B5000/B6000/B7000/A SERIES BASIC SYSTEM SUPPORT STUDENT GUIDE

March 27, 1986
Relative to Mark 36
Version 2.2

TABLE OF CONTENTS

INTRODUCTION

COURSE DESCRIPTION

ABSTRACT

This course is designed for those personnel who are responsible for maintenance, consulting and training of the large system environmental software and hardware. Emphasis is on the operating system, compilers, basic hardware, associated selected support utilities, and other subjects required for developing System Support positions.

SYSTEM SUPPORT POSITION DESCRIPTION

Provide cost-effective support in areas of vendor software, data communications, program development systems, capacity planning, quality control and database management. Technical services include consulting, training and support. The primary users of this group is the programming staff and operations.

Other key areas:

- Operations training
- Operational support tools
- Operations standards (established with operations manager)
- Establish hardware/software event monitoring and trending
- Assist in development of programming standards
- Effective testing procedures
- Effective testing environment
- Develop and/or acquire program development tools
- Provide internal programmer training
- Establish system software implementation policies and procedures
- Establish system software testing procedures
- Maintain library of technical documents
- Distribution of technical documents
- Research an development of current and projected technical hardware/software/techniques

PREREQUISITES

Prerequisites for this course are:

Basic understanding of large system concepts and their programming languages: **COBOL, ALGOL** and **WFL.** Support utility experience including: **CANDE, LOGANALYZER, DUMPALL,** and **FILEDATA.**

These requirements will have been met if the following courses have been completed:

FUNDAMENTAL LARGE SYSTEM SKILLS	(EP6190)
ADVANCED LARGE SYSTEM SKILLS	(EP6190)
ALGOL	(EP6314)
COBOL74	
(or acquired experience)	

CLASS MATERIALS

Title	Burroughs Form Number
A Series System Architecture (VOL-2)	5014954
System Software Support	5014434
B5900 Reference Card	5012099

Student Guide

Appendix

- A CODE file construction ✓
- B EVENT word layouts
- C File Information Block (FIB) words
- D Disk File Header (DFH) layout
- E A9 system description
- F Program Information Block (PIB) words
- G Misc information
- H ATTABLEGEN
- I Review questions
- J Software compilation WFL job

COURSE OBJECTIVES

Understand basic large system hardware.

Understand Burroughs stack architecture.

Understand code file structure.

Understand processor operators.

Analyze language constructs for correct and efficient code generation.

Understand basic components of the operating system.

Understand system software generation techniques.

Understand a majority of the elements of a Program dump.

Understand some of the items of a System dump.

Ability to diagnose program failures through the use of a Program dump and a System dump.

SCHEDULE

DAY 1

Introduce the student with the instructor and vice versa.

The design and content of the class is reviewed.

An entrance exam is given to determine the need for review subjects.

A hardware overview is given.

Universal I/O is discussed.

DAY 2

Data word formats are reviewed.

Basic memory management is discussed.

Arrays of all types are covered.

Actual Segment Descriptor (ASD) memory is introduced.

DAY 3

Reference words are presented.

Basic stack architecture is introduced.

Processor registers are discussed.

First program dumps are generated.

DAY 4

Stack Linkage words are covered.

Program initialization is discussed.

Parameter passing is presented.

DAY 5

DUMPANALYZER is covered.

Advanced stack architecture topics are introduced.

DAY 6

Code file construction is covered.

DAY 7

Reverse Polish Notation is reviewed.

Machine operators are introduced.

Samples of code generation are given.

DAY 8

More processor operators are covered.

DAY 9

Interrupts are covered.

FIBs are discussed.

System software generation is covered.

Program failure analysis techniques are enhanced using program dumps.

DAY 10

A review of the homework assignment is done.

An exit exam is given and reviewed.

A Question and Answer period, if time permits.

HARDWARE OVERVIEW

MONOLITHIC SYSTEMS

Resource sharing of memory, CPUs, and IO processors (such as on B7800s and B7900s).

Applications have access to all of: memory, CPUs, and IO processors.

ADVANTAGES

Better utilization of processors.

Reduced operational support/scheduling.

DISADVANTAGES

Memory limitation is 6 MB. vs A5P

TIGHTLY COUPLED SYSTEMS

Multiple processor configured under control of a single Master Control Program (MCP).

Each local box has own dedicated Central Processor Unit (CPU), IO processor, and local memory.

Communications between local boxes is accomplished through GLOBAL memory.

ADVANTAGES

Increased memory visibility for entire combined system. but not all visible

Multiple machine connectivity (i.e. B5900, B6900).

Maintain single machine environment.

DISADVANTAGES

Reduced fault tolerance.

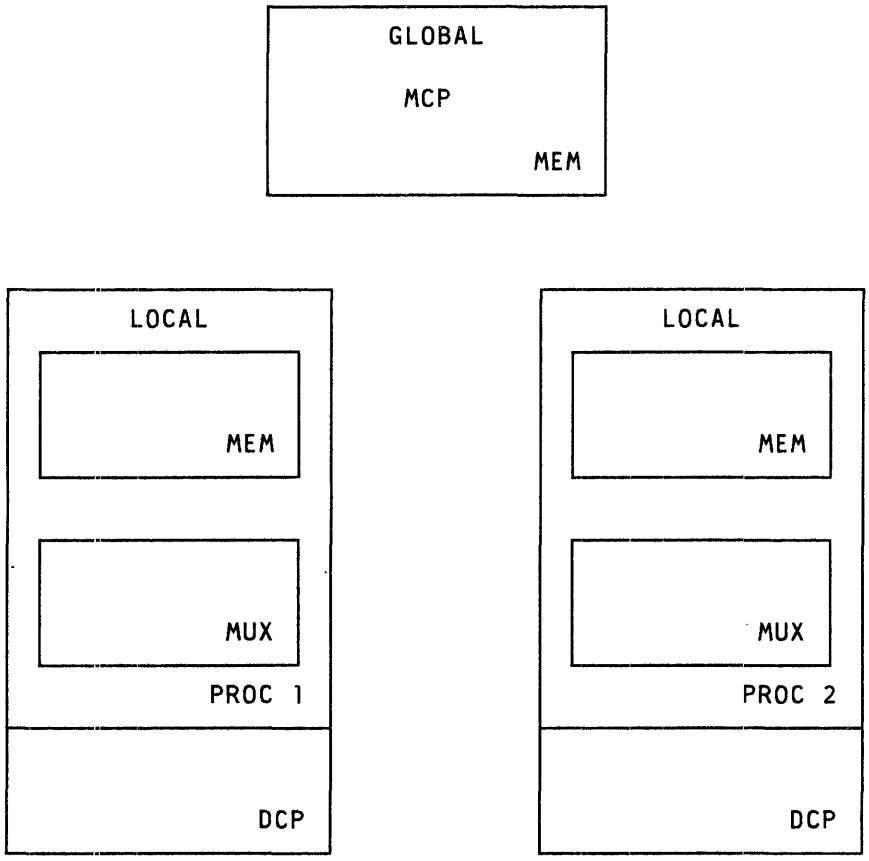
Adding or deleting subsystems requires Halt Load (H/L).

Partitioning of subsystems.

Crowding of GLOBAL memory.

Overhead factor in support of Tightly Coupled System (TC).

TIGHTLY-COUPLED SYSTEM DIAGRAM



LOOSELY-COUPLED SYSTEMS

Communication between multiple independent systems.

Each system has its own copy of the MCP.

Systems run independently of each other.

GLOBAL memory.

