM) communication facility. This facility supports cooperating but separate programs working in unison.

Normally a program includes one or more processes that interface to each other through CPA queues. On large systems, it may be better to break the system into a number of single or multitasking programs. It is easy to build and execute a multitasking program.

The cooperating programs require the IPM services to enter into a dialog, because the normal protection mechanisms of DCP/OS and CPA specifically prevent such dialog from taking place.

All IPM connections are one-way (transmitter to receiver) and the IPM protocol is purposely simple. The primitives are connect/disconnect type. Queues pass messages down an IPM pipe.

The IPM user completely defines the message content, thereby allowing any higher level protocol to be superimposed.

Typically, one program is the nucleus for such a large system. It registers itself as a receiver. Other programs then connect to the nucleus and register themselves as receivers opening a two-way connection.

Several IPM services either require, or return, lengthy parameters. For this reason, the parameters are passed in a packet which is specified in Appendix A. The packet cannot be in the first four SDRs, since they are architecturally cleared upon invoking a service.

The SVC mechanism invokes every IPM service. The service function code definitions are found in AASERVDEF (see Appendix C). All other IPM-related equates (IPM packet, contingency types, and so on) are found in the definition element, AAIPMDEF.

The IPM services preserve all user registers, except as stated in the individual service specifications. When you invoke any IPM service, five free user SDRs are required, starting at SDRIPM (currently SDR25).

Example

```
LOADC     R2,mypkt          . Point to IPM packet
SVC       IPM$CHK           . Go check it out!
```

7.8.1. IPM$CHK—Check Status of IPM Receiver

This service checks the status of the specified IPM receiver by name. This service typically inquires about the existence of a receiver before attempting a connection or registration.

A successful return from this service provides all information in the IPM interface packet.

Call	Entry	Exit
SVC IPM$CHK	R2 = virtual address of IPM packet IP$NAME = receiver name	R0 = 8xxx error R0 = 0 successful R1 = IPM-ID

7.8.2. IPM$CLOS—Close an IPM Connection

This service closes down a currently registered IPM receiver. This terminates all active connections to this receiver.

Call	Entry	Exit
SVC IPM$CLOS	R1 = IPM-ID	R0 = 8xxx error R0 = 0 successful R1 = IPM-ID

7.8.3. IPM$CONN—Connect to IPM Receiver

This service connects to a currently registered IPM receiver (IP$NAME). The optional contingency queue must be a literal type queue (LIT) and alerts the contingency handler when a state change occurs. Refer to Section 4 for details of this mechanism.

If a successful connection is made, an IPM pipe queue is entered in the queue list of procedure IP$PN at position IP$QLX. The caller may then queue directly to the paired IPM receiver.

The IP$REPLY name is optional. It is presented to the paired receiver in the contingency packet and may be used at the system designer's discretion.

It can, in fact, provide the necessary information for the receiver to open a reverse connection and establish a two-way pipe.

Call	Entry	Exit
SVC IPM$CONN	R2 = virtual address of IPM packet IP$NAME = target name (8 characters) IP$PN = procedure IP$QLX = required QL index IP$QNC = contingency QN IP$REPLY = reply name (8 characters)	R0 = 8xxx error R0 = 0 successful R1 = IPM-ID

.4. IPM$DISC—Disconnect an IPM Connection

This service disconnects a currently active IPM connection from the transmitter end. No IPM interface packet is required.

Call	Entry	Exit
SVC IPM$DISC	R1 = IPM-id	R0 = 8xxx error R0 = 0 successful R1 = IPM-id

.5. IPM$FREE—Free an IPM Connection

This service frees a currently active IPM connection from the receiver end. No IPM interface packet is required.

Call	Entry	Exit
SVC IPM$FREE	R1 = IPM-ID	R0 = 8xxx error R0 = 0 successful R1 = IPM-ID

7.8.6. IPM$RCV—Establish an IPM Receiver

This service registers an IPM receiver. The optional contingency queue must be a literal type queue (LIT) and alerts the contingency handler when a state change occurs. Refer to Section 4 for details of this mechanism.

If a contingency queue is not specified, the IPM receiver has to recognize the first message received and take appropriate action.

This service creates IP$MXCN number of queues of type IP$ queue, each of size IP$QSIZ, and enters them into the queue list of procedure IP$PN, starting at entry number IP$QLX. One queue per potential connection to this IPM receiver is therefore created. All messages from an IPM transmitter on a given connection are queued sequentially to the same queue. The actual queue is assigned to the connection at connect time.

The caller may optionally nominate a cascade queue (IP$SQLX) to limit the number of dispatches caused by queued IPM messages.

Call	Entry	Exit
SVC IPM$RCV	R2 = virtual address of IPM packet IP$NAME = receiver name (8 characters) IP$PN = receiver PN IP$QLX = QLx of 1st connection IP$QNC = contingency queue QN IP$UNPRC = initial pipe queue threshold IP$MXCN = maximum connections IP$QSIZ = queue size IP$QTYP = queue type IP$SQLX = SQL1 index (0 if not cascaded)	R0 = 8xxx error R0 = 0 successful R1 = IPM-id

8.7. IPM$STAT—Get an IPM Connection Status

This service gets the status of the specified IPM connection by IPM-ID. The relevant fields for a receiver, specific receiving connection, or specific transmitting connection are set up in the user packet on a successful return.

Call	Entry	Exit
SVC IPM$STAT	R1 = IPM-ID R2 = virtual address of IPM packet	R0 = 8xxx error R0 = status success

7.9. Line Module Services

Each communications port of a DCP has an associated line module. Line modules may be hard-wired, PROM, or loadable. These services provide the user program with a convenient mechanism for determining the line module type and loading the line module with a specified program.

All line module services create a temporary dynamic segment. If the SST is full, an error is returned. The line module services destroy user registers R0 through R5.

The SVC mechanism is used to invoke every line module service. The function code definitions are found in AASERVDEF in Appendix C.

Example

```
LOAD      R2,myport          . Get port number
SVC       LM$GLMID           . Go get LM-id
```

7.9.1. LM$GLMID—Get Line Module Status

This service retrieves the status of the line module and the line module microcode for the requested port. If the line module is a loadable type, then a short reload is automatically attempted. The returned status includes the results of this short reload attempt.

Call	Entry	Exit
SVC LM$GLMID	R2 = port number SDRs 15,16,17,18,19,26,27 must be free	R0 = 8xxx error R0 = 0 successful R2 = port number R3 = microcode-id R4 = line module dentifier Destroys R1,R5

9.2. LM$LOAD—Load Line Module

This service loads a line module (if loadable) and returns the status of the line module and the line module microcode. If the line module is of a loadable type then a short reload is automatically attempted. If the short reload is unsuccessful or if the caller requests a forced load (LM$FORCE), a full (long) microcode load is performed. The returned status includes the results of a long or short load.

The line module code programs are assumed to be online in the file SYS$*SYSLMC. Each line module program is held in an element in this file, named according to the MICROCODE-ID (which can be listed with @PORT,M) as follows:

```
SYS$*SYSLMC.LMC-24  [MICROCODE-ID 24 = BASIC SYNCHRONOUS]
```

Call	Entry	Exit
SVC LM$LOAD	R2 = port number R3 (bits 15-8) = function flags R3 (bit LM$FORCE) = force long load R3 (bit LM$QFE) = load with qual*file.elt (R1 = virtual address of string containing qual*file.eltname) R3 (bit LM$UNLOAD) = put LM in unloaded state R3 (bits 7-0) = line module code identifier SDRs 15,16,17,18,19,26,27 must be free	R0 = 8xxx error R0 = 0 successful R2 = port number R3 = microcode-id R4 = line module identifier Destroys R1,R5

7.10. Dispatch Services

The DCP/OS provides services to enable asynchronous scheduling of procedures. The requester can schedule itself or another procedure. The target procedure can be eligible for immediate execution or it can be delayed for either a particular time of day or a given amount of time.

The SVC mechanism invokes every dispatch service. The function code equates are found in AASERVDEF (see Appendix C).

The process-ID (PID), referred to by many of the dispatch services, uniquely identifies a task within a run. The DCP/OS dynamically assigns the process-ID. The PC$WHO service (7.10.24) determines the process-ID.

The dispatch services preserve all user registers, except as stated in the individual service specifications.

Example

```
LDK     R2,1000             . Set R2,R3 = pause time (msecs)
SVC     PC$PAUSE            . Go pause for a second
```

7.10.1. PC$CBUG—DEBUG Contingency-Suspended Process

This service sends a contingency-suspended process to the DCP/OS debug handler. The PID identifies the suspended process.

Call	Entry	Exit
SVC PC$CBUG	R1 = process-ID (PID)	R0 = 8xxx error R0 = 0 successful

7.10.2. PC$CKILL—Kill Contingency-Suspended Process

This service kills a contingency-suspended process. The PID identifies the suspended process.

Call	Entry	Exit
SVC PC$CKILL	R1 = process-ID (PID)	R0 = 8xxx error R0 = 0 successful

0.3. PC$CMOD—Modify Contingency-Suspended Process

This service modifies the context of a contingency-suspended process. The user registers and the restart address are the only fields changed. The contingency status packet provides new values. See Appendix A for a description of the contingency packet.

Call	Entry	Exit
SVC PC$CMOD	R1 = process-ID (PID) R2 = packet address	R0 = 8xxx error R0 = 0 successful

0.4. PC$CRD—READ Virtual Space from Suspended Process

This service permits the reading of the virtual space for the specified contingency-suspended process. An error is returned if the virtual space in the suspended process is not present.

Call	Entry	Exit
SVC PC$CRD	R1 = process-ID (PID) R2 = callers virtual address to receive data R3 = virtual address in suspended process R4 = length (in words) of data to copy	R0 = 8xxx error R0 = 0 successful

7.10.5. PC$CREG—Register a Contingency Handler

This service registers a contingency handling procedure for the specified type. If the type is CON$NONE, then all contingency handling registration is removed and the R2 parameter is not relevant.

The contingency queue must be a literal type queue (LIT). It alerts the contingency handler when an error occurs. Refer to Section 4 for details of this mechanism.

Call	Entry	Exit
SVC PC$CREG	R1 = contingency type CON$ALL = service all types CON$NONE = no contingency handling R2 = QN of contingency literal queue	R0 = 8xxx error R0 = 0 successful

7.10.6. PC$CRES—Restart Contingency-Suspended Process

This service restarts a process suspended due to a contingency. The PID specifies the required process.

Call	Entry	Exit
SVC PC$CRES	R1 = process-ID (PID)	R0 = 8xxx error R0 = 0 successful

7.10.7. PC$CRS—READ Contingency-Suspended PMAST

This service reads the PMAST of a contingency-suspended process.

Call	Entry	Exit
SVC PC$CRS	R1 = process-ID (PID) R2 = address to store list of PMAST SSNs	None

).8. PC$CSTAT—Get Contingency Status

This service retrieves the status of a process suspended due to a contingency. The PID specifies the required process.

The contingency status is returned in the contingency status packet. The virtual address is provided in R2 and must be at least 512 bytes long. See A.10 for a description of the contingency packet.

Call	Entry	Exit
SVC PC$CSTAT	R1 = process-id (PID) R2 = packet address	R0 = 8xxx error R0 = 0 successful

).9. PC$CWFE—Clear Wait for Event

This service takes a queue, previously used for Wait For Event (PC$WFE), out of the WFE mode. The queue's SAI is restored (to subsequently schedule the caller). Control is returned immediately to the calling procedure.

Call	Entry	Exit
SVC PC$CWFE	R2 = QLx R3 = SAI Parameter	R0 = 8xxx error R0 = 0 successful

).10. PC$CWT—Write Virtual Space in Suspended Process

This service writes data into the virtual space of a suspended process. An error results if the required space in the suspended process is not present.

.10.11. PC$IREG—Register for Input Contingency

This dispatch service registers a calling program for dispatch in the case of an @@X C keyin. When the keyin occurs, the selected procedure is dispatched with the ST$XC parameter in R1.

Call	Entry	Exit
SVC PC$IREG	R1 = GPL(x) of procedure to dispatch	R0 = 8xxx error R0 = 0 successful

.10.12. PC$MRET—Modify Return Address

This service modifies the current virtual return address on the process control stack.

Call	Entry	Exit
SVC PC$MRET	R1 = new return address	R0 = 8xxx error R0 = 0 successful R1 = old return address

.10.13. PC$POPS—Pop SRTN Entries Off the Stack

This service pops the specified number of SRTN entries off the process control stack.

Call	Entry	Exit
SVC PC$POPS	R1 = number of entries to pop	R0 = 8xxx error R0 = 0 successful R1 = entries popped

.10.14. PC$PAUSE—Suspend Process

This service suspends the calling process for the specified number of milliseconds.

Note: If the delay time is specified as zero, it is replaced with a system-defined default value (currently 50 milliseconds).

Call	Entry	Exit
SVC PC$PAUSE	R2,R3 = delta time (milliseconds) to suspend.	None

.10.14a PC$PKILL — Register for Program Kill Notification

Register for notification by DCP/OS whenever a program is killed. DCP/OS dispatches the registered procedure with the effected run number in the R1 dispatch parameter.

Call	Entry	Exit
SVC PC$PKILL	R1 = GPLx of the PN to be scheduled	R0 = 0 if registration is successful = error code otherwise

'.10.15. PC$SCSD—Set Common SDs

This service stores the specified segment descriptors (SDs) for subsequent loading on every dispatch for the current program. The SDs are loaded into SDR4 and SDR5. The program can have common data visible to every dispatched process without having to load the segments each time.

This service can be found most useful in setting up global work areas.

Either one or two segments can be specified; each segment, if specified, must be resident and present. The virtual addresses in R2 and R3 may not specify SDRs 0-3 (VA 0000-1800); therefore, a value of zero may indicate that a segment is not required.

Call	Entry	Exit
SVC PC$SCSD	R2 = virtual address to use for SD4 R3 = virtual address to use for SD5	R0 = 8xxx error R0 = 0 successful

Call	Entry	Exit
SVC PC$SCSD	R2 = virtual address to use for SD4 R3 = virtual address to use for SD5	R0 = 8xxx error R0 = 0 successful

7.10.16. PC$SKAT—Schedule Procedure at Absolute Time

This service schedules the specified procedure at the given time. If the specified time is already past, the DCP/OS immediately dispatches the specified procedure.

Call	Entry	Exit
SVC PC$SKAT	R1 = priority R2 = procedure (0 = self) R3 = parameter to pass in R1 R4,R5 = RTC value (milliseconds) for when PN is to dispatch.	R0 = 8xxx error R0 = 0 successful

7.10.17. PC$SKDT—Schedule Procedure after Delta Time

This service schedules a user procedure after the specified delay, which can be zero.

Call	Entry	Exit
SVC PC$SKDT	R1 = priority R2 = procedure (0 = self) R3 = parameter to pass in R1 R4/R5 = delta time (milliseconds) after which the PN is to be dispatched.	R0 = 8xxx error R0 = 0 successful

0.18. PC$SKGAT—Schedule Procedure at Time by GPLx

This service provides a function similar to PC$SKAT except that the required PN is specified indirectly by GPLx instead of the PN and priority scheduling is not possible. If the time specified is already past, the DCP/OS immediately dispatches the procedure.

Call	Entry	Exit
SVC PC$SKGAT	R1 = GPLx of PN to dispatch R2,R3 = RTC value (milliseconds) for when PN is to dispatch R4 = parameter to pass in R1	R0 = 8xxx error R0 = 0 successful

0.19. PC$SKGDT—Schedule Procedure after Delta Time by GPLx

This service provides a function similar to PC$SKDT except that GPLx (instead of direct PN) specifies the required PN and priority scheduling is not possible.

Call	Entry	Exit
SVC PC$SKGDT	R1 = GPLx of PN to dispatch R2,R3 = delta time (milliseconds) after which the PN is to be dispatched R4 = parameter to pass in R1	R0 = 8xxx error R0 = 0 successful

0.20. PC$SLEEP—Suspend Indefinitely

This service allows a calling process to suspend itself indefinitely. It can only be restarted by another process in the same program invoking the PC$WAKE service. For that reason, this service should always be preceded by a call to PC$WHO to determine the current PID. This PID should then be stored in a shared data area or queued off to a co-process with responsibility to awaken the sleeping process.

Call	Entry	Exit
SVC PC$SLEEP	None	R0 = 8xxx error R0 = 0 when the process is redispatched R1 = parameter passed by wake (when redispatched)

7.10.21. PC$SREG—Register for Status Changes

This dispatch service registers a calling program for dispatch in the case of a DCP/OS status change. The DCP/OS dispatches the registered procedure each time one of the status changes. The DCP/OS supports the following status changes:

- Throttle level change
- Date/time changed by command
- Midnight

Call	Entry	Exit
SVC PC$SREG	R1 = GPL(x) of procedure to dispatch	R0 = 8xxx error R0 = 0 successful

When a status change occurs, the selected procedure dispatches the following parameters in R1:

ST$THROT	Throttle level change
ST$DATET	Date/time changes
ST$MIDNT	Midnight

Note: *Note: Only one PC$SREG per program is allowed.*

0.22. PC$WAKE—Wake a Sleeping Process

This service wakes up a sleeping process, identified by its PID. The process must have suspended itself with PC$SLEEP and must belong to the same program as the calling procedure. The parameter value specified in R2 is passed to the target process as R1 on dispatch.

Call	Entry	Exit
SVC PC$WAKE	R1 = process-id (PID) to awaken R2 = parameter to pass as R1	R0 = 8xxx error R0 = 0 successful

0.23. PC$WFE—Suspend, Wait for Event

This service allows a calling process to suspend itself until an item is queued, without returning to the dispatcher. Therefore, the virtual environment of the process (for example, work areas) is completely frozen, ready for immediate use when control is returned.

If the specified queue is nonexistent, already armed, or not associated with this process, an error is returned. Otherwise, the current environment is saved and the process is suspended until the specified queue threshold is crossed.

Call	Entry	Exit
SVC PC$WFE	R2 = QLx R3 = queue threshold	R0 = 8xxx error R0 = 0 when the process is redispatched

0.24. PC$WHO—Determine Current Process-ID

This service returns the current PID, PN, and run-ID to the calling procedure.

This service is normally used in conjunction with the go-to-sleep (PC$SLEEP) and wake-up (PC$WAKE) services. The PID should be stored in a data area or queued off to another process which eventually invokes PC$WAKE to restart the sleeping process.

Call	Entry	Exit
SVC PC$WHO	None	R0 = 8xxx error R0 = 0 successful R1 = process-ID (PID) R2 = callers PN R3 = run-ID

7.11. PP Program Services

The DCP/OS provides services for building port processor (PP) environments, starting PPs, stopping PPs, and handling state items.

For all PP services, the indicated port is automatically but temporarily assigned to the calling program if currently free. If a more permanent assignment of the port is required, use PP$ASG/PP$FREE.

The SVC mechanism invokes every PP service. The function code definitions are found in AASERVDEF.

The PP program services preserve all user registers, except as stated in the individual service specifications.

Example

```
LOAD      R1,myport          . Pick up port number
SVC       PP$START           . Go start the PP
```

7.11.1. PP$ASG—Assign PP

This service assigns the specified port (PP) exclusively to the calling program. The port must currently be free.

Call	Entry	Exit
SVC PP$ASG	R1 = port number	R0 = 8xxx error R0 = 0 successful

11.2. PP$CNFG—Configure PP

This service builds a CPA interface control table (ICT) entry for the specified PP according to the environment specified by the PP program. If required, registers R3 through R6 are used to override the port processor program table (PPPT) visibility specification. If any of registers R3 through R6 is nonzero, it is assumed to contain the virtual address of some user space given to the PP. In either case, the following conventions apply when setting up the PP visibility, for each of the override SDRs specified by registers R3 through R6:

- Allocatable space is removed from the caller's visibility and given to the PP. (This does not include dynamic segments.)
- Segment space is given to the PP unless it is for SD0, in which case the contents are copied to an allocated buffer that is given to the PP. (This does not include dynamic segments.)
- Dynamic segments are not copied for SD0 and visibility is not removed. (That is, the PP and the calling procedure share access to these segments.)

If a queue list was created for the PP (using CPA$CPPQL), it is used. Otherwise, an empty queue list of the size specified in the PPPT entry is created. If a QUEDEF MASM procedure was used in the PP program, all static queues are copied into the created queue list.

Note: You can find the required PP program number as an entry parameter in R2 by using dictionary services.

Call	Entry	Exit
SVC PP$CNFG	R1 = port number R2 = PP program number R3 = SD0 override virtual address R4 = SD1 override virtual address R5 = SD2 override virtual address R6 = SD3 override virtual address	R0 = 8xxx error R0 = 0 successful

11.3. PP$CSTART—Configure and Start PP

This service builds an ICT entry for the specified PP according to the environment needed for the program specified. This service also commands the PP to start by setting the C-bit in the system control table (SCT) for the specified PP. See PP$CNFG (7.11.2) for a description of the configuration functions of this service.

Call	Entry	Exit
SVC PP$CSTART	R1 = port number R2 = PP program number R3 = SD0 override R4 = SD1 override R5 = SD2 override R6 = SD3 override	R0 = 8xxx error R0 = 0 successful

7.11.4. PP$ELIM—Eliminate PP

This service releases the PP environment and clears the ICT entry for the specified port. The use counts of all segments used by the PP are decremented and all allocatable space is released. The use counts of all queues in the PP queue list are decremented and the PP queue list is released.

Call	Entry	Exit
SVC PP$ELIM	R1 = port number	R0 = 8xxx error R0 = 0 successful

7.11.5. PP$FREE—Free PP

This service frees the specified port (PP) from the calling program. If the port is currently active, it is stopped and eliminated.

Call	Entry	Exit
SVC PP$FREE	R1 = port number	R0 = 8xxx error R0 = 0 successful

1.6. PP$FREEAS—Free a PP Assigned to a Program

This service stops and frees the first PP currently assigned to the specified program. This service is designed for internal DCP/OS use, but may be called by a user program. When called by a user program, the run-ID parameter (R1) is ignored.

In order that ports may be run down gradually (not all at once), each invocation of this service stops and frees the first port found that belongs to the specified run. Therefore, repeated calls to this service, with appropriate pauses or calls to PP$GETSI in between, eventually stop and free all PPs belonging to the target run. When no ports are found, the return register (R0) is set to EC$PP+ER$PPID.

Call	Entry	Exit
SVC PP$FREEAS	R1 = run-ID (DCP/OS call only)	R0 = 8xxx error R0 = port number successful

..7. PP$GETLM—Get LM Microcode ID from ICT

This service retrieves the line module-ID and the line module (LM) microcode-ID from the ICT for the requested port.

Call	Entry	Exit
SVC PP$GETLM	R1 = port number	R0 = 8xxx error R0 = 0 successful R2 = Line module-ID R3 = LM code-ID R4 = Run-id of owner

7.11.8. PP$GETSI—Get PP Status and State Item

This service returns the status of a PP, a state item (if so requested), and any PP MCTs as indicated by the request flags in R2.

The virtual address specified in R2 must be on an SDR boundary and that SD. The following 10 SDs are used to return the state item and any MCT/buffers held by the PP, if so indicated by the setting of the request flags in R2.

Call	Entry	Exit
SVC PP$GETSI	R1 = port number R2 = bits 15-11 virtual address for state item (or zero) bit 0 = 1 return output MCT bit 1 = 1 return input MCT bit 2 = 1 return line output MCT bit 3 = 1 return line input MCT	R0 = bit 15 - invalid PP bit 14 - PP configured bit 7 - PP active bit 6 - state item present bit 5 - output MCT returned bit 4 - input MCT returned bit 3 - PP C bit bit 2 - PP R bit bit 1 - PP A bit

7.11.9. PP$PUTLM—Store LM Microcode ID in ICT

This service stores the line module-ID. The line module microcode is in the ICT for subsequent reference.

This service is called on behalf of the user when any line module service (LM$) is invoked.

Call	Entry	Exit
SVC PP$PUTLM	R1 = port number R2 = line module-ID R3 = line module code-ID	R0 = 8xxx error R0 = 0 successful

..10. PP$START—Start PP

This service commands the PP to start by setting the C-bit in the SCT for the specified PP.

Call	Entry	Exit
SVC PP$START	R1 = port number	R0 = 8xxx error R0 = 0 successful

.11. PP$STATUS—Get PP Status

This service returns the C, R, and A bits from the SCT corresponding to the requested port.

Call	Entry	Exit
SVC PP$STATUS	R1 = port number	R0 = 8xxx error R0 = status successful bit-14 = port is configured bit-7 = port is active bit-3 = C-bit bit-2 = R-bit bit-1 = A-bit

.12. PP$STOP—Stop PP

This service commands the PP to stop by clearing the C-bit in the SCT for the specified PP.

Call	Entry	Exit
SVC PP$STOP	R1 = port number	R0 = 8xxx error R0 = 0 successful

'.11.13. PP$STOPAS—Stop a PP Assigned to a Program

This service stops the first PP currently assigned to the specified program. This service is designed for internal DCP/OS usage, but may be called by a user program. When called by a user program, the run ID parameter (R1) is ignored.

Each invocation of this service stops the first port found that belongs to the specified run. Therefore, repeated calls to this service, with appropriate pauses or calls to PP$GETSI in between, eventually stop all PPs belonging to the target run. When no ports are found, the return register (R0) is set to EC$PP + ER$PPID.

Call	Entry	Exit
SVC PP$STOPAS	R1 = run-ID (DCP/OS call only)	R0 = 8xxx error R0 = port number successful

'.11.14. PP$THROT—Establish PP Throttle

This service establishes the throttle value for a PP. The default value is used for any PP for which this service is not called.

Call	Entry	Exit
SVC PP$THROT	R1 = port number R2 = throttle value	R0 = 8xxx error R0 = 0 successful

.12. DCP/OS Level 5R1 Microcode support

The following services are used by Telcon and program products to support microcode on the DCP/30 and DCP/50. These services replace the functions performed by the TELSERV module of Telcon. If your site puts microcode level 14R1 on these DCP models without changing to the CD5R1 version of Telcon, your programs must use these calls.

Note: In addition to the microcode support services described in this section, you must also include CPA$DASVC.

12.1. CPA$SDRD – Set Up Read Access to a Segment

This service sets the read bit in the mode field for the indicated SDR (SDRs 0-3 are not allowed). You must verify that the segment descriptor is present.

Call	Entry	Exit
SVC CPA$SDRD	R1 = Virtual address indicating the SDR to be modified	R0 = 0 if successful R0 = error code otherwise

12.2. CPA$QSMD – Set Queue Mode

This service sets the mode of the specified queue to the indicated value. The queue must be an MCT type queue and must be empty.

Call	Entry	Exit
SVC CPA$QSMD	R1 = QLx R2 = (7-0) New mode field	R0 = 8xxx error R0 = 0 successful

.12.3. PC$CSTK – Return Information from Caller's Stack

This service returns information from the process control stack entry of the caller's stack. Such a stack may look like the following:

CALL PROC2 from procedure PROC1
SVC PC$CSTK from procedure PROC2

The returned information describes the CALL from PROC1 (instead of PROC2). No check is made that the preceding stack entry represents a CALL and not an SCALL or an SVC instruction.

Call	Entry	Exit
SVC PC$CSTK	None	R0-R1 = Program state word (PSW) following the CALL PROC2 instruction in PROC1 (see above) R2 = Initial PN of task R3 = Procedure number (PN) of PROC1

.12.4. PC$RAEQL – Read an Alternate Environment QL

Return the queue list entry and the queue number for the indicated PP alternate environment queue.

Call	Entry	Exit
SVC PC$RAEQL	R1 = PPID R2 = Alternate environment number R3 = QLx within the alternate environment queue list	R0 = 0 if successful = error code if not successful R1 = Queue number R2-R3 = Queue list entry

.12.5. PC$RICT – Read an ICT

Read the indicated interface control table (ICT) entry and copy it to a virtual address specified by the caller.

Call	Entry	Exit
SVC PC$RICT	R1 = PPID to indicate which ICT entry to read R2 = Virtual address at which to store a copy of the ICT entry for the indicated PPID	R0 = 0 if successful = Error code if not successful

.12.6. PC$RPPQL – Read a PP Queue List

Return the queue list entry and the queue number for the indicated PP queue.

Call	Entry	Exit
SVC PC$RPPQL	R1 = PPID R2 = QLx within the PP Queue List	R0 = 0 if successful = Error code if not successful R1 = Queue number R2-R3 = Queue list entry

.12.7. PC$RPT – Read a Procedure Table

Read the indicated procedure table (PT) entry and copy it to a virtual address specified by the caller.

Call	Entry	Exit
SVC PC$RPT	R1 = PN for the PT entry to read R2 = Virtual address at which to store a copy of the PT entry for the indicated PN	R0 = 0 if successful = Error code if not successful

7.12.8. PC$RQH – Read a Queue Header

Read the indicated queue header, including the two-word prefix, and copy it to the virtual address specified by the caller. The real byte address of the queue (excluding the prefix) is returned.

Call	Entry	Exit
SVC PC$RQH	R1 = Queue Number (QN) to read R2 = Virtual address at which to store a copy of the Queue Header for the indicated QN.	R0 = 0 if successful = error code if not successful R2-R3 = Real byte address of the queue

7.12.9. PC$RQL – Read a Procedure's Queue List

Return the indicated queue list entry and the corresponding queue number to the caller.

Call	Entry	Exit
SVC PC$RQH	R1 = Procedure Number (PN) owning the Queue List R2 = Queue List Index (QLx)	R0 = 0 if successful = error code if not successful R1 = Queue Number (QN) R2-R3 = Queue List entry

12.10. PC$RSST – Read an SST

Read the indicated SST entry and copy it to a virtual address specified by the caller. Return the number of words actually defined for this segment.

Call	Entry	Exit
SVC PC$RSST	R1 = SSN to be read R2 = Virtual address at which to store a copy of the SST entry for the indicated SSN	R0 = 0 if successful = error code if not successful R1 = Number of words in the segment

Section 8
Common Utility Subroutines

This section describes common utility subroutines. The segment SAGSUB contains subroutines that pertain to string manipulation and INFOR$/IPF structure creation/utilization. To use these routines, the segment must be loaded into the user's virtual memory at location SDR3 (as part of the CALL visibility). You may then use SCALL to call the routines as needed.

If you wish to use the SAGSUB routines in a program, you need only refer to the segment in a PROCDEF statement. The @BUILD utility program generates the request linkage to this segment. Subroutine names are in the definition element AASUBENT, which must be included in the user program.

8.1. S$ADTOB—ASCII Decimal to Binary

This routine takes a four-digit ASCII decimal string, left-justified and blank-filled, and converts it to a binary value.

Call	Entry	Exit
SCALL S$ADTOB	R2 = upper two digits (must be left-justified blank-filled) R3 = lower two digits	R0 = zero if value was legal decimal number R0 = -1 if illegal R1 = binary value Destroys: R4,R5

8.2. S$ADTOBE—ASCII Decimal to Binary, Extended

This routine takes an ASCII decimal string, up to nine digits long, and converts it to a binary value.

Call	Entry	Exit
SCALL S$ADTOBE	R10,11 = pointer to ASCII string	R0 = zero if value was legal decimal number R0 = -1 if illegal R4,5 = binary value R11 = updated pointer Destroys: R2,R3

3. S$AHTOB—ASCII Hexadecimal to Binary

This routine takes a four-digit, left-justified and blank-filled ASCII hexadecimal string and converts it to a binary value.

Call	Entry	Exit
SCALL S$AHTOB	R2 = upper two digits (must be left-justified blank-filled) R3 = lower two digits	R0 = zero if value was a legal hexadecimal string R0 = -1 if illegal R1 = binary value R2 = unchanged R3 = unchanged R4 = destroyed

4. S$AHTOBE—ASCII Hexadecimal to Binary, Extended

This routine takes an ASCII hexadecimal string, up to eight characters long, and converts it to a binary value.

Call	Entry	Exit
SCALL S$AHTOBE	R10,11= pointer to ASCII string	R0 = zero if value was legal hexadecimal number R0 = -1 if illegal R4,5 = binary value R11 = updated pointer Destroys: R2,R3

8.5. S$AQUAL—Copy the Assumed Project-ID into FRP

This routine copies the current assumed project-ID RC$AQUAL (assumed default qualifier) from the caller's run control table (RCT) entry into the specified file request packet (FRP).

Call	Entry	Exit
SCALL S$AQUAL	R8 = virtual address of file request packet (FRP) R13 = virtual address of current run control table (RCT) entry	R2 = destroyed

8.6. S$ATOB—ASCII to Binary

This routine converts an ASCII decimal value of two digits into a binary value.