Using GLOBAL memory as a mailbox between systems.

Inter-System Control (ISC).

High speed electronic interface between systems.

DIRECT IO interface now available.

Remote Job Entry (RJE).

Other Datacomm options. WWBNA

ADVANTAGES

Reduced overhead as compared to tightly-coupled system.

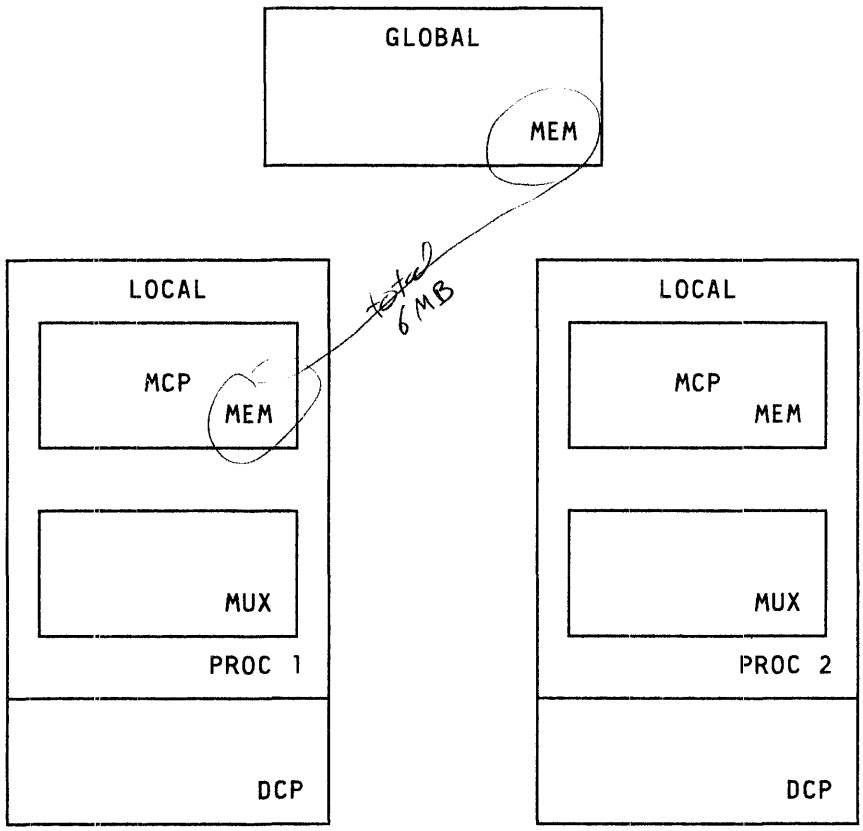
Independent systems.

DISADVANTAGES

Potential split databases.

Volume communication between systems may be slow.

LOOSELY-COUPLED SYSTEM DIAGRAM



7800 6 A series

like tightly coupled mem org.

ENVIRONMENTAL MEMORY

One CPU ?

Partitioning memory into LOCAL areas called Address Spaces.

Each address space is given an Address Space Number (ASN).

Shared partition is visible to all LOCAL boxes.

Application memory visibility is: 6 MB data, 6 MB code.

Increased overall memory capacity.

System must still be configured with SUBSYSTEMs.

ADVANTAGES

Increased overall memory capacity.

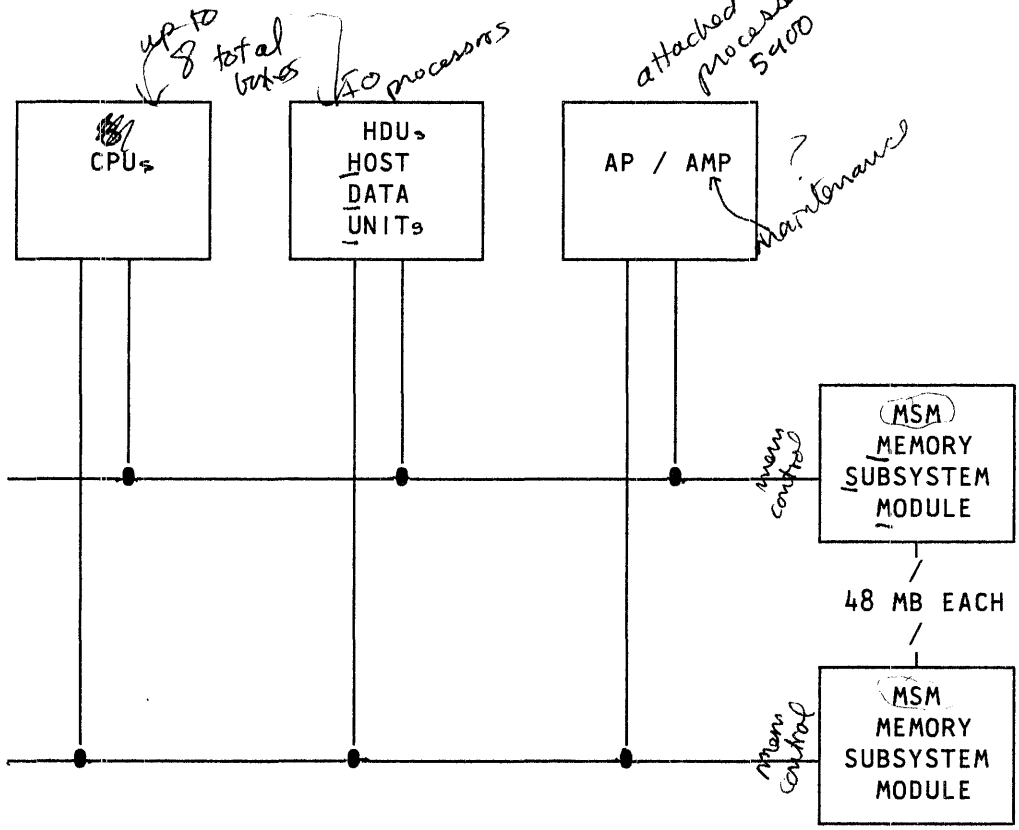
Sharing of CPU and IO processors.

DISADVANTAGES

Potential GLOBAL memory crowding.

Sites must still partition application location (SUBSYSTEM).

B-7900 REQUESTOR-TYPE ORGANIZATION



PARTITIONING ENVIRONMENTAL MEMORY

Partitioning is done in increments of "page"s.

A "page" is 128KW of memory.

No more than eight (8) pages (1 MW) of memory may be included in any address space.

Number of Shared pages will be the same for each Address Space.

Size of each Local may be different, depending upon the machine.

On an A9:

Assume that 16 pages are available
Assume site specified Shared as three (3) pages
MCP would create two (2) Locals with five (5) pages each (8 pages max - 3 pages global = 5)
MCP would create a third Local with three (3) pages (16 pages total - 3 pages for Shared - 2 locals with 5 pages each = 3)

On an A15:

The site can specify the size of the Shared component
Can also specify the size of each Local component
To do this, must use a configuration file

B7900 CENR AND DENR

B7900 computers have additional registers in the processors called:

CENR Code Environment Number Register

DENR Data Environment Number Register

Each **xENR** can address up to **1 MW** of memory.

Makes total memory visible at one time **2 MW.**

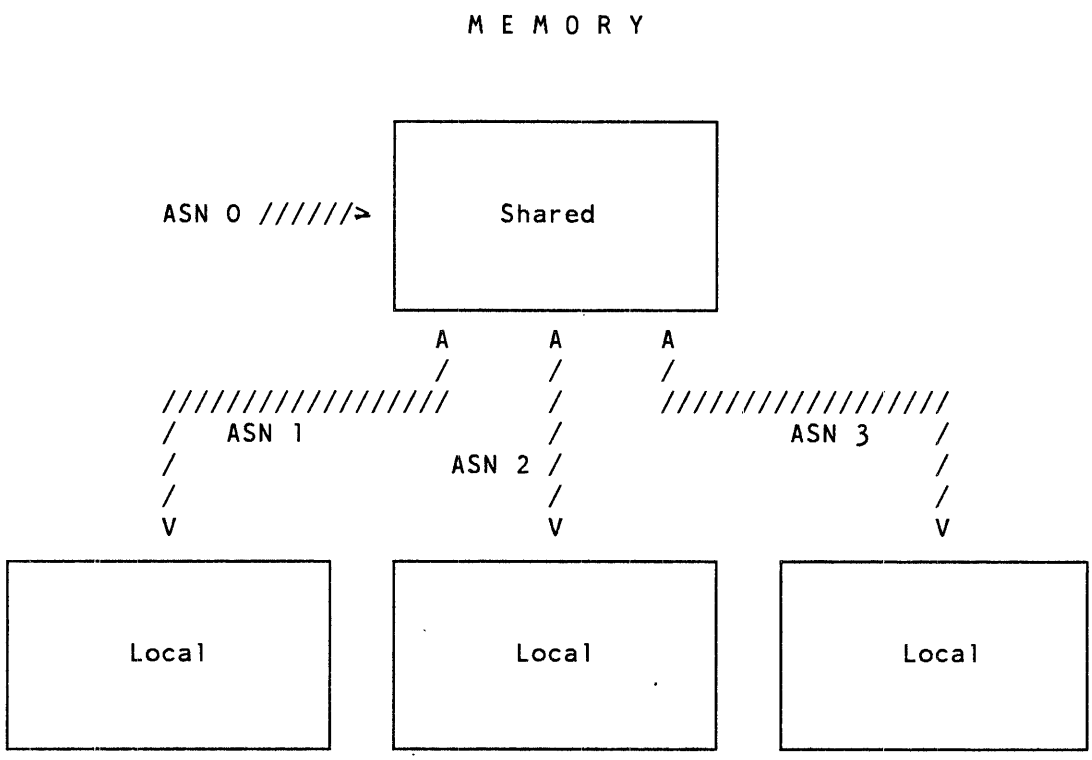
Configuration like the following is possible:

	Shared	Local	(Units are in Pages)
Code	4	2	
Data	3	4	

Above yields a total of **13** pages or around **1.6 MW.**

Limitation is that no Shared component plus any Local component can exceed eight pages.

ENVIRONMENTAL MEMORY DIAGRAM



SYSTEM NETWORKING

BURROUGHS NETWORK ARCHITECTURE

Burroughs Network Architecture (BNA) is the environmental software required to communicate between systems.

(GLOBAL connected systems, ISC, other datacomm)

NETWORK services.

HOST services.

File transfer.

HOST to HOST.

HOST through HOST to HOST.

File OPEN. may open file on other system.
portfio.

Virtual terminal.

Task execution.

HOST mix display and control.

Least cost and alternate route routing.

Transport media: ISC, DATACOMM, GLOBAL.

INTER-SYSTEM CONTROL (hub)

Inter-System Control (ISC) is a high speed channel electronic interface system to system.

Connection is point to point (each connection requires a path).

Up to 4/16 ports per each connecting unit (HUB).

Depending on which type HUB.

Most DLP and PCC systems can connect.

Limited to only BURROUGHS systems.

DIRECT IO interface available (BNA not required).

REMOTE JOB ENTRY

Remote Job Entry (RJE) provides limited file transfer and connectivity without requiring BNA.

File transfer.

Virtual terminals.

LARGE SYSTEMS PROGRESSION

The following pages are intended to show the changes that have occurred in Burroughs Large Systems hardware over the years.

It is also hoped that those new to Burroughs equipment will become familiar with a little of Burroughs' history.

B-5500

Monolithic machine.

"Floating" I/O channels could connect to any I/O device.

Limited memory of 32 KW.

First virtual memory machine, circa 1960.

Discrete components (transistors, resistors, etc.) and diode-register logic as opposed to chips.

B-6700 c. 1967?

Chip architecture version of B-5500. MCP rewrite

Multiplexor now has hard I/O channels (one per device controller).

Faster.

Has memory capacity of one million words, although early style core memory units were too large to physically assemble units close enough for 1 MW.

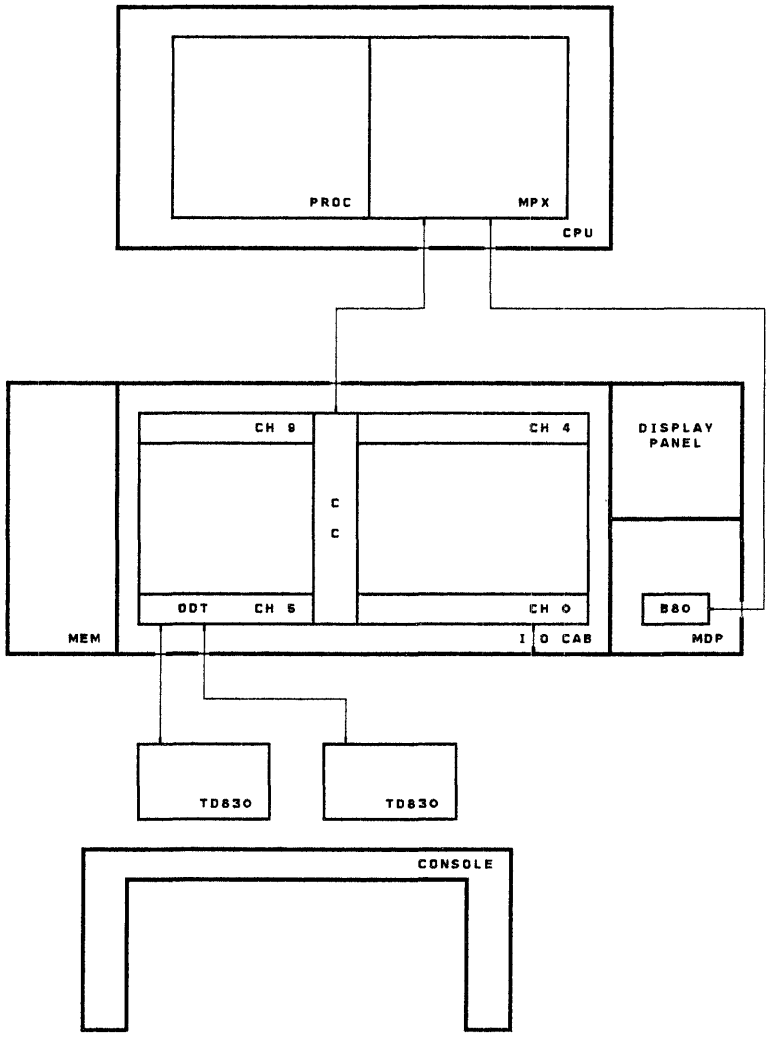
Evolved to planar core memory which could complete the 1 MW memory.

B-6800

Introduction of "Global" memory and the concepts of SUBSYSTEMs, "tightly-coupled", and "loosely-coupled" machines.

Integrated Circuit (IC) memory available on this mainframe (Model II) 16K chips.

Fixing design problems or changing design requires wiring or board changes. This is complicated, expensive, and very time-consuming.