Call	Entry	Exit
SCALL S$ATOB	R1 = two ASCII decimal digits (leading zeros required)	R0 = binary value or -1 if unsuccessful R1 = unchanged R2 = destroyed R3 = destroyed

⁷. S$BLDFRP—Build File Request Packet

This routine builds the qualifier, file, and element name fields in a standard FRP from a parameter contained in an INFOR$ packet. If the caller has visibility to the current RCT entry, the alternate entry point S$BLDFRPR may be used with the RCT virtual address in R13.

Call	Entry	Exit
SCALL S$BLDFRP	R1 = command line spec. number R8 = virtual address of FRP R10 = virtual address of INFOR$ packet	R0 = 0 if ok (else R1=error code) Destroys: R1-R5,R9,R11

⁾. S$BREL—Deallocate a Buffer Area

This routine deallocates an MCT-type message (five SDRs) from the virtual memory address supplied. It requires the address to be pointing to the message (data) area and not the MCT.

If R10 = 0, then no attempt at deallocation is made.

Call	Entry	Exit
SCALL S$BREL	R10 = address of buffer (Example. AV$MSG)	R10 = zero Destroys: R0 R1 R2

). S$BTOA—Binary to ASCII Decimal

This routine takes a binary value and converts it into an ASCII string. The data is zero-filled and right-justified (with a blank or sign at the leftmost byte).

Call	Entry	Exit
SCALL S$BTOA	R3 = binary value	R0(U)= thousands digit R0(L)= hundreds digit R1(U)= tens digit R1(L)= ones digit R2(U)= blank or '-' R2(L)= ten thousands digit R3 = destroyed R4 = destroyed R5 = original value of R3

8.10. S$BTOAE—Binary to ASCII Decimal, Extended

This routine takes a 24-bit binary value and converts it into an ASCII decimal string. The data is zero-filled and right-justified.

Call	Entry	Exit
SCALL S$BTOAE	R4,R5 = binary value (24 bits only)	R0(U)= ten millions digit R0(L)= millions digit R1(U)= hundred thousands digit R1(L)= ten thousands digit R2(U)= thousands digit R2(L)= hundreds digit R3(U)= tens digit R3(L)= ones digit Destroys: R4,R5,R6

8.11. S$BTOAH—Binary to ASCII Hexadecimal

This routine takes a binary value and converts it into a displayable ASCII hexadecimal string that is right-justified and zero-filled.

Call	Entry	Exit
SCALL S$BTOAH	R3 = binary value to convert	R0 (U) = first nibble R0 (L) = second nibble R1 (U) = third nibble R1 (U) = fourth nibble R2,R3,R4 = destroyed

.2. S$BYST—Store Characters

This set of mini-subroutines stores characters from registers in the correct sequence, left to right. Normal byte store instructions store only the lower byte of a register. This method allows the register to be "packed" with ASCII data.

Call	Entry	Exit
SCALL S$BYSTn	Variable, depends on subroutine. (See the following subroutine names.)	R11 = Byte offset is modified so that it points to next store position in buffer.

The subroutine names are as follows:

S$BYST0	Stores byte upper (R0) and then byte lower (R0) into the buffer pointed to in R10/R11 (Im byte).
S$BYST1	Stores byte upper (R1) and then byte lower (R1) into the buffer pointed to by R10/R11 (Im byte).
S$BYST2	Stores byte upper (R2) and then byte lower (R2) into the buffer pointed to by R10/R11 (Im byte).
S$BYST3	Stores byte upper (R3) and then byte lower (R3) into the buffer pointed to by R10/R11 (Im byte).
S$BYST01	Combination of S$BYST0 and S$BYST1.
S$BYST23	Combination of S$BYST2 and S$BYST3.

3. S$CANBLD—Build a Message Skeleton

This routine takes the number supplied and assumes the canned message segments are available at PSTDEF entries zero through *n*.

There is an additional entry point (S$CANBLDU) through which the caller gives a specific procedure segment table (PST) index in R2.

The message number supplied is used to index a list of actual message addresses and lengths. The list must be at the start of the message segment and the first entry in the list must be a dummy entry, the first word of which defines the number of messages in the segments. If the message address (base 0) exceeds X'800', it then presumes the segment given by the following PST entry.

For example:

AARUNDEF contains equates for message numbers, such as

```
SYSCM69  EQU  69
```

The canned message segment contains:

```
GEN       maxmsgs          . Maximum message number
GEN       0                .
GEN       SYSCM69A         . Address of string
GEN       SYSCM69L         . Byte length of string
....".....

SYSCM69A 'This is the canned message'
SYSCM69L EQU ($-SYSCM69A)*2 .
```

Segment descriptor register (SDR) AV$SUB must be free on entry.

Call	Entry	Exit
SCALL S$CANBLD	R1 = canned message number R2 = PST index (Only for S$CANBLDU)	R10 = address of AV$AMSG R11 = pointing at MH$FLTH*2 R12 = remaining bytes in granule MH$MTYP = MH$MCO MH$DBO = MH$FLTH*2 MH$DBC = byte count of the text Destroys: R0-R2 and R10-R12

14. S$CDTOB—Convert ASCII Decimal Character to Binary

This routine converts a single decimal ASCII character to binary, checking that a valid character was supplied.

Call	Entry	Exit
SCALL S$CDTOB	R2 = ASCII character	R0 = 0 if ok R2 = binary value of decimal character Destroys: R0

15. S$CHKNAME—Check Validity of ASCII Name

This routine checks the validity of a given ASCII name by looking for invalid characters. A lists of valid characters include

- Alpha characters (A through Z or a through z) (lowercase is converted to uppercase)
- Numeric characters (0 through 9)
- Dollar ($) and hyphen (-)
- The at symbol (@) after the first character
- Trailing spaces (the first character may not be a space).

Call	Entry	Exit
SCALL S$CHKNAME	R2 = virtual address of name field R4 = length of name field in bytes	R0 = 0 if name is valid otherwise not zero

8.16. S$CHTOB—Convert ASCII Hexadecimal Character to Binary

This routine converts a single hexadecimal ASCII character to binary, checking that a valid character was supplied.

Call	Entry	Exit
SCALL S$CHTOB	R2 = ASCII character	R0 = 0 if ok R2 = binary value of hexadecimal character Destroys: R0

8.17. S$CINF/S$CIPF—Build INFOR$/KEYWORD Structure

This routine builds a standard INFOR$ or KEYWORD structure. The explanation and definition of the structure are given in Section 2.

Call	Entry	Exit
SCALL S$CINF or SCALL S$CIPF	R6 = address of the console input message R10 = address where to create packet message requires previous SD (for MCT) and the next four to be empty	R0 = undefined R1 = zero if good packet R1 = error code if packet not built SYSCM43 (Invalid parameter found) R2 = undefined R3 = undefined R10 = address of structure or zero if bad (& INFOR$ NOT allocated) R11 = destroyed R12 = destroyed Destroys R0-R3, R10-R12

MH$MTYP	MH$MINFOR
IF$IPF	Set if keyword format
IF$FGN	Set if any specifications refer to a host filename
MH$DBO	MH$FLTH*2
MH$DBC	byte count of structure's data
MH$SUPPA	packet flags
MH$SUPPB	Spec 0 options if S$CINF call
MH$SUPPC	Spec 0 options continued...

All other header words are transferred.

.8. S$CONSCK—Check Workstation PDT Entry

This routine searches the physical device table (PDT) for the requested workstation entry. The PDT segment (SPDT) must be in the caller's visibility at virtual address X'3000'.

Call	Entry	Exit
SCALL S$CONSCK	R1 = workstation number PD$ = PDT segment	R5 = virtual address of PDT entry (0 if not found) Destroys: R0,R2,R3

.9. S$DATEA—Build Date Message

This routine builds the current date value into the caller's message space. The message pointed to by R10 must be in standard message format containing four "@" fields, as follows:

```
@@@@  @@  @@  @@
day   dd  mm  yy
```

Call	Entry	Exit
SCALL S$DATEA	R2 = year (1900=0) R3 = month (Jan=1) R4 = day (1 through 31) R10 = message buffer address R11 = byte offset to start of day-field	R10 = unchanged R11 = points past last character inserted Destroys: R0-R6

8.20. S$DTMSG—Build Date and Time Message

This routine builds the current date and time values into the caller's message space. The message pointed to by R10 must be in standard message format containing seven "@" fields, as follows:

```
@@@@ @@ @@@@ @@   @@ @@ @@
day  dd mm  yy   hh mm ss
```

Field ONE - day (four @ characters, last character is a blank)
Field TWO - date (two @ characters, 1 through 31)
Field THREE - month (four @ characters, last character is blank)
Field FOUR - year (two @ characters, 2 digits)
Field FIVE - hours (two @ characters)
Field SIX - minutes (two @ characters -> format is hh:mm:ss)
Field SEVEN - seconds (two @ characters)

Call	Entry	Exit
SCALL S$DTMSG	R10 = message buffer address	R10 = unchanged R11 = points past last character inserted Destroys: R0-R6

8.21. S$FINDC—Search for Character

This routine searches the specified buffer for the specified character.

Call	Entry	Exit
SCALL S$FINDC	R1 = required character R10,R11 = initial buffer pointer MH$DBC = data byte count	R0 = number bytes remaining (0 = not found) R11 = points to next byte Destroys: R0,R2

22. S$GETM—Allocate Message in Primary

This routine allocates an MCT in AV$MCT and a subsegmented data space of the size supplied into AV$MSG. The message size is rounded up to an integral number of granules (128-byte buffers).

Call	Entry	Exit
SCALL S$GETM	R2 = message size	R2 = unchanged R10 = address of AV$MSG R11 = zero R12 = total bytes acquired

23. S$GETMA—Allocate Message in Alternate

This routine allocates an MCT in AV$AMCT and a subsegmented data space of the size supplied into AV$AMSG. The message size is rounded up to an integral number of granules (128-byte buffers).

Call	Entry	Exit
SCALL S$GETMA	R2 = message size	R2 = unchanged R10 = address of AV$AMSG R11 = zero R12 = total bytes acquired

8.24. S$GIPF—Get Parameter Type - KEYWORD Structure

This routine searches for the first occurrence of the IF$ type supplied.

Call	Entry	Exit
SCALL S$GIPF	R2 = IF$ type requested R10 = base address of keyword packet	R2 = length of data if found R3 = IF$ type or zero if not found R10 = unchanged R11 = byte offset of first data character (or undefined) Destroys: R0-R3

8.25. S$IACBUFG—Generate Inspect/Change Buffer

This routine generates a display buffer in the format address/hexadecimal/ASCII. Debug and the utility programs LIST and DISK use this format. The output buffer (R10/R11) must be allocated before calling this subroutine.

Call	Entry	Exit
SCALL S$IACBUFG	R7 = address mode (0 = word 1 = byte) R8,R9 = address of data to convert R10,R11 = virtual address of output buffer R12,R13 = address for display R15 = data byte count	R11 = updated Destroys: R0-R5,R12-R15

26. S$IACBUFM—Generate Modified Inspect/Change Buffer

This routine processes an input buffer (possibly modified) and generates a buffer corresponding to the original memory contents.

Call	Entry	Exit
SCALL S$IACBUFM	R4 = number of lines to skip R8,R9 = virtual address of data buffer R10,R11 = virtual address of input buffer R15 = data byte count	R0 = 0 if ok R8,R9 = address +1 of last byte inserted R10,R11 = address +1 of last byte scanned Destroys: R0-R4,R14-R15

27. S$MOVFN—Move Name String

This routine moves a name string (no embedded blanks) to another area. If a blank is encountered, the remainder of the target string is filled with the fill character supplied by the caller.

Call	Entry	Exit
SCALL S$MOVFN	R1 = number of bytes to move R2 = virtual address of name string to move R4 = character (in lower byte) to use for fill R10 = virtual address of target buffer R11 = byte offset into target buffer (lm byte)	R0 = destroyed R1 = zero R2 = unchanged R3 = destroyed R4 = unchanged R10 = unchanged R11 = points to next byte in target buffer after the move

8.28. S$MVSTR—Move Character String

This routine moves a string to another space. If the FROM string is shorter, the remaining TO string bytes are filled with the user-supplied character. If the FROM string is longer, the move stops (truncates) at the length of the TO area. See the DCP Series Implementation Reference Manual, Volume 1, Volume 2, Rev. 1, and Volume 3 (UP-12728) for a description of the Im byte function.

Call	Entry	Exit
SCALL S$MVSTR	R1 = fill character R2 = address of from string R3 = byte offset of from string (Im byte) R4 = byte length of from string R10 = address of to area R11 = byte offset of to area (Im byte) R12 = byte length of to area	R0 = destroyed R1 = unchanged R2 = unchanged R3 = destroyed R4 = destroyed R5 = destroyed R10 = unchanged R11 = points to character+1 of the area

8.29. S$NXT—Get Next Character

This routine gets the next character from the user message. It checks for end of text and returns either the character or an end-of-text status.

The end-of-text status is set on detecting any of the following:

- End of message
- Space character
- Horizontal tab character
- Null character
- Carriage return character

Call	Entry	Exit
SCALL S$NXT	R10,R11 = pointer to string	R2 = zero if end-of-text R2 = found character R11 = incremented if R2 is nonzero Destroys R2

30. S$PARS—Parse Characters

This routine takes the standard message supplied and parses information into registers for subsequent analysis. The following conditions terminate the parse scan:

- Blank character
- Comma character
- Tab character
- Any INFOR$/IPF structure control byte
- Four bytes parsed
- End of message

The data is returned in registers left-justified and blank-filled. If a terminator was encountered, the terminator register shows the character that caused the parse to stop. If the parse stopped because four data characters were loaded, the terminator register shows the next character that loads if S$PARS is called again. If the terminator was an INFOR$/IPF control byte, the terminator register is the control byte. Subsequent calls to S$PARS load more data or return with blanks if it is pointing at a terminator character. This is useful for routines that do not want to be length sensitive.

For example, in an INFOR$ packet, a user has supplied a field of one character length. The target field allows up to 12 characters. Three calls to S$PARS give the equivalent of a 12-character, left-justified, blank-filled string. If the parse was terminated by EOM, the terminator is set to a blank. Trailing blanks and tabs are skipped until a blank, tab, or comma terminator is found.

Call	Entry	Exit
SCALL S$PARS	R10 = base address of message R11 = offset to start scan MH$DBO = beginning of message MH$DBC = total byte count of text	R0 = separator character found that stopped parse R1 = destroyed R2 = first two characters of field R3 = second two characters of field R4 = undefined R10 = unchanged R11 = points to next nonblank character

8.31. S$PERCENT—Double Register Percentage Calculation

This subroutine calculates the percentage of a double register numerator and a double register denominator. If the numerator is equal to or greater than the denominator, the result is 100 percent.

Call	Entry	Exit
SCALL S$PERCENT	R0,R1 = denominator R2,R3 = numerator	R3 = whole number as percent value Destroys: R0,R1,R2,R4

2. S$PDTCHK—Check a PDT Entry

This routine searches the PDT for the requested entry. The PDT segment (SPDT) must be in the caller's visibility at virtual address X'3000'.

Call	Entry	Exit
SCALL S$PDTCHK	R1 = ordinal number R3 = PDT entry type PD$ = PDT segment	R5 = virtual address of PDT entry (0 if not found) Destroys R0,R2,R3

3. S$QUAL—Copy Project-ID into FRP

This routine copies the current project-ID RC$AQUAL (default qualifier) from the caller's run control table entry into the specified file request packet.

Call	Entry	Exit
SCALL S$QUAL	R8 = virtual address of file request packet (FRP) R13 = virtual address of current run control table (RCT) entry	R2 = destroyed

4. S$REL—Deallocate a Buffer Area

This routine deallocates an MCT-type message (five SDRs) from the virtual memory address supplied. It requires that the address be pointing to the MCT and not the message (data) area.

If R10 = 0 then no attempt is made at deallocation.

Call	Entry	Exit
SCALL S$REL	R10 = address of MCT (for example, AV$MCT)	R10 = zero Destroys R0,R1,R2

8.35. S$RJBFn—Right-Justify Blank Fill

This set of mini-subroutines stores a number, provided in registers, into the user message at the given address, right-justified and blank-filled.

Call	Entry	Exit
SCALL S$RJBFn	R0 = thousands digit/hundreds digit R1 = tens digit/ones digit R2 = /ten thousands digit R5 = binary value of number to be output R10,R11 = virtual address of output field (word/byte)	R11 = points past last inserted character Destroys R3

The actual subroutine names are

S$RJBF2	2-digit field
S$RJBF3	3-digit field
S$RJBF4	4-digit field
S$RJBF5	5-digit field

8.36. S$SEARCHC—Find Insert Character

This routine scans a buffer for an at (@) character from the current position defined by R10,R11. Refer to 8.37 for full interface details.

37. S$SEARCHM—Find Insert Character

This routine scans a buffer for an at (@) character, from the beginning of the buffer. If the character is found, the byte offset register points to it.

Call	Entry	Exit
SCALL S$SEARCHM	MH$DBC of message (number of bytes to search) R10 = base word address of message buffer R11 = byte offset into message buffer	R0 = number of bytes remaining (0 if not found) R1 = unchanged R2 = destroyed R10 = unchanged R11 = points to the character found

38. S$SINF—Search INFOR$ Structure

This routine searches an INFOR$ structure for the supplied specification/field type. If the field type supplied is zero, the first field within the specification supplied is returned.

Call	Entry	Exit
SCALL S$SINF	R2(U)= specification number to look at R2(L)= specification type to look for OR zero if first type found wanted R10 = base address of INFOR$ packet	R0 = requested spec number (copy of R2(U) on entry) R1 = requested spec type (copy of R2(L) on entry or undefined) R2 = length of field found R3 = IF$ type found or zero if not found R10 = unchanged R11 = byte offset of field requested (or undefined)

8.39. S$SIPF—Search KEYWORD Structure

This routine searches a KEYWORD structure, first for a supplied keyword and then for the supplied parameter number. If the parameter number supplied is zero, the first field of the keyword is returned.

Call	Entry	Exit
SCALL S$SIPF	R5 = address of search compare string must be 12 characters left-justified blank-filled on an even boundary R6 = parameter number to find (or zero if first) R10 = address of packet	R2 = length of parameter if found R3 = parameter type or zero if not found R10 = unchanged R11 = byte offset of parameter data Destroys or modifies R0-R4, R10, R11

8.40. S$SKIP—Skip Past Leading Spaces, Tabs, and Nulls

This routine takes the standard message pointed to by R10,R11 and skips past all spaces, tabs, and nulls.

Call	Entry	Exit
SCALL S$SKIP	R10 = base address of standard message R11 = byte offset to start of scan MH$DBO = beginning of message MH$DBC = total byte count of message	R1 = character which terminates scan R4 = offset to EOM R10 = unchanged R11 = points to first next nonblank character in message or EOM (R4 = R11)

41. S$SRE—Report System Recoverable Error

This routine takes the message supplied in AV$MSG and displays its control header plus the error code supplied to any error-logging workstation.

Call	Entry	Exit
SCALL S$SRE	R1 = error code message must be visible in AV$MSG	R0-R6 and R10-R12 destroyed original message in AV$MSG generated S.R.E. message in AV$AMSG

42. S$STRC—String Compare

This subroutine compares two strings of characters against each other. If the strings are not of the same length, the longer string must contain blanks in the remaining characters. Otherwise, a status of no match is returned.

Call	Entry	Exit
SCALL S$STRC	R2 = address of string A R3 = byte offset of string A R4 = length of string A R10 = address of string B R11 = byte offset of string B R12 = length of string B	R0 = destroyed R1 = zero if equal R1 = nonzero if not equal R2 = unchanged R3 = unchanged R4 = destroyed R10 = unchanged R11 = unchanged R12 = destroyed

8.43. S$TIMEA—Store Time into Message

This routine converts and stores the time value supplied in R2 and R3 into the buffer (Im byte) address (hh:mm:ss format).

The field pointed to by R10 and R11 must be of the format @@:@@:@@.

Call	Entry	Exit
SCALL S$TIMEA	R2 = time past midnight in milliseconds (upper) R3 = time past midnight in milliseconds (lower) R10,R11 = Message pointer (Im byte)	R0-R6 destroyed R10 = unchanged R11 = modified

8.44. S$UBTOA—Convert Binary to ASCII Unsigned

This routine converts an unsigned binary value into ASCII in the user's registers, right-justified and zero-filled.

Call	Entry	Exit
SCALL S$UBTOA	R3 = binary value	R0 = thousands digit/hundreds digit R1 = tens digit/ones digit R2 = sign/ten thousands digit R5 = original R3 binary value Destroys: R3,R4

Section 9
Memory Management

This section describes the following:

- DCP memory
- Disk cache
- Thresholds and throttling
- Resident and transient segments
- System tuning

9.1. Introduction

Memory is an important resource in any computer system. This is especially true when the computer is a communications processor. In a networking environment, memory is generally used in a relatively static manner. But for message traffic, memory is used in an extremely fragmented and volatile manner.

In this type of environment, memory must be available immediately in order to prevent messages from backing up throughout the network. It is quite normal to see tremendous fluctuations in message flow from minute to minute, or even from second to second. The apportioning of memory for various purposes (for example, the topping up of buffer pools and the detection of threshold conditions) is called "memory management."

9.2. DCP Memory

It is necessary for the DCP/OS to assign memory for several different purposes:

- Global CPA tables (for example, SCT, ICT)
- Bootstrap kernel

- DCP/OS resident segments and tables
- Program code/data and CPA structures
- Program structures expansion area
- Transient segments
- Dynamic segments
- Message buffers

In order to meet all of these needs, the DCP/OS initializes memory into the following pools at boot time:

- 128K-byte Buffer Pool

  A pool of 128K-byte buffers used for program transients, dynamic segments, dynamic work areas, and message buffers.

- 4K-byte Buffer Pool

  A pool of 4K-byte buffers used for hardware instrumentation buffers.

- 16K-byte Bank Pool

  A pool of 16K-byte buffers used for program resident segments and CPA structures.

  Note: *The 4K-byte buffer pool is very small and is therefore rarely used. From now on, when the term "buffer pool" is used, we are referring to the 128K-byte buffer pool.*

The initial buffer pool is fairly small, usually just several hundred buffers. You can tune the system and specify a larger pool. System tuning is discussed later in this section. Once the buffer pool has been established, all remaining memory is placed in the bank pool.

When programs are executed, there will typically be a larger demand on the buffer pool. The initial (small) pool can be quickly drained of all available memory. When this happens, the DCP/OS takes a spare bank from the bank pool, breaks it up into 128 buffers, and places them in the buffer pool. (The demand-driven thresholds are discussed later in this section.)

The bank pool is used primarily for resident program code and CPA structures. The bank pool also acts as a write-through mass-storage cache (see 9.3).

Note: *It is important to recognize that many CPA structures must be contiguous in memory and often exceed 4K bytes in length. The DCP/OS concept of 16K-byte banks does support contiguous structures up to 16K bytes in length.*

Internal memory management routines can combine up to three sequential banks. This provides for requests of up to 48K bytes of contiguous memory and would be, for example, sufficient memory to provide a procedure table of 1536 entries. Requests for contiguous memory are transparent to an executing program and typically occur as a consequence of a program-invoked service call (SVC) to expand a particular CPA structure (for example, CPA$XLA, expand link area).

9.3. Disk Cache

Every record that is less than 1K bytes in length is automatically cached in spare memory banks (if available). This cache can dramatically improve the performance of disk-intensive code by eliminating the need to access the disk on subsequent reads.

All updates to records residing in cache are also written out to disk, thereby avoiding any risk of missed updates on a system crash. The cache is comprised of one or more banks from the free bank pool. Each bank is used to hold up to sixteen 1K-byte "pages" of disk data. The oldest page is purged when no more free banks are available.

The mass storage cache is automatically enabled and expands up to a maximum of 28 banks (if available) unless the system has been tuned to override these default actions. As the system requires more banks, the DCP/OS automatically retrieves banks from the cache manager if the bank pool is empty. Again, this action can be governed by the system tuner.

9.4. Thresholds and Throttling

The buffer pool is very dynamic in nature; the demands fluctuate rapidly according to changes in the network load. As there are only a finite number of buffers available, memory management is chiefly concerned with "topping up" the buffer pool from the bank pool until there are no more banks available. When no more banks are available, the memory manager monitors the size of the buffer pool and alerts the system when low thresholds are crossed.

There are two threshold levels that indicate low buffer pool conditions: soft throttle and hard throttle.

1. Soft Throttle

Soft throttle is entered when the buffer pool has been drained to twenty percent of its original capacity. A soft throttle condition is not deemed a serious situation; rather, it is an indicator of an undesirable trend.

Note: *The soft throttle level can be set by the system tuner, either as a percentage of the buffer pool or to an absolute value (see 9.6).*

2. Hard Throttle

Hard throttle occurs when there are no buffers left in the buffer pool. This happens after the system has entered soft throttle and the demand for buffers has continued until the buffer pool is exhausted.

When hard throttle is triggered, the DCP/OS must wait for sufficient buffers to be returned to the pool in order to exit hard throttle. When sufficient buffers are returned to the buffer pool within the default time of one minute, then the system will return to soft throttle.

If the system is still in hard throttle after one minute (or whatever value has been set), the DCP/OS suspends all program dispatches, sets the PP hard throttle value, and releases the DCP/OS hard throttle pool to the buffer pool. Then the operating system begins to kill the running programs, starting with the lowest run priority and going up to the highest priority.

The reason for killing the programs is that if a program is incapable of responding to the throttle status change alerts, then it is probably incapable of recovering.

Note: *If the program is executing in batch mode, the runstream controls what action is to be taken when the program is killed. Typically, a dump will be taken and the program will be restarted.*

Programs executing on the DCP can either help or hinder the process of memory management. Any program that may be a potentially large user of the buffer pool should be registered with the DCP/OS for contingency notification.

A program that is so registered is informed about every change in the buffer pool status. In order to work well with the operating system, programs should attempt to curtail their demands on the buffer pool when soft throttle is entered, then gradually open up for full bore activity when notified that the system has reached normal status.

Figure 9-1 is an example buffer pool profile, showing a recoverable dip into soft throttle and a critical hard throttle situation.

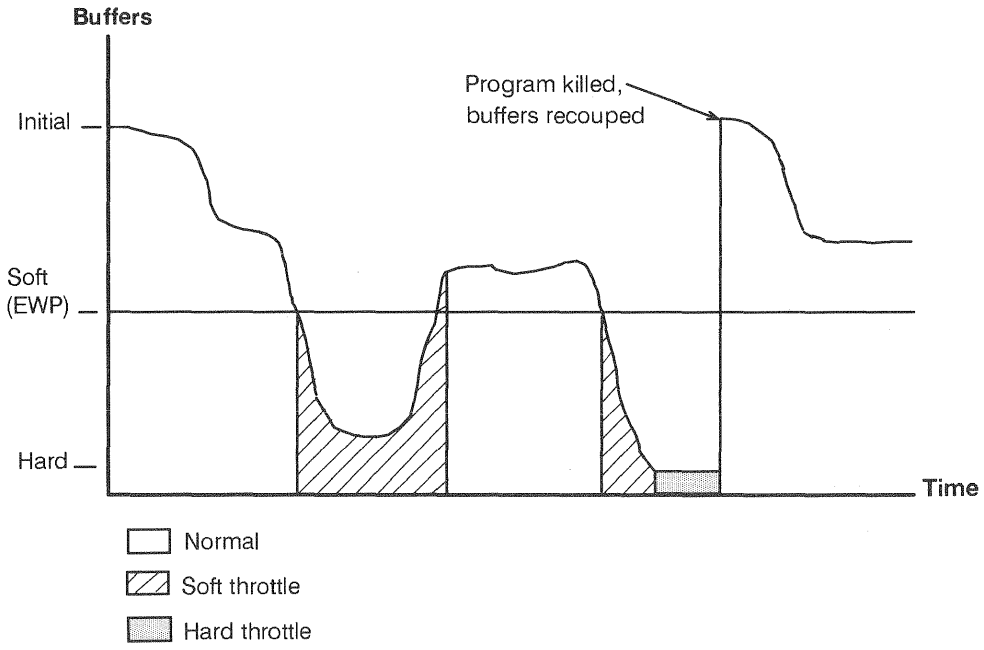


Figure 9-1. Buffer Pool Profile

. Resident and Transient Segments

Resident segments are segments of code or data that reside in memory at all times. Transient segments are segments of code or data that are pulled into memory only as required. If memory were not a critical resource, there would be no need for transient segments and all programs could be made resident. When programs are too large to be made completely resident, infrequently accessed code and read-only data segments should be marked as transient and pulled into memory only on demand.

There are a couple of disadvantages to marking segments as transient. Transient segments execute out of subsegmented space (buffers obtained from the buffer pool). Subsegmented (transient) code does not execute as efficiently as contiguous (resident) code. The relative efficiency varies by DCP model.

Secondly, if a transient segment is required more frequently than projected, there is a potential danger of excessive swapping. Swapping occurs when the buffer pool is depleted and the space occupied by all transient segments not currently in use is returned to the buffer pool. The next request for a released segment causes that segment to be pulled in from disk again.

There are three ways to mark a segment resident:

- Mark the segment resident in the code (SSTDEF)
- Mark the segment resident at build time
- Mark all segments resident at execution time (@PROG)

Note: *Remember that if transient segments are present, system performance may be affected. See the next section for more information on the transient segment sticking factor.*

. System Tuning

Within the DCP/OS, there are algorithms that adjust the memory pools according to demands placed upon the total system. These algorithms have been designed to enable smooth adjustments to the networking load for the majority of customer applications.

There are, however, cases where the mix of load, hardware, and protocols can benefit from some modifications of the system parameters within these algorithms. The modification of these parameters is called "system tuning"; the parameters are set by a program called @TUNER.

The @TUNER program can be run in demand mode or in batch mode. In demand mode, the user is presented with a series of menus showing the current value and maximum value for sets of system parameters. In both modes (demand and batch), the system parameters may be updated either in memory or, more permanently, on the disk file containing the DCP/OS.

Note: *Once a good working set of parameters has been determined, it may be useful to invoke the tuner program from the SYSJOB.STARTUP job.*

Although many parameters can be modified in tuning the system profile, most users will probably be most interested in tuning the minimum cache banks, the transient sticking factor, the initial buffer pool size, and the soft throttle level.

9.6.1. Minimum Cache Banks

By default, the minimum number of cache banks is zero. This means that all banks in use for the mass storage cache will be taken back from the cache manager if required for other purposes. If you run a disk-intensive program, you may want to ensure that a certain minimum number of banks are reserved for cache (in order to speed up file record access). To reserve cache banks, set the minimum cache banks value to nonzero.

9.6.2. Transient Sticking Factor

The transient sticking factor is the time that a transient segment is guaranteed to remain in memory after the use count goes to zero. The default value for the transient sticking factor is five seconds; the objective is to prevent excessive swapping of segments that are used regularly (but not frequently enough to be made resident).

3. Initial Buffer Pool Size

Because memory requirements are initially unknown, the initial buffer pool size is relatively small. The system automatically increases the buffer pool on demand, adjusting to the unique network load at any particular customer site. It may be determined that a larger initial buffer pool is needed. This is easily accommodated by setting the minimum buffer pool size to the desired value.

4. Soft Throttle Level

The soft throttle level is the system parameter that is probably most dependent upon unique customer configurations and network profiles. For this reason, the @TUNER program allows you to set the throttle level to an absolute number or to a percentage of the current buffer pool. It is generally more useful to set the throttle level to a percentage rather than to an absolute number.

By default, the soft throttle pool is set to 20 percent of the buffer pool. Internal monitoring discounts any buffers in use as dynamic segments in this percentage calculation (since dynamic segments are usually created for the life of the program and are not candidates for re-use). As more buffers are required, the buffer pool is increased and the 20 percent figure automatically governs the size of the soft throttle pool.

'. Commands for Use with Memory Management

There are several DCP/OS commands and utility programs that display information pertinent to memory management. The information focuses mostly on memory used by programs in one of the major memory classifications (buffers and banks).

1. RC Command

The RC display includes a size field which indicates the size of the banks (in K-bytes) currently allocated to the program. The size displayed does not include the memory used for transient segments, dynamic segments, or message buffers.

9.7.2. RD Command

The RD display appends data to the RC display, detailing the amount of memory used for resident segments, transient segments, and dynamic segments. This display does not show the buffers currently allocated by the program (message buffers and work areas).

9.7.3. T Command

The T command displays run-names, the programs that are active, and the size of the programs. As with the RC command, this size represents only the banks allocated to the program.

9.7.4. BIGB Program

The BIGB program provides a detailed breakdown of the CPA structures and banks in use by a program, for example, the number of queues and the amount of memory used for those queues.

9.7.5. SYS Program

The SYS program with the B option displays a bank map for the whole of memory. The run owning each bank is denoted on the map and a list of run-names and numbers is also shown. It should be noted that all banks broken into buffers, subsequent to the initial buffer pool being created, are shown as belonging to the DCP/OS, but the buffers are in the buffer pool and available to all users.