LEGEND:

MDP: MAINTENANCE DIAGNOSTIC PROCESSOR

CHn: I/O CHANNEL

MPX: MULTIPLEXOR; I/O PROCESSOR

CC: CENTRAL CONTROL; TRAFFIC COP FOR I/O CHANNELS TO TRANSFER DATA TO MPX

B-6900

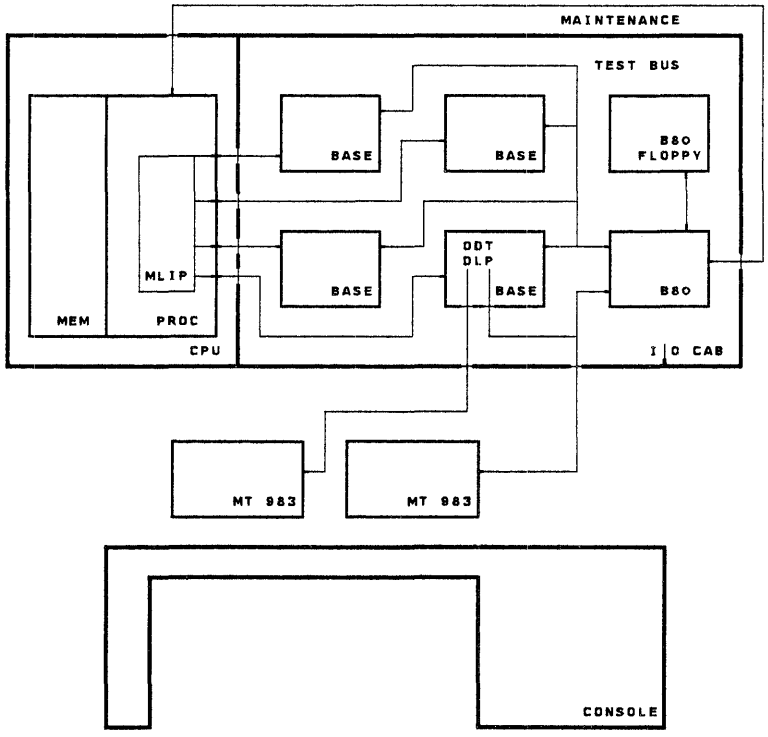
Same processor as B-6800.

IC memory on all models.

Multiplexor becomes MLIP (Message Level Interface Processor).

Universal I/O (UIO) devices are added. These are PROM driven, so any logic changes usually require only PROM changes.

Footprint for I/O boxes approximately 1/8 of hard I/O boxes from previous architecture. Up to 64 I/O channels fit where 10 used to.



B-5900

Processor is now "soft" microcode driven.

card level Maintenance (vs chip level)

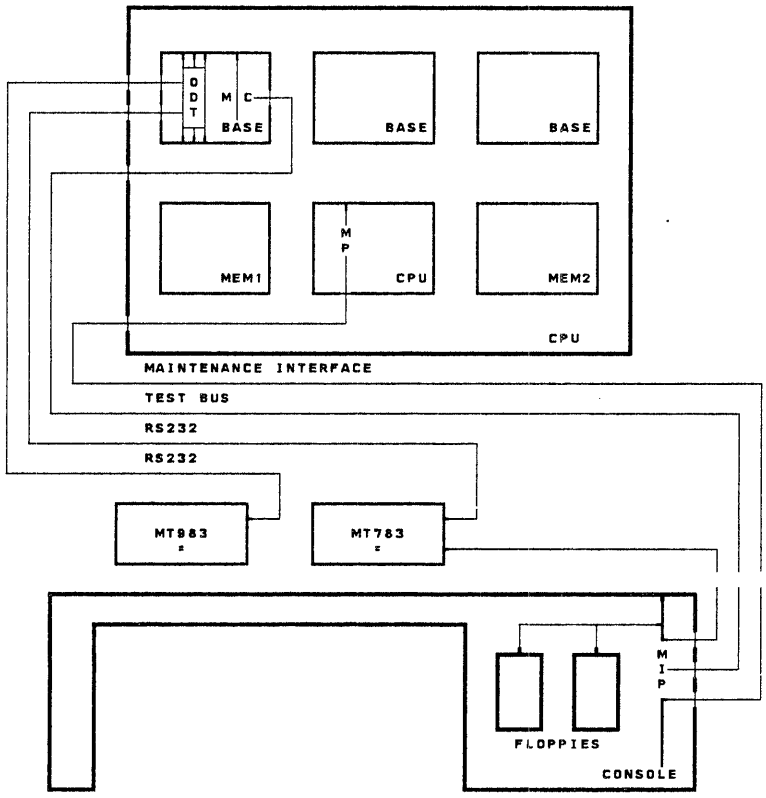
16K chips on B-5930 (upright).

64K chips on B-5920 (low boy).

UIO is same as B-6900.

Still only one million words of memory (6.2 Mb).

Entry level system to Large Systems.



LEGEND:

MIP: MAINTENANCE INTERFACE PROCESSOR

MC: MAINTENANCE CARD

MP: MAINTENANCE PROCESSOR

B-7900

B-7000 entry into UIO.

B-7800 processor with memory addressing capabilities greater than one million words (CENR and DENR), plus some instruction enhancements.

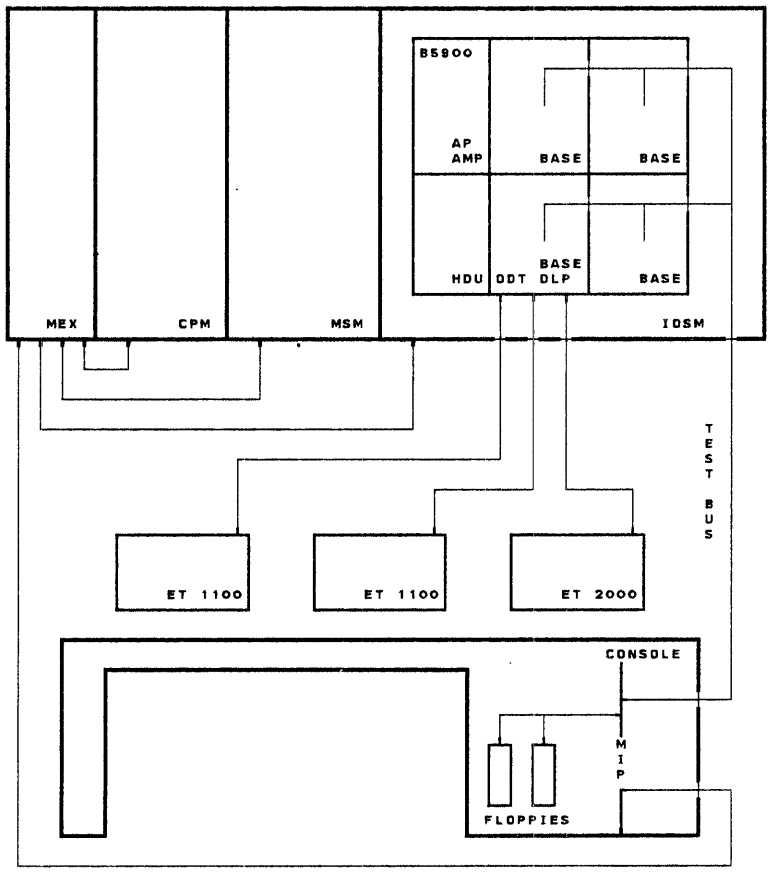
Total addressable memory 96Mb.

B-5900 is Maintenance Processor with capability of Attached Processor online.