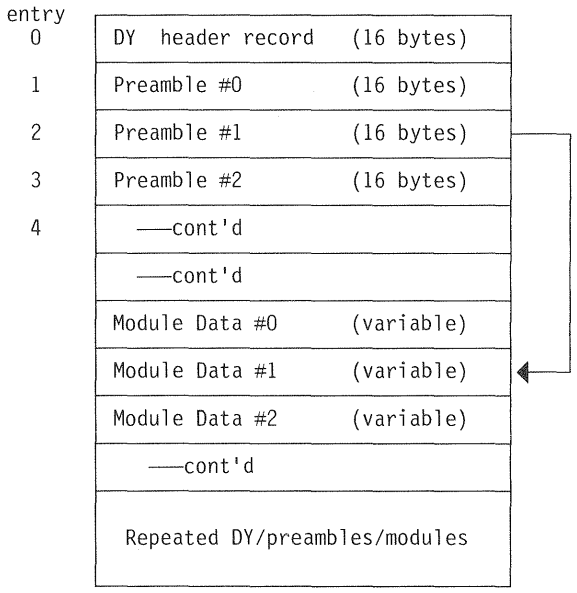
In higher levels of the SYS program, there is an additional display which is triggered by the M option. This display shows the amount of memory currently apportioned to buffers and to banks. It shows the amount available and in use; it also gives a breakdown of the utilization of the buffer and bank pools (for example, the number of buffers currently in use as transient segments and the number of buffers allocated in the system).

Appendix A
Data Structures

This appendix describes the DCP/OS data structures. The field definitions shown in the diagrams may change in future DCP/OS releases. Therefore, always use the field name, not the numeric values. This practice reduces complications on any future upgrades.

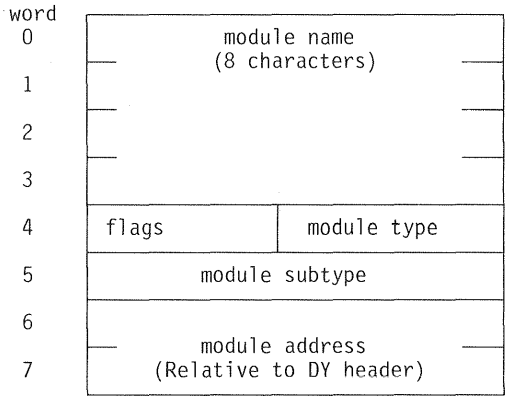
A.1. Module Library File Organization

The module library file (MLF) is the output from the DCPAPP processor and the input to the BUILD processor. See 1.2 for a description of how programs are built. The format of the MLF is as follows:



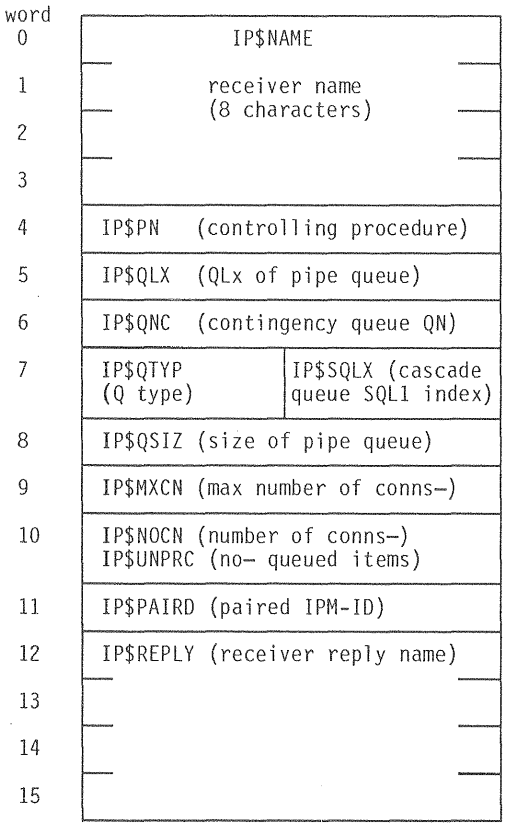
2. Preamble Format

The output of the MASM processor contains information about each module in entries called preambles (see 1.2.3). Each PRIVDEF, PROCDEF, IOPDEF, QUEUE, and SSTDEF statement results in MASM generating a preamble. All preambles have the following format:



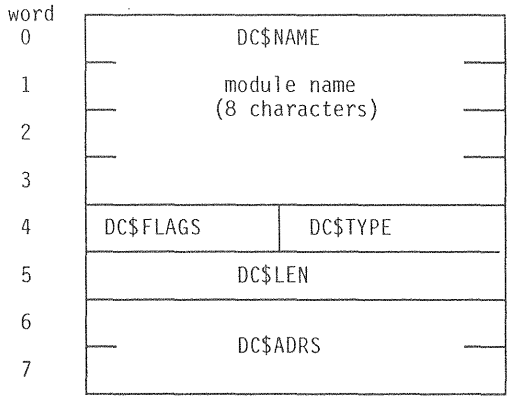
A.3. IPM Request Packet

Several IPM services either require or return lengthy parameters. For this reason, the parameters are passed in a packet described here. The packet cannot be in the first four SDRs, since they are architecturally cleared upon invoking a service. See the description of IPM services in 7.8.



.4. Dictionary Entry Format/Request Packet

The @BUILD utility program generates a dictionary containing the names of all CPA entities. A dictionary entry has the following format:



The flag word is defined as follows:

	Flag Word	Definition
bit 15	DC$FLDF	1 = entry is defined (condition)
bit 14	DC$FLINI	1 = initial procedure number (PN) (condition)
bit 12	DC$FLPRV	1 = privileged procedure (PRIVDEF)
bit 10	DC$FLMRK	1 = marked procedure (MARKDEF)

The meaning of the field DC$ADRS varies according to the type of dictionary entry (DC$TYPE), as follows:

Type Mnemonic	DC$TYPE	DC$ADRS (2nd word)
DC$TPPPP	Port processor (PP) program	PPPN
DC$TPPN	Procedure	PN
DC$TPQ	Queue	QN
DC$TPSEG	Segment	SSN

Note: The INFO file on the DCP/OS level 5R1 release tape contains an expanded version of this packet.

A.5. ABS Element Header Block

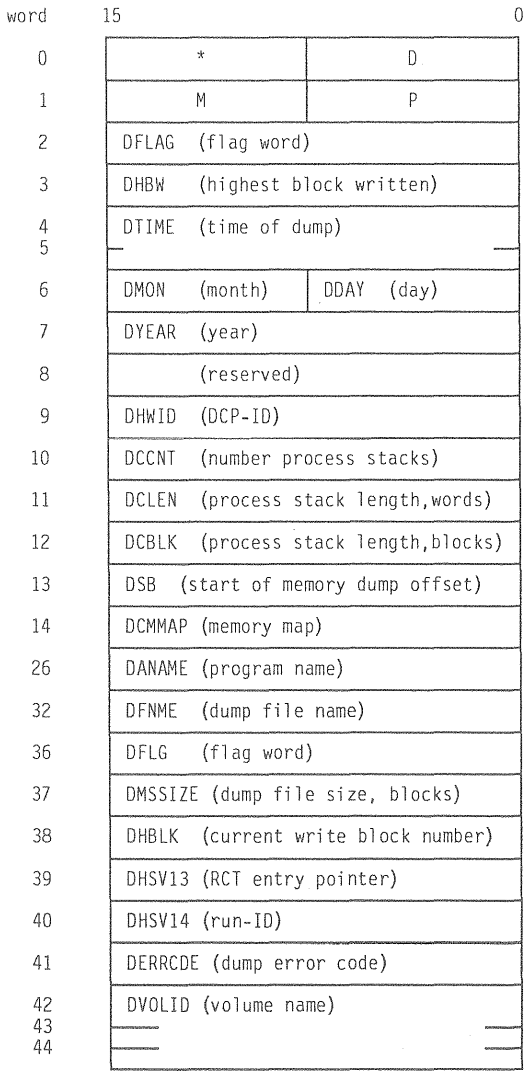
The first block of each ABS element is a header record that describes the executable system contained in the remainder of the element. See 1.3.3 for a description of absolute element. The format of this header block is as follows:

word	content (15 … 0)
0	* | L
1	D | *
2	
3	HD$LOCKS (# lock table entries)
4	HD$FLAG (flag word)
5	HD$FBNO (first free block no.)
6	HD$INIPN (initial PN)
7	HD$BANK (# memory banks required)
8	HD$BUFS (# 128-byte buffers required)
9	HD$NPCS (# stacks required)
10	HD$TRANS (# transient granules)
11	HD$RES (# resident granules)
12	HD$CPA (# CPA table granules)
13	Reserved
14	HD$QSL1 (System Q list 1 SSN/Offset)
15	HD$ALSSN (next free SSN)
16	HD$ALPN (next free PN)
17	HD$ALPPPN (next free PPPN)
18	HD$ALQN (next free QN)
19	HD$CTUND (count of undefined)
20	HD$CTDX (count of DIC entries)

21	HD$STBNO (SST block number)
22	HD$SSTN (maximum SST entries)
23	HD$PTN (maximum PT entries)
24	HD$PPPTN (maximum PPPT entries)
25	HD$QTN (maximum QT entries)
26-29	HD$FILE (4-word filename of load elt)
30-35	HD$ELT (6-word element name of load)
36-37	HD$TIME (time of build)
38	HD$DATE (date of build)
39	Reserved
40-47	HD$LEV (name from LEV command)
48	HD$ DPST (dictionary PST 40 entries)
88	HD$QHSG (Q header segment list)

A.6. Dump File Header Block

Dump files have a defined internal structure that is mapped by the header block described as follows:



.7. File Request Packet Format

word	content (15 … 0)	# words
0-3	reserved	4
4	FR$QN (queue # of return queue)	1
5	FR$USER (user parameter field)	1
6	FR$FNC (function code)	1
7	FR$CC (completion code) FR$OT (open type)	1
8	FR$FCB (file control block) FR$NAM (file name)	2 4
10	FR$BN (block number)	1
11	FR$WC (word count for I/O)	1
12	FR$BUF (word offset to data)	1
13	FR$RWC (reserved)	1
14 16	FR$VID (volume-ID) FR$DT (device type) FR$LMST (error - line mod status)	3 1 1
17	FR$DEVS (error - device status)	1
18	FR$LOC (location) FR$SENS (error - sense bytes)	4 1
19	FR$DVT (dev.char.table offset)	1
20	FR$PDT (Physical device table offset)	1
21	FR$HBW (highest block written)	1
22	FR$BLK (block number)	2
24	FR$ADATE (last access date)	1
25	FR$ATIME (last access time)	1
26	FR$XATDBO (ext.catalog attrib.offset)	1
27	FR$XATDBC (ext.cat.attrib.byte count)	1
28	FR$XATOPT (ext.catalog attrib option)	2
30	(reserved)	
31	FR$WCSAV (FR$WC save)	1

32	FR$QUAL (qualifier)	
38	FR$RDKEY	
41	FR$WRKEY	
44	FR$ENAME (element name)	
50	FR$ETYPE	FR$ESUBT
51	FR$ELBN (element start block)	
52	FR$ELEN (element length, blocks)	
53	FR$EYEAR	FR$EMON
54	FR$DAY	FR$EHOUR
55	FR$EMIN	FR$ESEC
56 : 63	FR$ELRSAV	Register save area

A.8. Table of Contents Block

The DCP/OS file structure supports subfiles called elements. For every eight elements in a file there is a table of contents (TOC) block. Each TOC block consists of a six-word header plus up to eight 14-word entries, each containing information about a single element. See 1.3 for a description of the DCP/OS file system. The following is the format of a TOC block:

word	content 15 8 7 0	
0	*	P
1	F	*
2	EF$HNXTWL (next write location)	
3	EF$NXTBN (next TOC block)	
4	EF$HECNT (TOC entries in this block)	
5	EF$HPRTBN (previous TOC blk #)	

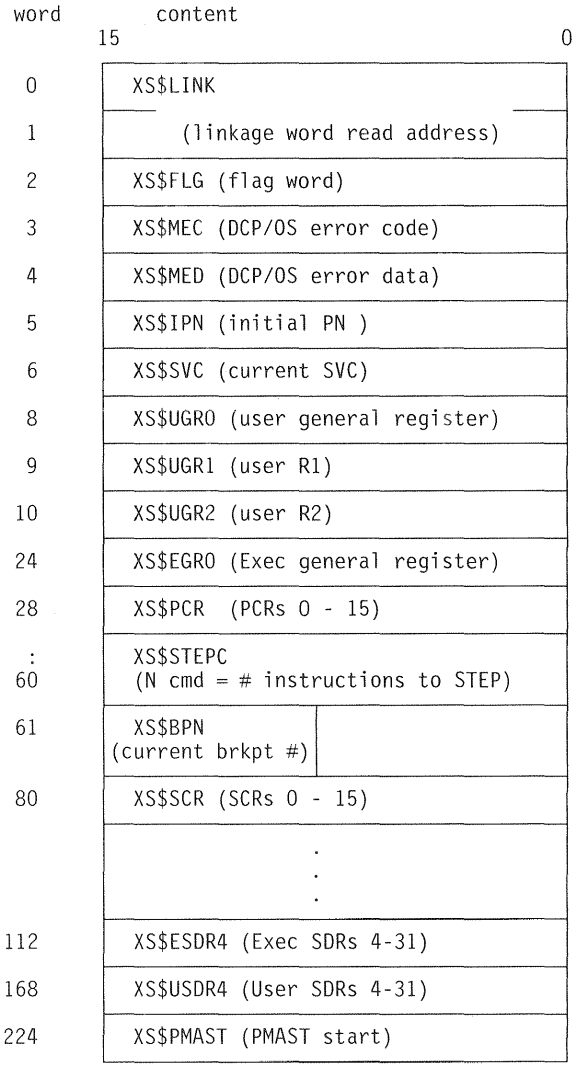
EF$HTOC + 0	EF$ENAM	
1	(12-character element name of first TOC entry)	
2		
3		
4		
5		
6	EF$EBN (start block, in file)	
7	EF$ELEN (element length, in blocks)	
8	EF$EFLGS (element flags)	
9	EF$ETYPE (type)	EF$ESUBT (subtype)
10	EF$EYEAR (year)	EF$EMON (month)
11	EF$EDAY (day)	
12	EF$EHOUR (hour)	EF$EMIN (minute)
13	EF$ESEC (sec.)	

9. MCT/Buffers Interface to File Manager

SD	content
n	MCT
n+1	data buffer 1
n+2	data buffer 2
n+3	FRP
n+4	reserved

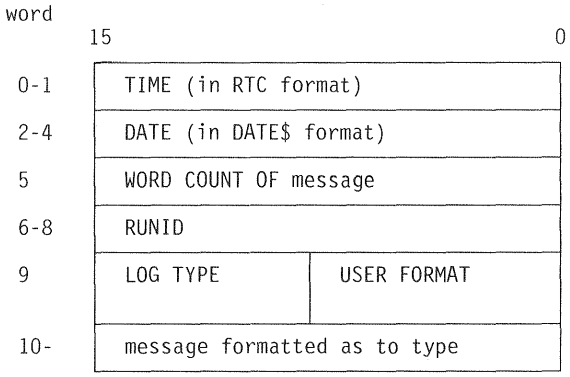
A.10. Contingency Packet Format

A contingency handler can employ dispatch services to retrieve the status of a process suspended due to a contingency and also to modify the content of that process. See 7.10 for a description of dispatch services. This is the interface to the services description:



11. Log File Entry

The LOG$ (log message) service logs the message in the system log file (SYS$*SYSLOG). The caller run-ID, the date, and the time are prefixed to the message. The format of a log file entry is as follows:



12. File Control Block

A file control block (FCB) for a file is a 24-word entry in the SFCB segment which maintains current file information while the file is in use by the file manager. The FCB (sometimes referred to as an FCB entry) is a temporary data structure which is initialized when the file manager accesses a file.

It is cleared when the file is no longer in use. Typically, an FCB is assigned when a file is opened. It remains active and in use for file I/O until the file is closed. The FCB is then released.