B-7800 IOM becomes HDU with 8Mb bandpass per port (versus 1 to 2 Mb IOM).

Still memory limitations, although new limit is 1 Mw Code plus 1 MW Data.

64K memory chips.



LEGEND:

MEX: MAINTENANCE EXCHANGE

CPM: CENTRAL PROCESSOR MODULE

MSM: MEMORY SUBSYSTEM MODULE

IOSM: INPUT OUTPUT SUBSYSTEM MODULE

AP/AMP: B5900

HDU: HOST DATA UNIT

A9

B-6000 entry into microcode systems.

New logic: current mode (emitter-coupled logic) plus a pipeline processor with three stages.

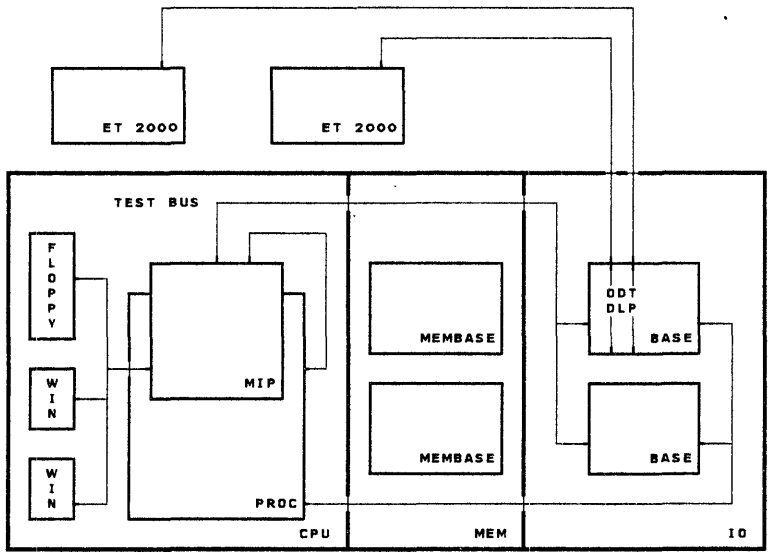
UIO same as before.

Memory capability now greater than one million words.

Memory released ad 24 Mb, but capable of 96 Mb.

64K memory chips.

ASD capable without hardware upgrade.



A3

"A" series entry level system to large systems.

A13K is dual processor.

A10 is dual

UIO same as before.

Capable of greater than one million words of memory.

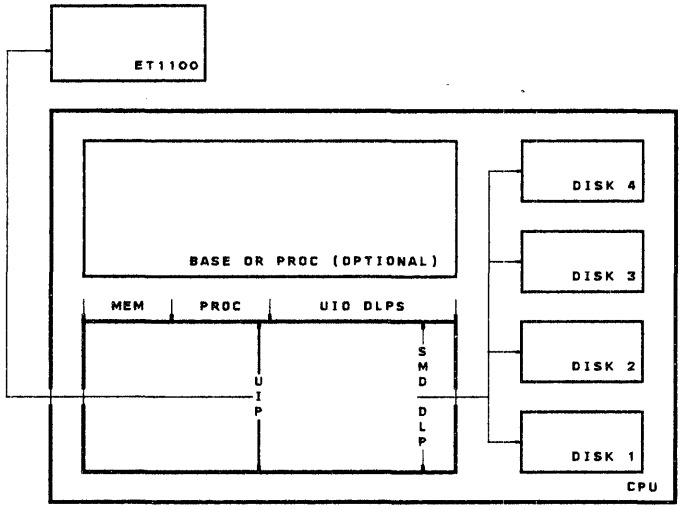
Max memory is 48 Mb.

256K memory chips.

In-built winchester disks. up to 4

Has monolithic 2-by capability (two processors can see, share, and address all of memory.

ASD capable with hardware upgrade.



LEGEND:

UIP: USER INTERFACE PROCESSOR

SMD: SYSTEM MAINTENANCE DISK

A10

Similar to A9.

Has 256K memory chips.

Has improved arithmetic capabilities.

Monolithic 2-by machine.

ASD capable without hardware upgrade.

A15

7900 upgrade

only one mem controller

Largest current system in Burroughs line.

Monolithic B7000 requestor type architecture.

Same UIO as B7900.

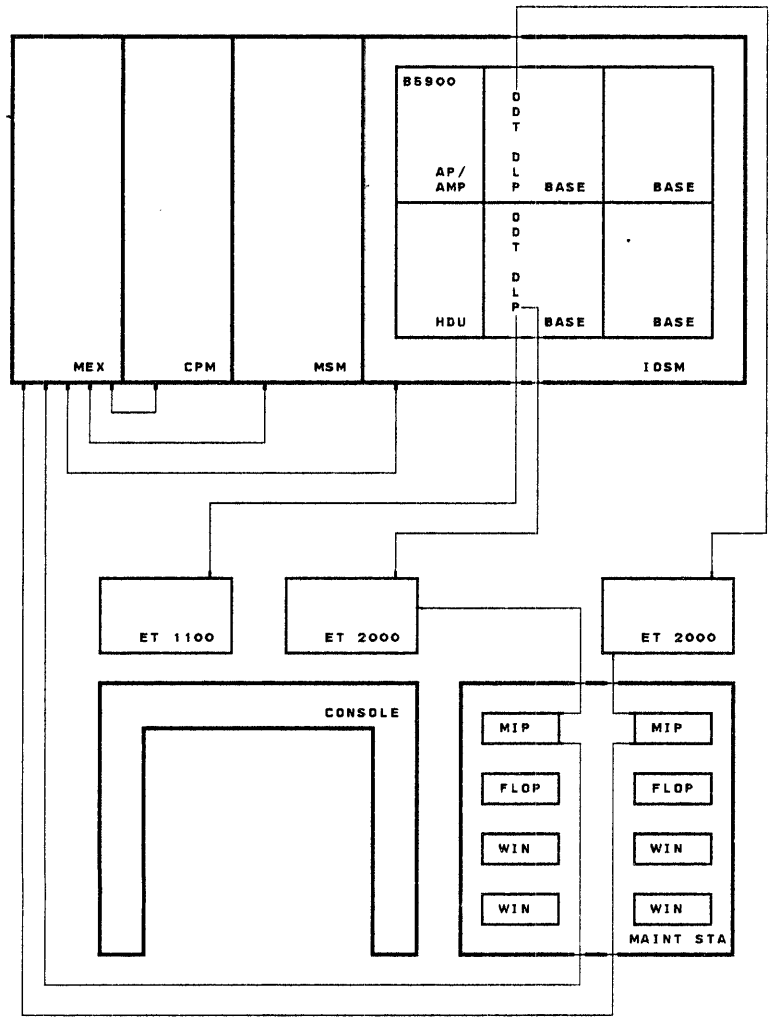
New processor and memory with purgeless cache.

Current maximum memory, as documented by marketing, is 96 MB.

256K memory chips.

ASD capable with hardware upgrade.

≤ max 6 CPUs?



LEGEND:

SMP: SYSTEM MAINTENANCE PROCESSOR

NOTE: BASE 0 (ZERO) IS SHARED BY SMP AND A15.

UNIVERSAL INPUT/OUTPUT

UIO PHILOSOPHY

Universal I/O was originally developed to reduce the amount of work done by the mainframe and offload it to a separate I/O subsystem.

The intent of UIO is that any given I/O is generic to the MCP. In actual fact, this is not so. A card reader can't be read backwards, for example. However, much of the work of older "hard" I/Os has now been offloaded to a "smart" I/O subsystem.

MAINFRAME INTERFACE

All Universal I/O (UIO) operations originate on the mainframe and connect to the UIO Subsystem through an I/O processor.

I/O processors are known as HDUs (Host Data Unit) [B-7900 and A15] or MLIPs (Message Level Interface Processor).

The output from these I/O processors is called a Message Level Interface (MLI) which is an I/O protocol for UIO ports.

I/O BASES

24 cards

An I/O Base is nothing more than a rack of circuit boards with a common backplane.

Certain cards must be in a base. These include:

- o Terminator Cards (TC) at both ends.
- o Base Control Card (BCC).
- o Maintenance Card (MC) to the left of the right-hand Terminator Card.

Other cards may optionally appear in a base. They are:

- o Distribution Cards (DC).
- o Path Selection Modules (PSM).
- o Line Expansion Modules (LEM).
- o Data Link Processors (DLP).

TERMINATOR CARD

A Terminator Card (TC) provides electrical signal termination for the common bus (backplane) which all cards in the base connect to.

There must be one at each end of the bus.

BASE CONTROL CARD

A Base Control Card (BCC) contains information about which host owns which cards (DLPs). This allows two host systems to **FREE** and **ACQUIRE** different devices.

At Halt/Load time, the **BCC** can also inform the **MCP** what devices reside in this base.

MAINTENANCE CARD

Provides a path for maintenance to the base. This maintenance can be used to test cards in the base.

Provides clocking for the base.

DISTRIBUTION CARD

A Distribution Card (DC) provides the interface between the base and the host.

A DC communicates with the host through an MLI.

A DC communicates with the base through the DLI (backplane bus).

There will be one DC for each host which has an MLI into this base.

PATH SELECTION MODULE

= controller

If more than one host talks to the same base (i.e. more than one DC in a base), a Path Selection Module (PSM) needs to be inserted into the base.

The PSM provides a "traffic cop" function to insure that both mainframes do not attempt to use the backplane of the base or the BCC simultaneously.

LINE EXPANSION MODULE

A Line expansion Module (LEM) enables a host to be connected to more bases by expanding one MLI to up to seven MLIs.

Even though a LEM physically resides in a base, it accepts only electrical power from the base. It does not use the backplane for communication with other elements in the base.

LEMs are used when a port needs to control more than one DC (eight DLPs). By feeding an MLI (port) into an LEM, the output from the LEM can be input to from two to seven DCs which yields control over sixteen to fifty-six DLPs.

DATA LINK PROCESSOR

Data Link Processors (DLP) are device-dependent logic cards that provide the interface between the host(s) and the peripheral device(s).

A DLP can consist of two, three, or more cards, depending upon the device type.

Some of the current DLPs are:

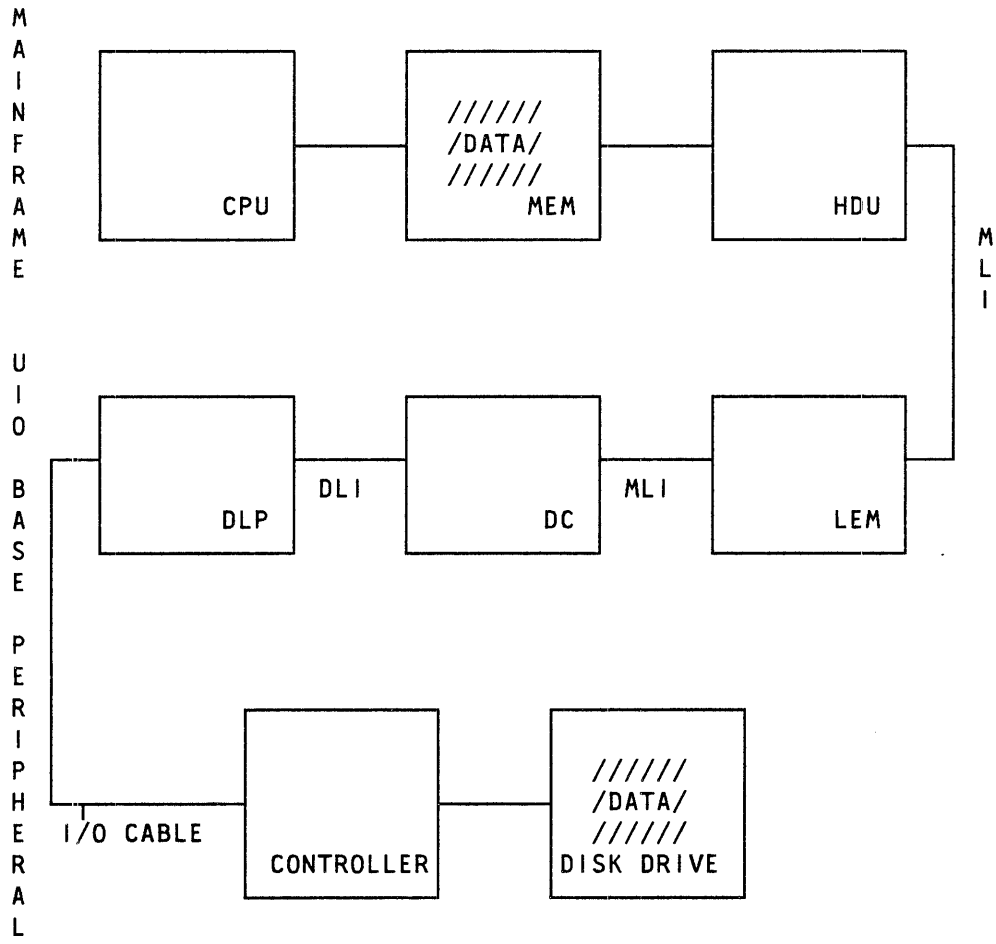
CR1	Card Reader
HT1	206, 207, 659, 677 Disk Pack
MT1	PE Tape
MT2	PE/GCR Tape
ODT1	Operator Display Terminal
LSP1	Line Support Processor (Datacom)
TP2	Buffered Printer 1200/2000 lpm
NSP3	Datacom Network Support Processor