2.1. FCB Format

Field	Word Number	Description
FC$CBN	0	Current block number
FC$HBW	1	Highest block written

Field	Word Number	Description
FC$FLW	2	Flag word:
FC$RO	(bit 15)	- File is read-only
FC$DNS	(bit 14)	- 1600 BPI
FC$CAT	(bit 13)	- Update the catalog entry when the file is closed
FC$ACTV	(bit 12)	- A file management function is active
FC$TERM	(bit 11)	- Delete the FCB entry
FC$USED	(bit 10)	- FCB entry is in use
FC$SUSP	(bit 9)	- Suspension interface used
FC$CSN	(bits 8-0)	- Catalog sequence number (extent 0)
Note: Bit numbering has bit 15 as the most significant bit.		
spare	3	Not used
FC$OT	4 (bits 15-8)	Open type field
FC$ASIO	4 (bits 7-0)	Run number using the file
FC$FLW2	5	Flag word 2:
FC$REM	(bit 15)	- Remote access
FC$PIO	(bit 14)	- Physical I/O
FC$ASG	(bit 13)	- Device assigned
FC$ABORT	(bit 12)	- Abort in progress
FC$FASG	(bit 11)	- File assigned
FC$FASGX	(bit 10)	- File assigned exclusively
FC$INQ	(bit 9)	- Inhibit queued return
FC$TEMP	(bit 8)	- Temporary file
FC$PRIV	(bit 7)	- Private file
FC$SEQ	6	Cyclic sequence number
FC$PID	7	Process ID (for make up)
FC$CNXN	8	Remote connection number (2 words)
FC$CNXN2	9	
	-- OR--	
FC$EXT	8	Beginning of extent definition
FC$EXT1	FC$EXT	(old name)

Note: *Word 8 has double meaning depending on flag FC$REM.*

2.2. Extent Profile

An Extent is a section of a file on a physical device. The DCP/OS does not currently support multiple extents for a file. Each extent is defined with four words in the FCB starting with the first extent at FC$EXT. The word numbers given here are relative to the start of the extent definition.

Field	Word Number	Description
FC$HPB	0	Highest physical block (2 words)
FC$HLB	2	Highest logical block
FC$PDT	3	PDT entry offset

2.3. FCB Flags

In field FC$FLW:

FC$RO	read only if set
FC$DNS	1600 BPI if set
FC$CAT	write catalog entry at close
FC$ACTV	function active
FC$TERM	delete FCB entry
FC$USED	entry in use
FC$SUSP	suspension interface used

In word FC$FLW2:

FC$REM	remote file access
FC$PIO	physical I/O
FC$ASG	device assigned
FC$ABORT	abort in progress
FC$FASG	file assigned
FC$FASGX	file assigned exclusively
FC$INQ	inhibit queued return
FC$TEMP	temporary file
FC$PRIV	private file

A.12.4. File Manager Functions

The following tables clarify file manager functions. Refer to 7.6 for more information.

FR$STAT (file is not open)

The following FRP fields are updated:

FR$ADATE	Date of last file access
FR$ATIME	Time of last access (to nearest minute)
FR$VID	Qualifier of file

Selected FCB fields are written into a copy of the FCB (first 12 words) located at FR$FCBDEF. (See Appendix A for the FCB format). These fields are:

FC$CSN	Catalog sequence number
FC$HBW	High block written
FC$DBN	Current block number (set to 0)
FC$RO	Read only file flag
FC$TEMP	Temporary file flag
FC$FASG	File assigned
FC$FASGX	File assigned exclusively
FC$EXT1	Extent definition

FR$STATO (File is open)

The first 24 bytes of the FCB for the open file are copied into the FRP at location FR$FCBDEF. The rest of the FCB is not used by the file manager right now. See Appendix A for a description of the FCB format.

Appendix B
Segment Names of Available User System Structures and Code

This appendix lists the various architectural and operating system segments that can be made visible to programs at execution time. These segments may be referenced by name in the program, and the operating system satisfies such references at program load time. The executing program can then access such segments either explicitly by loading them (LSEG) or implicitly by including a system segment in the initial visibility of a user procedure.

The following list of segments may be accessed by user programs.

Data Structure/Code	Segment Name	Field Definitions
Common utility subroutines	SAGSUB	AASUBENT (entry points)
System control table (SCT)	SCT	AACPADEF
Interface control block (ICB)	SICB	AACPADEF
Physical device table (PDT)	SPDT	AAFILDEF
System information table	SYSINF	AARUNDEF

Appendix C
Definition Elements

This appendix lists the definition elements that are used to define all system-wide definitions, including the assembler instructions.

The elements are categorized according to use and context. The primary definition element is AAWRENCH, which contains the central processor (CP) instruction set, extended instruction definitions, and module definition MASM procedures.

Element	Context	Definitions
AABOOTDEF	Bootstrap	
AACPADEF	CPA definitions	SC$,IC$,QU$,SD$,PT$,QT$,SS$
AADEVDEF	Device definitions	DV$
AAERRDEF	DCP/OS errors	EC$,ER$
AAEXECDEF	Exec definitions	XB$,XI$,XL$,XS$,XV$
AAEXTPROC	Extended MASM procs	Ancillary MASM procedures
AAFILDEF	File manager	FR$,FC$,CT$,PD$
AAIPMDEF	IPM definitions	IP$,IPC$
AAPPEQU	PP definitions	General PP definitions
AAPPROC	PP op-codes	PP MASM procedures
AARUNDEF	Run manager	MH$,IF$,RC$,AV$,AK$,SI$, CN$,DC$,SE$,LOG$
AASERVDEF	Service codes	CPA$,PP$,PC$,IS$,run$,LM$
AASLDEF	Loader definitions	
AASTRPRC	Structured MASM procs	
AASUBENT	DCP/OS subroutines	S$,X$
AAWRENCH	CP op-codes	CP MASM procedures

Appendix D
Service Function Codes

This appendix lists the service function codes in the order of hexadecimal value and briefly describes each service. The hex value of each SVC function may be useful to assembler programmers when debugging code.

D.1. Function Codes - File Services (FILE$)

```
FILE$        . FILE MANAGER

FUNCTION CODES

  Service    Hex Value     Description

  FR$CAT     (X'8000')     CATALOG A FILE
  FR$STAT    (X'8001')     STATUS OF FILE
  FR$OPN     (X'8002')     OPEN A FILE
  FR$OPNI    (X'8003')     OPEN A FILE IMMEDIATE
  FR$POS     (X'8004')     POSITION A FILE
             (X'8005')     Reserved
  FR$RD      (X'8006')     READ
  FR$WRT     (X'8008')     WRITE
  FR$WCT     (X'8009')     WRITE CATALOG
  FR$CLS     (X'800A')     CLOSE A FILE
  FR$CLSI    (X'800B')     CLOSE IMMEDIATE
  FR$DEL     (X'800C')     DELETE A FILE
  FR$PREP    (X'800D')     FORMAT A DISK
  FR$QFE     (X'800F')     PUT Q*F.ELT STING INTO FRP
  FR$ENXWL   (X'8011')     ELEMENT - GET NEXT WRITE LOCATION
  FR$EINS    (X'8012')     INSERT AN ELEMENT INTO THE TOC
  FR$ERS     (X'8013')     ERASE A FILE
  FR$ASG     (X'8014')     ASSIGN A DEVICE
  FR$FREE    (X'8015')     FREE A DEVICE
  FR$ESRCH   (X'8016')     SEARCH THE TOC
```

FR$UP	(X'8017')	UP A DEVICE
FR$EDEL	(X'8018')	DELETE AN ELEMENT FROM THE TOC
FR$DOWN	(X'8019')	DOWN A DEVICE
FR$EUPWL	(X'801A')	UPDATE NEXT WRITE LOCATION
FR$STATO	(X'801B')	STATUS OF OPEN FILE
FR$RESET	(X'801C')	RESET HIGHEST BLOCK WRITTEN
	(X'801D')	Reserved
FR$RENF	(X'801E')	RENAME A FILE
	(X'801F')	Reserved
	(X'8020')	Reserved
FR$FDN	(X'8021')	DOWN A FILE
FR$FUP	(X'8022')	UP A FILE
FR$SFRO	(X'8024')	SET FILE TO READ-ONLY
FR$CFRO	(X'8025')	CLEAR FILES READ-ONLY FLAG

2. Function Codes - CPA Services (CPA$)

Service	Hex Value	Description
CPA$GSSN	(X'8101')	GET SSN FROM PST ENTRY
CPA$GPN	(X'8102')	GET PN FROM GPL ENTRY
CPA$GQN	(X'8103')	GET QN FROM QL ENTRY
CPA$AQL	(X'8104')	ADD QUEUE TO QL (PN)
CPA$RQL	(X'8105')	REMOVE QUEUE FROM QL (PN)
CPA$APST	(X'8106')	ADD SSN TO PST
CPA$RPST	(X'8107')	REMOVE SSN FROM PST ENTRY
CPA$CVIS	(X'8108')	CHECK SDR VISIBILITY
CPA$CQL	(X'8109')	CREATE PN QL
CPA$CPPQL	(X'810A')	CREATE PP QL
CPA$DPPQL	(X'810B')	DELETE PP QL
CPA$APPQL	(X'810C')	ADD QUEUE TO PP QL
CPA$RPPQL	(X'810D')	REMOVE QUEUE FROM PP QL
CPA$CSEG	(X'810E')	CREATE A SEGMENT
CPA$DSEG	(X'810F')	DELETE DYNAMIC SYSTEM SEGMENT
CPA$CQUE	(X'8110')	CREATE A QUEUE
CPA$DQUE	(X'8111')	DELETE DYNAMIC QUEUE
CPA$QSTAT	(X'8112')	GET QUEUE STATUS
CPA$ESSN	(X'8113')	EXTEND DYNAMIC SYSTEM SEGMENT
	(X'8114')	Reserved
	(X'8115')	Reserved
CPA$CLONQ	(X'8116')	CLONE A QUEUE

CPA$HOLDS	(X'8117')	HOLD SEGMENT IN MEMORY
CPA$FREES	(X'8118')	FREE SEGMENT HOLD
CPA$FFPST	(X'8119')	FIND FREE PST ENTRY
CPA$FFQL	(X'811A')	FIND FREE QL ENTRY
CPA$QMOD	(X'811B')	MODIFY QUEUE
CPA$GETSD	(X'811C')	GET SDR CONTENTS
CPA$CKSD	(X'811D')	CHECK SEGMENT DESCRIPTOR
CPA$QMDE	(X'811E')	MODIFY QUEUE MODE TO 0
CPA$CLA	(X'811F')	CREATE LINK AREA
CPA$XPST	(X'8120')	EXTEND PST
CPA$XLA	(X'8121')	EXTEND LINK AREA
CPA$XQL	(X'8122')	EXTEND QUEUE LIST
	(X'8123')	Reserved
CPA$QL1X	(X'8124')	GET SYSTEM QUEUE LIST 1 INDEX
CPA$AGPL	(X'8125')	ADD PN TO GPL
CPA$FFLA	(X'8126')	FIND FREE LA ENTRY
CPA$PINFO	(X'8127')	GET PN INFORMATION
CPA$ASPT	(X'8128')	ADD SEGMENT TO PROCEDURE TABLE ENTRY
CPA$RSPT	(X'8129')	REMOVE SEGMENT FROM PT ENTRY
CPA$LPST	(X'812A')	MAKE PST A LOADABLE SEGMENT
CPA$CAET	(X'812B')	CREATE ALTERNATE ENVIRONMENT TABLE
CPA$DAET	(X'812C')	DELETE ALTERNATE ENVIRONMENT TABLE
CPA$CAE	(X'812D')	CREATE ALTERNATE ENVIRONMENT ENTRY
CPA$DAE	(X'812E')	DELETE ALTERNATE ENVIRONMENT ENTRY
CPA$AAQL	(X'812F')	ADD QUEUE TO ALTERNATE QUEUE LIST (PP)
CPA$RAQL	(X'8130')	REMOVE QUEUE FROM ALTERNATE QUEUE LIST
CPA$RSSN	(X'8131')	RESERVE CONSECUTIVE SSN ENTRIES
CPA$USSN	(X'8132')	UNRESERVE CONSECUTIVE SSN ENTRIES
	(X'8133')	Reserved
	(X'8134')	Reserved
CPA$FFQLR	(X'8135')	FIND FREE QL ENTRY WITHIN RANGE
CPA$FFLAR	(X'8136')	FIND FREE LA ENTRY WITHIN RANGE
CPA$FFPSTR	(X'8137')	FIND FREE PST ENTRY WITHIN RANGE
CPA$FFGPL	(X'8138')	FIND FREE GPL ENTRY
CPA$FFGPLR	(X'8139')	FIND FREE GPL ENTRY WITHIN RANGE
CPA$RGPL	(X'813A')	REMOVE GPL ENTRY
CPA$ASVC	(X'813B')	ADD EXTENDED SVC PROCEDURE
CPA$RSVC	(X'813C')	REMOVE EXTENDED SVC PROCEDURE
CPA$GSVC	(X'813D')	GET SVC INFORMATION
CPA$DASVC	(X'813E')	DISABLE AUTOMATION ON NEXT SVC
CPA$MPPQL	(X'813F')	MODIFY ACCESS RIGHTS FOR PPQL
CPA$QSMD	(X'8141')	SET QUEUE MODE

.3. Function Codes - Run Control Services

Service	Hex Value	Description
READ$	(X'8201')	READ IMAGE
PRINT$	(X'8202')	SEND MESSAGE TO CURRENT OUTPUT STREAM
EXIT$	(X'8203')	TERMINATE PROGRAM
RINFO$	(X'8204')	GET RUN INFORMATION
READT$	(X'8205')	READ IMAGE (TRANSPARENT)
FACMSG$	(X'8206')	EXPAND FILE MANAGER ERROR CODE
ERR$	(X'8207')	TERMINATE PROGRAM WITH ERROR CODE
READW$	(X'8208')	READ IMAGE (FROM WORKSTATION)
TREAD$	(X'8209')	TYPE AND READ IMAGE
DATE$	(X'820A')	GET CURRENT SYSTEM DATE
INFO$	(X'820B')	GET RUN INFORMATION
CSF$	(X'820C')	ISSUE CONTROL STATEMENT
PRTCN$	(X'820D')	SET WORKSTATION PRINT CONTROL
LINFO$	(X'820E')	GET LOAD INFORMATION
SETC$	(X'820F')	SET RUN CONTROL WORD
GETC$	(X'8210')	GET RUN CONTROL WORD
COM$	(X'8211')	CONSOLE OUTPUT MESSAGE
COMW$	(X'8212')	SOLICIT CONSOLE RESPONSE
RQUAL$	(X'8213')	READ PROJECT-ID (DEFAULT QUALIFIER)
RAQUAL$	(X'8214')	READ ASSUMED QUALIFIER
RDQUAL$	(X'8215')	READ DUMP FILE QUALIFIER
PREAD$	(X'8216')	PRIORITY READ
LOG$	(X'8217')	LOG A MESSAGE INTO LOG FILE
RRCT$	(X'8218')	GET READ ACCESS TO RCT ENTRY

.4. Function Codes - Process Control (PC$)

Service	Hex Value	Description
PC$SKAT	(X'8301')	SCHEDULE PROCEDURE AT ABSOLUTE TIME
PC$SKDT	(X'8302')	SCHEDULE PROCEDURE AFTER DELTA TIME
PC$PAUSE	(X'8303')	SUSPEND PROCESS
PC$WFE	(X'8304')	SUSPEND, WAIT FOR EVENT
PC$CWFE	(X'8305')	CLEAR, WAIT FOR EVENT
PC$SKGDT	(X'8306')	SCHEDULE PROCEDURE AFTER DELTA TIME BY GPL(X)
PC$SKGAT	(X'8307')	SCHEDULE PROCEDURE AT ABSOLUTE TIME BY GPL(X)
PC$SCSD	(X'8308')	SETUP COMMON SDS

PC$WHO	(X'8309')	DETERMINE CURRENT PROCESS-ID
PC$SLEEP	(X'830A')	SUSPEND INDEFINITELY
PC$WAKE	(X'830B')	WAKE A SLEEPING PROCESS
PC$CREG	(X'830C')	REGISTER A CONTINGENCY HANDLER
PC$CSTAT	(X'830D')	GET CONTINGENCY STATUS
PC$CKILL	(X'830E')	KILL CONTINGENCY - SUSPEND PROCESS
PC$CMOD	(X'830F')	MODIFY CONTINGENCY - SUSPEND PROCESS
PC$CRES	(X'8310')	RESTART CONTINGENCY - SUSPEND PROCESS
PC$CBUG	(X'8311')	DEBUG CONTINGENCY - SUSPEND PROCESS
PC$CRD	(X'8312')	READ VIRTUAL SPACE FROM SUSPEND PROCESS
PC$CWT	(X'8313')	WRITE VIRTUAL SPACE IN SUSPEND PROCESS
PC$CRS	(X'8314')	READ CONTINGENCY - SUSPEND PMAST
PC$POPS	(X'8315')	POP SRTN ENTRIES OFF THE STACK
PC$MRET	(X'8316')	MODIFY RETURN ADDRESS
PC$SREG	(X'8317')	REGISTER FOR STATUS CHANGES
PC$IREG	(X'8318')	REGISTER II/C CONTINGENCY PROCEDURE
PC$RCSD	(X'8319')	REMOVE COMMON SD'S
PC$CSTK	(X'831A')	RETURN INFORMATION FROM STACK
PC$PKILL	(X'831B')	REGISTER FOR PROGRAM KILL NOTIFICATION