DCDLP

each path has a dlp

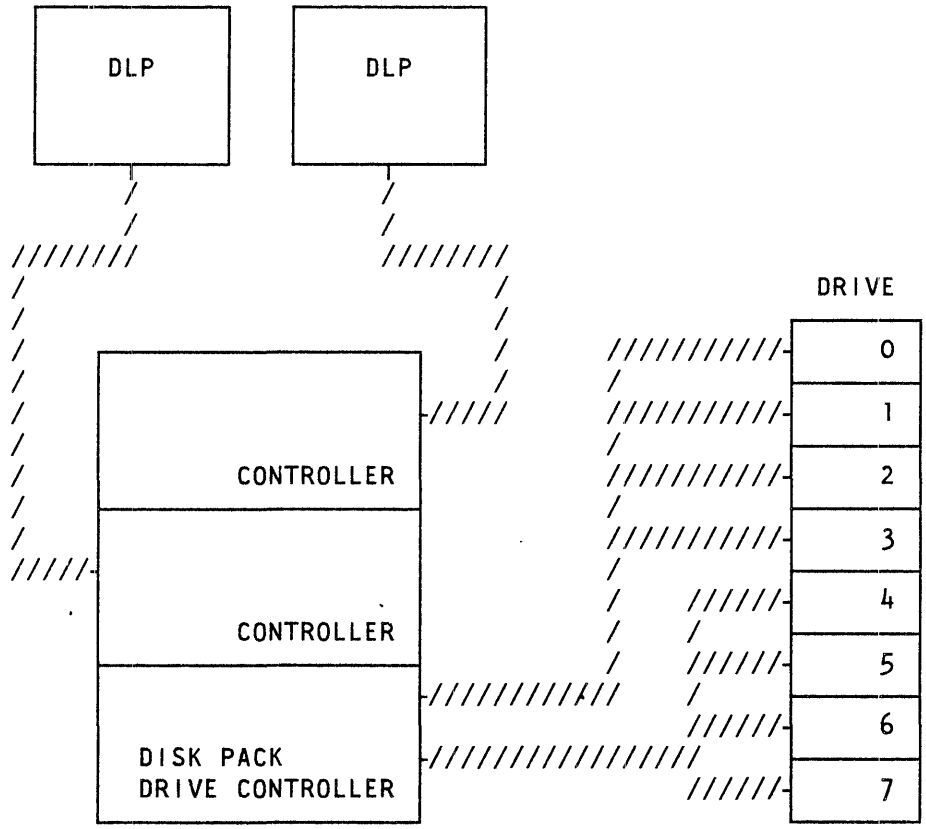
UIO EXAMPLE

A typical path for data in the UIO environment.



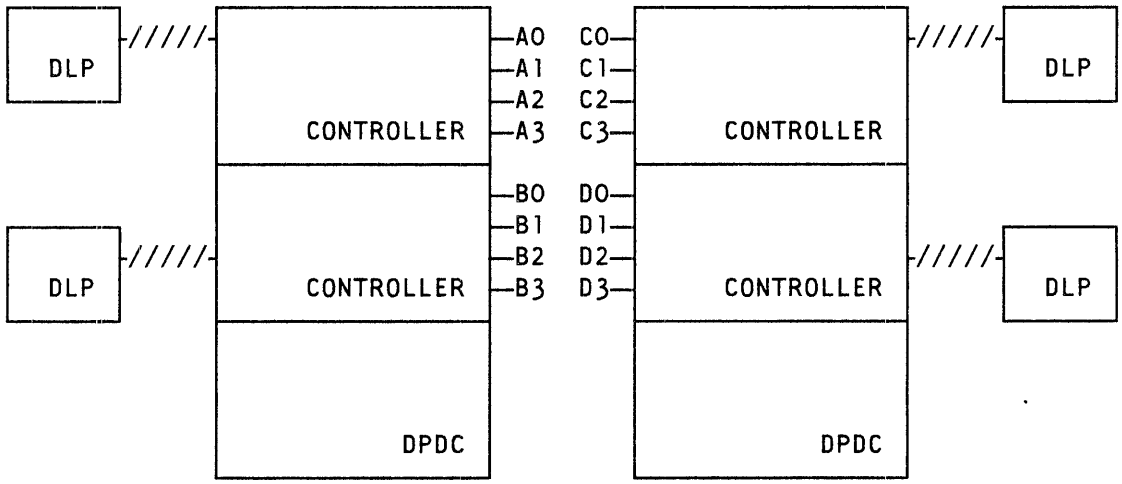
FROM DLP TO DISK PACK

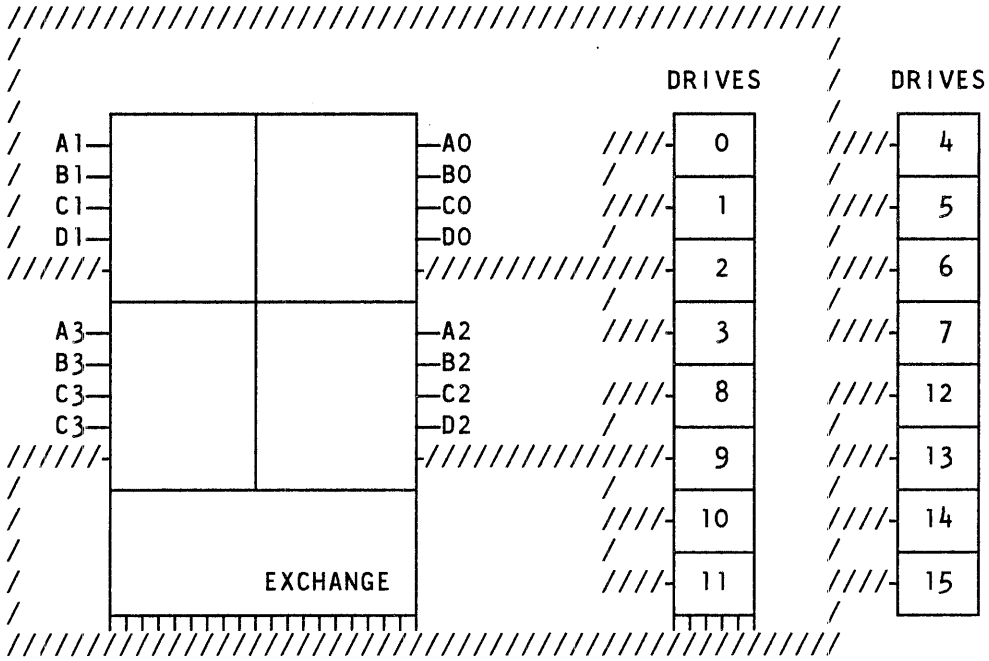
The following diagram is typical of what is outboard from a disk DLP (without an exchange). It shows a 2 X 8 configuration.



FROM DLP TO DISK PACK

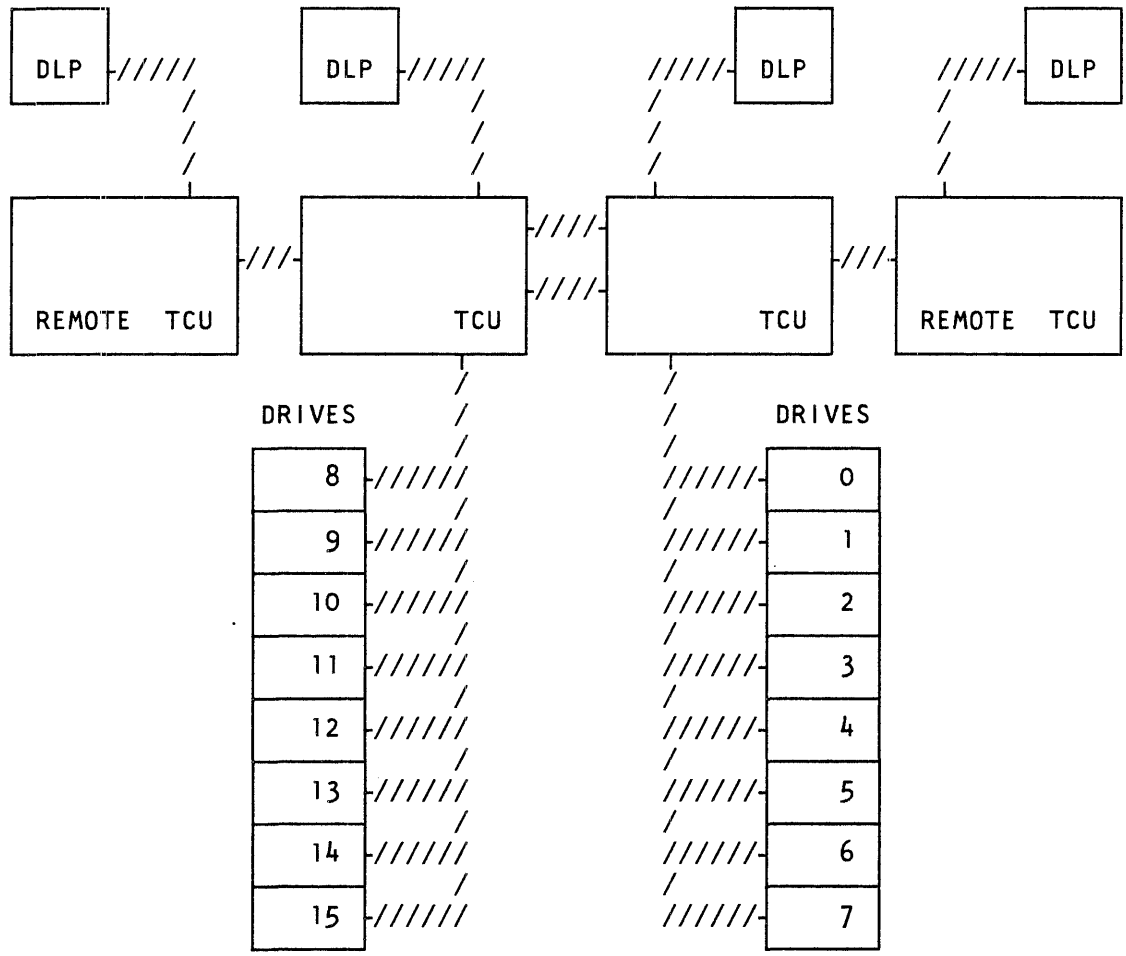
The following diagram shows a typical **4 X 16** disk pack configuration with exchange.