D.5. Function Codes - PP Services (PP$)

Service	Hex Value	Description
PP$CNFG	(X'8401')	CONFIGURE PP
PP$START	(X'8402')	START PP
PP$STOP	(X'8403')	STOP PP
PP$ELIM	(X'8404')	ELIMINATE PP
PP$ASG	(X'8405')	ASSIGN PP
PP$FREE	(X'8406')	FREE PP
PP$CSTART	(X'8407')	CONFIGURE AND START PP
PP$STOPAS	(X'8408')	STOP ALL PPS ASSIGNED TO A PROGRAM
PP$STATUS	(X'8409')	GET PP STATUS
PP$GETSI	(X'840A')	GET PP STATUS AND STATE ITEM
PP$PUTLM	(X'840B')	STORE LM MICROCODE ID IN ICT
PP$GETLM	(X'840C')	GET LM MICROCODE ID FROM ICT
PP$THROT	(X'840D')	ESTABLISH PP THROTTLE
PP$FREEAS	(X'840E')	FREE A PP ASSIGNED TO A PROGRAM

.6. Function Codes - Line Module Loader (LM$)

Service	Hex Value	Description
LM$LOAD	(X'8601')	LOAD LINE MODULE
LM$GLMID	(X'8602')	GET LINE MODULE STATUS
LM$XLOAD	(X'8603')	EXTENDED LINE MODULE LOAD

.7. Function Codes - Dictionary Services (DIC$)

Service	Hex Value	Description
DIC$FINM	(X'8901')	DICTIONARY SEARCH BY NAME
DIC$FINMC	(X'8902')	SEARCH BY NAME FOR SPECIFIED RUN
DIC$FENT	(X'8903')	FIND BY FLAGS/TYPE FROM ENTRY
DIC$GENT	(X'8904')	RETURN INFORMATION GIVEN ENTRY NUMBER
DIC$FIADC	(X8906')	SEARCH BY ADDRESS FOR SPECIFIED RUN
DIC$FIAD	(X'8905')	DICTIONARY SEARCH BY ADDRESS

.8. Function Codes - IPM Services (IPM$)

Service	Hex Value	Description
IPM$RCV	(X'8A01')	ESTABLISH AN IPM RECEIVER
IPM$CONN	(X'8A02')	CONNECT TO IPM RECEIVER
IPM$DISC	(X'8A03')	DISCONNECT AN IPM CONNECTION
IPM$FREE	(X'8A04')	FREE AN IPM CONNECTION
IPM$CLOS	(X'8A05')	CLOSE AN IPM CONNECTION
IPM$STAT	(X'8A06')	GET AN IPM CONNECTION STATUS
IPM$CHK	(X'8A07')	CHECK STATUS OF IPM RECEIVER

.9. Function Codes - SDF Record Services (E$)

Service	Hex Value	Description
E$READ	(X'8B05')	READ RECORD

E$SDFOI	(X'8B0A')	INITIALIZE SDF OUTPUT
E$SDFO	(X'8B0B')	WRITE SDF RECORD
E$SDFOC	(X'8B0C')	CLOSE SDF OUTPUT

D.10. Function Codes - Instrumentation Services (I$)

Service	Hex Value	Description
I$SETUP	(X'8C01')	SETUP INSTRUMENTATION
I$RUN	(X'8C02')	SET RUN ICW
I$PN	(X'8C03')	SET PROCEDURE ICW
I$PP	(X'8C04')	SET PP ICW
I$PPBUF	(X'8C05')	ESTABLISH PP INSTRUMENTATION BUFFER
I$FLUSH	(X'8C06')	FLUSH ALL INSTRUMENTATION STRUCTURES
I$PPSETUP	(X'8C07')	ESTABLISH PP INSTRUMENTATION ONLY
I$RUNBUF	(X'8C08')	ESTABLISH RUN INSTRUMENTATION BUFFER
I$STARUN	(X'8C09')	RETURN RUN ICW
I$STAPN	(X'8C0A')	RETURN PROCEDURE ICW
I$STAPP	(X'8C0B')	RETURN PP ICW
I$SQL5G	(X'8C0C')	GET ACCESS TO SQL5
I$SQL5R	(X'8C0D')	REMOVE SQL5 ACCESS

Appendix E
Programming Examples

This appendix presents the following programming examples:

- CP Program
- Assembly (on OS 1100)
- Collection (on OS 1100)
- Build (on DCP/OS)

E.1. CP Program Example

```
          $INCLUDE  'AAWRENCH'              .
          WRENCH                            .
          $INCLUDE  'AARUNDEF'              .
          $INCLUDE  'AASUBENT'              .
          $INCLUDE  'AASERVDEF'             .
$(0)                                        .
.
.
.
.
. ***********************************
. *                                 *
. *          Example Program        *
. *                                 *
. ***********************************
.
. ARCHITECTURAL TABLES
.
          PSTDEF    'SDUMMY'          . example PST

DEFTIM    EQU       30                . example equate
MINTIM    EQU       5                 . ------"-------
.
. Procedure: PEXAM
.
. Function:  Read a couple of inputs and display
.            a message to the workstation.
.
. Entry:     Initial PN of program.
.
. Exit:      EXIT$
.
.
```

```
. Read the INFOR$ packet
.
PEX01
         LOADC     R1,AV$MCT          . point to mct
         SVC       READ$              . go read it
.
. Release the INFOR$ packet
.
         LOADC     R10,AV$MCT         . get INFOR$
         SCALL     S$REL              . go deallocate it
.
. Build up a prompt message
.

         LOADC     R2,256             . message size
         SCALL     S$GETMA            . get a message buffer
         LOADC     R11,MH$FLTH*2      . get msg offset
         STORE     R11,MH$DBO,R10     . store msg offset
         LOADC     R0,MSGEXA          . point to msg
         LOADC     R1,0               . (byte offset 0)
         LOADC     R2,MSGEXL          . msg length
         STORE     R2,MH$DBC,R10      . store msg length
         MBMI      R0,R10             . move msg to buffer
         LOADC     R0,MH$MCO          . get msg type
         STORE     R0,MH$MTYP,R10     . store msg type
.
. Send message (prompt) to workstation
.

         LOADC     R1,AV$AMCT         . point to MCT
         SVC       PRINT$             . go display it!
.
. Wait for a message from the workstation
.

         LOADC     R1,AV$MCT          . point to i/p mct
         SVC       READ$              . go read a message
.
. Now let's pull the plug!
.

         SVC       EXIT$              . bye bye
.
. Data area (under location counter 1)
.
$(1)                                  . LC 1
.

        $RES       SDR1               . force VA to X'800'
.
MSGEXA    'Just hit transmit!'
MSGEXL    EQU       ($-MSGEXA)*2
.
. ARCHITECTURAL TABLES
.
.
SEXAM     SSTDEF    'AA$ZRES' CP      SDR0 0            . code (location centers 0)
SEXAMD    SSTDEF    'AA$ZTRN' READ    SDR1 1            . definition (location center 1)
PEXAM     INITDEF   'PEX01' 'SEXAM' 'SEXAMD' 0 'SAGSUB'. program entry point

$END
```

E.2. Assembly Example (on OS 1100)

```
@USE      ASM$PF,DCPOSEQU                    . point to system definitions
@ASG,A    ASM$PF.                            . assign for MASM
@HDG      EXAM                               . set title
@MASM,SE  TP$.EXAM                           . create temporary relocatable
```

E.3. Collection Example (on OS 1100)

```
@MAP,S   ,TP$.EXAM                           . create temporary absolute
.

          NOT       SYS$*RLIB$.              .
          LIB       DCPOSABS                 .
          MINGAP    2048                     .
          MINSIZ    0                        .
.
          IBANK     EXAM,0
          IN        EXAM($63,62,61,60)
          $ALLBUT,63,62,61,60
          IN        EXAM
.
          IBANK     END,0
          IN        MLF-END($63,62,61,60)
.
          END

@DCP1100ABS.DCPAPP,M TPF$.EXAM,QUAL*FILE.EXAM      . create MLF element
```

E.4. Build Example (on DCP/OS)

The @BUILD utility program is executed under DCP/OS and **not** on the OS 1100 host. The MLF element that was created as part of the collection process must first be copied from the OS 1100 system to the DCP. See the DCP Series DCP/OS Operations Reference Manual (UP-11541.2) for use of DCPFT.

```
@CAT QUAL*FILE.
@COPY,O CH0:QUAL*FILE.EXAM,QUAL*FILE.EXAM          . copy MLF from channel CH0
@BUILD,L  QUAL*FILE.EXAM                           . build DCP absolute
LEV       Tuesday sys                              . user identifier
IN  QUAL*FILE.EXAM                                 . include MLF
END                                                . end of builder
```

Appendix F
Multiple CP Processing (DCP/35 and 55)

DCP/OS supports a multiprogramming environment that allows the concurrent (interleaved) execution, regardless of the number of central processors (CPs), of DCP/OS multiple tasks, and each program's multiple tasks by sharing CP use through switching. DCP/OS also supports a multiple CP processing environment that allows the simultaneous execution of multiple tasks by emloying two or more CPs that access common main storage.

When you design and code in a multiple CP processing environment, consider the following:

- STRUCTURES that are accessible to multiple tasks.
- UPDATE and multicell read OPERATIONS requiring indivisible (atomic) execution.
- SERIAL and PARALLEL execution of code.
- TIMESLICE attribute (through monitor events) of executing task at the point of the locking bit.
- SUSPENSION potential of executing task for duration of locked bit.
- QUEUE and CALL inerfaces to code.
- Operational NARROWING of CPA access lists from many logical items to fewer physical items.

The multiple CP processing environment allows better performance than a single processing environment. No attempt is made to recover a failure in DCP/OS/microde/hardware on a given CP. When one CP goes down, logically the entire DCP/OS has erred, and the other CPs stop in time. To better use the available CP power, DCP/OS is designed to:

- Maximize parallelism within the system
- Minimize the control needed to support the parallelism with the controlled access and serial processing

Controlled access and serial execution are provided within the system by the following mechanisms under the stated conditions:

- Microcode supported atomic operations (such as RIN, RD, SD, TS) can be used where they are convenient. Use of these atomic operations reduces use of the other more expensive mechanisms.

.1. GQITEM/ARMQ (include PC$WFE)

1.1. Typical User

The typical users of GQITEM/ARMQ (include PC$WFE) are:

- Various DCP/OS services
- PASCAL run-time services (uses PC$WFE)

1.2. Scenario

The scenairo for use of GQITEM/ARMQ (include PC$WFE) are:

- When the queue threshold is crossed, a task is dispatched; another task is not dispatched until the queue is rearmed and another threshold crossing occurs. This mechanism provides serial processing on a queue basis.
- Cascaded queueing (multiple work queues cascading to a single event queue to cause a single dispatch) is an extension of this basic mechanism to provide serial processing across a number of queues.
- Wait-for-event (using PC$WFE) allows the calling task to suspend itself until the specified queue threshold is crossed. The calling task is then redispatched with its saved environment. The queue can be either normal or cascaded.
- Otherwise, multiple queues can cause multiple dispatches and parallel processing.

1.3. Cost

The following describes the cost of the GQITEM/ARMQ service:

- Requires at least one queue.
- Task needs to be redispatched upon each queue threshold crossing.

.1.4. Benefit

The benefit of using the GQITEM/ARMQ service is that the task can serially work on queue items without causing a task (re)dispatch for each queue item. The task is (re)dispatched only when the threshhold has been crossed.

.2. LOK

.2.1. Typical User

The typical users of LOK are:

- Internal DCP/OS handlers.
- DCP/OS SVC services user interface.

.2.2. Scenario

The scenairo for use of GQITEM/ARMQ (include PC$WFE) are:

- Use of LOK can result in the task operating system (OS) or user, respectively, being placed automatically in a suspended state if the LOK is already being used. When the LOK is available for the task, the task is reactivated automatically and given the LOK.
- If possible, avoid setting other lock bits when the task is to be suspended.

.2.3. Cost

The following describes the cost of the LOK service:

- The task can be placed in a suspended state from which it needs to be reactivated by DCP/OS.
- Requires a lock table entry.

.2.4. Benefit

The following describes the benefit of the LOK service:

- Task is only reactivated when it can have the LOK.
- Task does not burn cycles waiting for the lock to clear.

.3. TS/PAUSE

.3.1. Typical User

The typical user of DCP/OS SVC services interface.

.3.2. Scenario

The scenario for use of TS/PAUSE is:

- This can result in the user task being placed in a pause state if the TS lock bit is already set. The task comes out of the pause state and the task must retry the TS lock bit.
- If possible, avoid setting other lock bits when the task is to be paused.
- If the task is running in the process mode register and SDR sets, the pause is performed through SVC$ PC$PAUSE.

.3.3. Cost

The cost of the TS/PAUSE service is that the task can be repeatedly placed in a pause state from which it must be reactivated by DCP/OS each time the pause expires.

.3.4. Benefit

The following describes the benefits of using the TS/PAUSE service:

- Requires only one lock bit.
- Task does not burn cycles waiting for the lock to clear.

.4. TS/SPIN

.4.1. Typical User

The typical user of TS/SPIN is internal DCP/OS services.

F.4.2. Scenario

The scenario for use of TS/SPIN is the task continues testing the TS lock bit until the task sets the TS lock bit.

F.4.3. Cost

The cost of the TS/SPIN service is that the task burns cycles waiting for the lock to clear.

F.4.4. Benefit

The benefits of the TS/SPIN service are;

- Requires only one lock bit.
- Does not require the overhead to save and reactivate the locking task.

F.5. TS/SPIN/PAUSE

F.5.1. Typical User

The typical user of the TS/SPIN/PAUSE service is the DCP/OS SVC services interface.

F.5.2. Scenario

The scenario for use of TS/SPIN/PAUSE is:

- The task continues testing the TS lock bit until either the task sets the TS lock bit or the tunable iteration count expires with the TS lock bit already set. This expiration places the user task in a pause state. The task comes out of the pause state and the task must retry the TS lock bit.
- If possible, avoid setting other lock bits when the task is to be paused.
- If the task is running in the process mode register and SDR sets, the pause is performed through SVC PC$PAUSE.

.5.3. Cost

The cost of the TS/SPIN/PAUSE service is:

- Task burns cycles waiting for the lock to clear.
- The task can be (repeatedly) placed in a pause state from which it must be reactivated.

.5.4. Benefit

The benefits of the TS/SPIN/PAUSE service are:

- Requires only lock bit.
- The task does not continually burn cycles waiting for the lock to clear.
- The task does not require the overhead of frequent saving and reactivating the locking task.

  You must consistently follow nested locking bit order for both setting and clearing. You must clear in the exact reverse order of the setting. The lock bits must maintain the same order position relative to each other. The following is an example.

```
        :                 :                 :
Set–lock-bit 1    Set-lock-bit 1    Set-lock-bit 2
Set-lock-bit 2    Set-lock-bit 3    Set-lock-bit 3
Set-lock-bit 3          :                 :
        :               :                 :
        :               :                 :
Clear-lock-bit 3        :                 :
Clear-lock-bit 2  Clear-lock-bit 3  Clear-lock-bit 3
Clear-lock-bit 1  Clear-lock-bit 1  Clear-lock-bit 2
        :               :                 :
```

F.5.5. Cautions

The following lists specific multiple CP processing precautions::

- Do not modify a code instruction that immediately follows a LOK on a pipeline machine.
- The program may need more task stacks to take advantage of the increased number of CPs. The maximum number of task stacks is determined by the number of potential concurrent executing tasks with the number of suspended tasks and by the number of CPs.
- The size of an event queue may need to be larger than the number of worker queues that cascade into it. This depends on the dequeueing and arming sequences used for the various queues.
- A task requesting SVC PC$WAKE of another PC$SLEEP task must determine if the intended sleeping task is asleep. If it is not asleep, the task must pause and try the waking operation again. The retry is necessary because the supposedly sleeping task may not have requested its own PC$SLEEP yet. If the waking task gives up too soon, the sleeping task may never be awakened once it finally gets to sleep.
- Queue specified on PC$WFE must be disarmed (either explicitly or at queue definition) before triggering an asynchronous task to queue an item to the specified queue on the PC$WFE.

  Otherwise, the queued item will not correspond to the WFE, and it can result in a parallel task to the original task, leaving the original task in the WFE state.
- Since PASCAL runtime arms the cascaded event queue at definition (and uses the PC$WFE with explicit disarming of the event queue), look for any potential non queue-related dispatch of the procedure and queue threshold crossing dispatch of that same procedure.
- Lock bits that remain set by a task to be cancelled (using PC$CKILL) must be cleared. Otherwise, the lock bits remaining set result in other tasks waiting for the lock bits to clear.
- Use of DEBUG with breakpoints can be sensitive to lock-out due to suspension of a task with lock bits set.

.6. Cascading of Queues

Cascading of queues is a method of reducing the number of a procedure's dispatches when there are several queues set up to dispatch the same procedure. Instead of dispatching this procedure each time any one of the queues crosses its threshold, the software attention item (SAI) of each of these queues is queued to a cascade queue on system queue list 1 (SQL1). The QX value of the cascade queue is usually the same as one of the DCP/OS dispatch queues. A task dispatches when the cascade queue crosses its threshold.

Queues provide a mechanism for triggering the dispatch of new tasks within a program. This mechanism involves the following fields in a queue header:

Threshold

This is a value set by arming the queue using the arm queue (AQ) instruction. The AQ indicates the number of unprocessed items on the queue that trigger the queueing of an SAI. When the threshold value is 0 (which is usual), a single item added to this queue triggers the queueing of the SAI.

SAI

The SAI is set up automatically by DCP/OS. The SAI is queued automatically to a queue on the SQL1, or the run's pseudo SQL1, when the threshold is crossed.

QX

The queue index field in the queue header is the queue list index (QLx) of the queue in SQL1 (or it designates that the SAI contains the address to the cascade queue) that receives the SAI.

.6.1. System Queue List 1

System queue list 1 (SQL1) is architecturally defined as a queue list containing queues of type literal. DCP/OS uses SQL1 to keep track of its four displatch queues plus all cascade queues.

There is a system limit of only 251 cascade queues. This is a limitation caused by the architecture definition of the QX field in the queue header as 8-bits long.

The QX field in a cascaded queue must either indicate another cascade queue or one of the four dispatch queues. Any chain of cascaded queues must eventually end at one of the dispatch queues, if a task is ever to be dispatched.

The run also has a run pseudo SQ1 that allows 251 cascade queues on a run basis.

A cascaded queue is placed on the system SQL1 if it is defined as an SAICPA queue, or if it is placed on the run's pseudo SQL1 and defined as an SAI queue type.

.6.2. Setting Up Cascading Queues

Cascade queues are defined by the QUEUE proc, as are other named queues. The difference is that cascade queues are referenced in the QX parameter by at least one other queue. The following example illustrates a typical usage (see the AAAWRENCH service in this manual for more information.

```
QCASCADE QUEUE SAI 7 0 -,'PNXX',3 .
Cascade queue QDATA1 QUEUE LIST 65535 0 0,'PNXX','QCASCADE' .
data queue QLIT QUEUE LIT 25 0 0,'PNXX','QCASCADE' .
Literal queue
```

Queues of different types can cascade to the same queue, as long as they all invoke the same procedure when their thresholds are crossed.

A cascade queue is automatically added to SQL1, or the run's pseudo SQL1, by the program loader (SYS-LOAD) and is removed when the program terminates.

.6.3. Runtime Example of Cascading

The following steps illustrate a typical use of cascading:

1. An MCT is queued to QDATA1 whose threshold is crossed.
2. The SAI from QDATA1 is queued to QCASCADE, which has a threshold of 0.
3. The item on QCASCADE causes its threshold to be crossed, and the SAI from QCASCADE is queued to QDISP, the priority 3 dispatch queue.
4. A literal can be queued to QLIT at any time, sending a second SAI to QCASCADE. Until QCASCADE is rearmed, no additional SAIs are queued to QDISP.
5. Items are gradually removed from QDISP and dispatched until reaching the SAI from QCASCADE. This SAI indicates that the dispatcher should start procedure PNXX for the run owning QCASCADE.

6. Procedure PNXX removes all items from QDATA1, QLIT, and QCASCADE, then rearms these queues before performing a RTN to the dispatcher.

Without QCASCADE, the following sequence of events occur:

1. An MCT is queued to QDATA1.
2. The SAI from QDATA1 is queued to QDISP.
3. A literal is queued to QLIT.
4. Another SAI for PNXX is queued to QDISP.
5. PNXX is dispatched twice. No checks are made that the first task is complete before the second task is dispatched.

Note that cascading of queues not only reduces the number of dispatches (and possibly the number of concurrent tasks), but it also avoids the problems that can be caused by having two tasks active at the same time in the same procedure.

Glossary

A

AABOOTDEF
A definition element for bootstrap.

AACPADEF
A definition element that defines Communications Processor Architecture (CPA) definitions.

AADEVDEF
A definition element that defines device definitions.

AAFILDEF
A definition element file manager. *See also* FR$.

AAIPMDEF
A definition element for inter-program messages (IPMs). *See also* IP$ and IPC$.

AAPPEQU
A definition element for general port processor (PP) definitions.

AAPPROC
A definition element for port processor (PP) op-codes.

AARUNDEF
A definition element that defines message field offsets, message types, run control table (RCT) entries, console information table entries, system information table entries, and system directory equates. *See also* AV$, MH$, RC$.

AASERVDEF
A definition element that defines the SVC arguments or function codes. *See also* CPA$, PC$, LM$.

AASLDEF
A definition element that defines the loader definitions.

STRPRC
A definition element of the structured Meta-Assembler (MASM) procedures.

SUBENT
A definition element that defines the DCP/OS subroutines. *See also* S$.

WRENCH
The primary definition element, which contains the central processor (CP) instruction set, extended instruction definitions, and module definition MASM procedures.

olute element (ABS)
An element containing a complete program form suitable for execution. Such elements normally occur as output from a collection of relocatable elements.

L
Alternate environment list.

T
Alternate environment table.

lstreams
During program development, the addstreams are symbolic elements. *See* symbolic element.

ı
To set the queue threshold.

Architectural tables.

XREF
A MASM procedure that allows the sharing of the GPL, PST, QL and LA lists between assemblies.

o debug trap
The process of detecting or tracing errors under specified conditions which functions without intervention by a human operator.

$
A part of the run manager services, AARUNDEF. Associated with the following: AV$MCT, AV$AMCT, AV$MSG, AV$AMSG, AV$SUB.

B

batch mode
The execution of programs serially.

block size
Logically contiguous number of fixed-length blocks, which are fixed at 256 bytes for disk files, regardless of the physical device sector sizes (for example, cartridge disk sector size is 256 bytes, but diskette sector size is 128 bytes).

@BUILD
The DCP/OS utility program used to create an executable program. This program resolves intercomponent Communications Processor Architecture (CPA) entity references and automatically generates a dictionary containing the names of these CPA entities.

C

CALL
Dynamic-addressing control instruction that allows a process to move back and forth among procedures.

control mode register set
A mode register set supported by the CPA environment that contains a set of segment descriptor registers (SDRs) and a set of general registers (R0-R15).

CP
Central processor.

CPA
Communications Processor Architecture.

CPA$
All CPA structure services that manipulate entries in the QL, GPL, and PST tables. Specific services create queue lists, link areas, and queues.

CPADEF
A MASM procedure used at assembly time to fix the maximum number of concurrent tasks. The CPADEF MASM procedure defines default reservations for dynamic segments, dynamic queues, and process control stacks.

tingency handling
The feature of a program that may directly dispatch a specific procedure when certain exception conditions are detected. This feature enables programs to survive trivial errors.

ss-reference list
A method used to specify by name a list that is generated in another assembly.

a file
One of the two basic file formats. Data files are user-defined.

?
The Distributed Communications Processor, or a member of the Distributed Communications Processor family.

CPAPP
The part of the collection process that converts OS 1100 format into DCP format.

?FT
The host utility program used to pair with DCP/OS across a channel.

?/OS
The DCP Operating System. DCP/OS is composed of procedures, segments, port processor programs, and queues, which are all CPA-defined architectural entities.

?OSEQU
A library file on the release tape that contains optional MASM utility procedures.

ı
Data set label.

ıp files
Normal data files which are created by the system and have a defined internal structure, mapped by the header block.

E

EEP
External entry point.

element
A named grouping of information typically manipulated as a unit, and typically defining a logical program part such as a subroutine. There are four basic types of elements: symbolic, relocatable, absolute, and omnibus.

$END
The typical assembler program ends with this statement.

end-of-file (EOF)
Terminator - record byte count of X'FFFF'.

EP
Entry point.

element
The substructure of a program file in DCP/OS.

EQ
Symbol used (in syntax) to indicate a simple equated value with no relocation.

EQF
Symbol used (in syntax) to indicate a field defined as an EQUF or simple equate with no relocation.

executable programs
See absolute element.

export-reference
The name used to label architectural tables and prefix their entry points. It is the name a potential sharer of an AT uses to specify where it should be picked up.

F

FCB
File control block.

file manager
The portion of DCP/OS that handles file requests.

ced call
A mechanism whereby the Communications Processor Architecture (CPA) traps all errors in CP programs and all control and errors are transferred to a DCP/OS procedure.

$
File manager services.

P
File request packet. The interface packet that file users pass to file control (file manager) to access files.

eral registers
R0-R15.

L
Gated procedure list.

LDEF
A MASM procedure that defines the procedure-related structure GPL.

W
Highest block written.

Instrumentation buffer pointer.

Interface control block.

Interface control table.

/
Instrumentation control word.

ort reference
The name used to label a potential sharer of an AT. It specifies where to place the exported reference.

$INCLUDE
The typical assembler program begins with this statement, followed by MASM procedures defining the procedure-related structure.

IIBP
Indirect instrumentation buffer pointer.

I/O
Input/output.

IOP
Input/output processor.

IOPDEF
A MASM procedure used with PP programs to define procedure-related structure.

IP$
Prefix to fields defined within the IPM packet.

IPC$
Inter-program message services.

IPC$CONN
The value of an IPM contingency status that means the transmitter issued a connect.

IPC$DISC
The value of an IPM contingency status that means the transmitter issued a disconnect.

IPC$FREE
The value of an IPM contingency status that means the receiver issued a free of the IPM queue.

IPM services
The inter-program message (IPM) services provide a facility to transfer messages between separate, cooperating programs.

IPN
Initial procedure number. This number is defined in the header fields of an executable program.

Link area.

Location counter.

EG
Load segment.

[
Literal type queue.

[
Line module. The hardware in the DCP that terminates serial communications lines, host channel connections, and peripheral connections.

[$
Line module services.

KDEF
A MASM procedure that defines procedure-related structure in the link area.

gical block size
See block size.

B
Least significant bit (rightmost bit).

MAP
The part of the collection process that collects relocatable elements.

ASM
OS 1100 Meta-Assembler. This processor assembles programs for any target machine by processing each instruction as a MASM procedure. All system-wide definitions are defined in definition elements.

CT
Message control tables.

MH$
User-defined message types.

MLF
Module library file.

MSB
Most significant bit (leftmost bit).

O

OMN
Omnibus element. An element of arbitrary format used to store general information in a program file.

P

PC$
Dispatch services.

PD$
Prefix to fields defined in the physical device table.

PDT
Physical device table.

PHYSIO
Physical I/O package. After a file manager is called, the request is validated and I/O is initiated as necessary by queuing off a request to PHYSIO.

PID
Process-ID.

PN
Procedure number.

PP
Port processor.

PPPN
Port processor program number.

PPPT
Port processor program table.

›c
A user-defined assembler directive.

OCDEF
A MASM procedure that assigns the process mode register set to a user procedure.

›cess mode register set
A mode register set supported by Communications Processor Architecture (CPA) environment that contains a set of segment descriptor registers (SDRs) and a set of general registers (R0-R15). User programs must use this mode. (It is automatically specified by the PROCDEF MASM procedure.)

›gram file
One of the two basic file formats. It is defined by the DCP/OS. The program file structure supports subfiles called **elements**.

ſ
Procedure segment table.

ſDEF
A MASM procedure that defines the procedure-related structure PST.

Procedure table.

Queue list.

Queue number.

AL
File qualifier.

EDEF
A MASM procedure that defines the procedure-related structure QL.

Queue table.

Queue index.

R

RC$
Run service.

RCT
Run control table.

RCW
Run control word.

REL
Relocatable.

relocatable element
An element containing a program part in relocatable binary format, suitable for combination with other relocatable elements to produce an executable program (absolute element). Such elements occur most commonly as the output of a language processor to be input to a collection. Usually an element that is produced by MASM from the source element and is stored in a temporary program file.

resident segment
A segment of code or data that resides in memory at all times.

RSI
Remote Symbiont Interface.

S

S$
Common utility subroutines.

SAGSUB
A common utility subroutine segment name.

SAI
Software attention item. The item placed on the appropriate queue of system queue list 1 when the queue threshold is crossed.

SAI PN
The procedure to be dispatched when the queue threshold is crossed.

SCALL
Dynamic-addressing control instruction that allows a process to move back and forth among subroutines.

:T
System control table. Part of AACPADEF.

)
Segment descriptor.

)F
System data format file. One of three major file types generated by the OS 1100 system.

)R
Segment descriptor register.

CB
Segment name for interface control block. Part of AACPADEF.

ecific list
A method used to specify by name a named list within the assembly. This allows the use of multiple lists within an assembly.

'DT
Segment name for physical device table. Part of AAFILDEF.

:TN
Each process starts at a given procedure and may CALL or SCALL additional procedures and subroutines. When the process finally works, back up the CALL/RETURN stack by issuing paired RTN/SRTN instructions.

N
System segment number.

T
System segment table. The SST starts at a fixed block number in an absolute element and is used as a directory for the rest of the system.

TDEF
A MASM procedure that defines the SST.

C
System service calls. SVC is the MASM procedure (instruction) that you must use to invoke a system service call.

ymbolic element
An element containing information generally in human-intelligible format. The most common use of symbolic elements is as source language to be input to a language processor. A symbolic element contains variable-length text records terminated by end-of-file sentinels. Each record begins with a 16-bit count that specifies the number of data bytes in the record. The ASCII data bytes follow. Each element ends with an end-of-file terminator that is a record byte count of X'FFFF'.

SYS
The DCP/OS uses various files for system functions. These file names all begin with SYS.

SYSINF
Segment name for system information table defined in AARUNDEF.

SYSLIB
The DCP/OS file for system library.

SYSJOB
The DCP/OS file for system runstreams.

SYSLMC
The DCP/OS file for line module code.

T

transient
A segment of code or data that is loaded into memory only as required and is released (overwritten) when it is no longer needed.

V

virtual address range
The range X'0000' to X'FFFF'.

/

›rkstation
The workstation is a UTS terminal physically connected to a port owned by the DCP/OS.

REC
The MASM procedure that initializes PP programs.

RENCH
The MASM procedure that initializes CP programs.

Bibliography

DCP Series Distributed Communications Processor Operator System (DCP/OS) Operations Reference Manual (UP-11541). Unisys Corporation.

DCP Series Implementation Reference Manual, Volume 1, Volume 2, Rev. 1, and Volume 3 (UP-12728). Unisys Corporation.

DCP Series Telcon Internals Programming Reference Manual (UP-9255). Unisys Corporation.

DCP Series Telcon Terminal Handler Platform Programming Reference Manual (UP-13460). Unisys Corporation.

DCP Series Telcon Operations Reference Manual (UP-9256). Unisys Corporation.

OS 1100 Collector Programming Reference Manual (UP-8721). Unisys Corporation.

OS 1100 Meta-Assembler (MASM) Programming Reference Manual (UP-8453). Unisys Corporation.

OS 1100/DCP Series Communications Delivery Software Configuration Guide (UP-9957). Unisys Corporation.

OS 1100/DCP Series Communications Delivery Software Configuration Reference Manual (UP-11580). Unisys Corporation.

OS 1100/DCP Series Communications Delivery Software Installation Guide (UP-9956). Unisys Corporation.

Index

A

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

S

T

V

W

X

Y

Z