FROM DLP TO TAPE DRIVE

The following example shows a typical GCR/PE configuration.



LEGEND:

DLP: DATA LINK PROCESSOR (UIO)

TCU: TAPE CONTROL UNIT

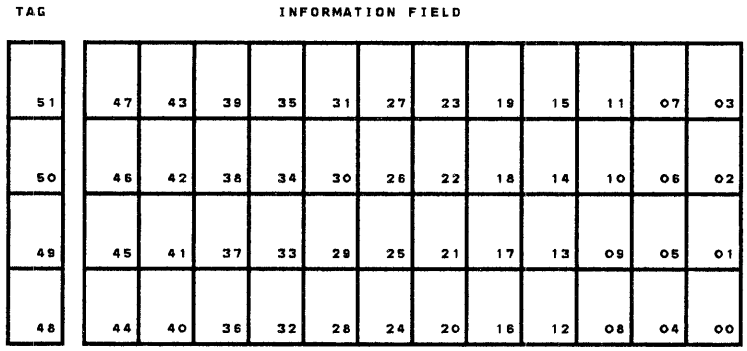
NOTE:

EXCHANGE CAPABILITIES ARE BUILT INTO THE TCU'S

DATA WORD FORMATS

GENERAL INFORMATION

Words are the fundamental units of data.



TAG FIELD

Provides general interpretation of data contained in word information field.

The Tag:

Bit 51 of the TAG is only valid on E-MODE systems.

Bits 50:3 are the only bits currently valid.

Current value range: 0-7.

Even tags serve primarily as computation arguments.

Odd tags primarily serve as reference arguments or control structures.

Bit 48 is considered the "Memory Protect" bit; Most operators will get an error if they try to update a word where bit 48 is on.

INFORMATION FIELD

Main portion of the word.

Interpretation depends on TAG value.

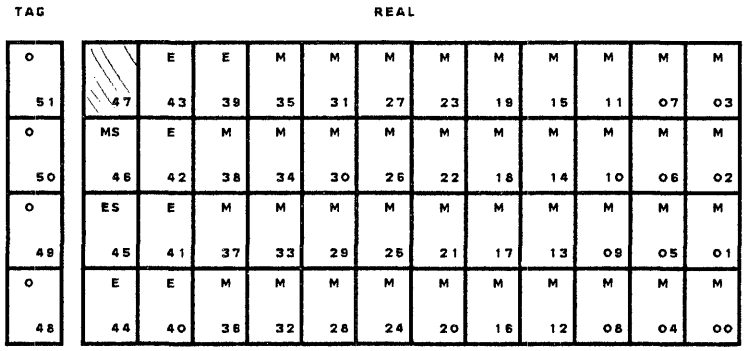
Bits 47:48.

Meaning of the word may also depend on context and different bit values.

Bits may wrap from 0 to 47 if operators require.

SINGLE PRECISION OPERANDS

REAL



TAG 0

46:01 Sign of the **MANTISSA** (0 = positive , 1 = negative)

45:01 Sign of the **EXPONENT** (0 = positive , 1 = negative)

44:06 **EXPONENT** field.

38:39 **MANTISSA** field.

MANTISSA is the magnitude of the number before scaling.

EXPONENT is the power of eight to which the **MANTISSA** is scaled.

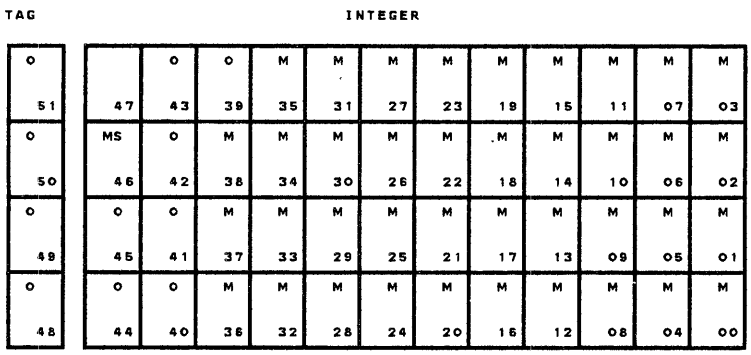
Formula for calculation of decimal value:

(MS) (M) * (8 ** (ES) (E))

Limits on the range of values for a **REAL** values are:

Absolute value may not exceed 4.31359146674 @ 68
Except for zero, absolute value may not be less than
8.75811540203 @ -47

INTEGER



TAG 0

46:01 Sign of the **MANTISSA** (0 = positive , 1 = negative)

45:01 Sign of the **EXPONENT** (Always 0)

44:06 **EXPONENT** field (Always 0).

38:39 **MANTISSA** field.

MANTISSA is the magnitude of the number before scaling.

Formula for calculation of decimal value:

(MS) (M)

INTEGER is the same as a **REAL** except that the **EXPONENT** is always 0.

The compilers force the exponent 0 by generating integerizing operators.

The hardware can not distinguish between a **REAL** or a **INTEGER**.

The absolute value for an **INTEGER** may never exceed:

549,755,813,887

BOOLEAN

no equiv in cobol

TAG	BOOLEAN											
0 51	47	0 43	0 39	0 35	0 31	0 27	0 23	0 19	0 15	0 11	0 07	0 03
0 50	0 46	0 42	0 38	0 34	0 30	0 26	0 22	0 18	0 14	0 10	0 06	0 02
0 49	0 45	0 41	0 37	0 33	0 29	0 25	0 21	0 17	0 13	0 09	0 05	0 01
0 48	0 44	0 40	0 36	0 32	0 28	0 24	0 20	0 16	0 12	0 08	0 04	B 00

TAG 0

00:01 **BOOLEAN** logical value (1 = TRUE, 0 = FALSE)

positive integer ϕ

The hardware uses bit 0 for conditional branching.

All other bits are ignored.

The hardware can not distinguish between a **REAL**, **INTEGER** and a **BOOLEAN**.

Other bits may be on or off, but they are not used in condition checking.

Many logical operations operate on all bits in parallel.

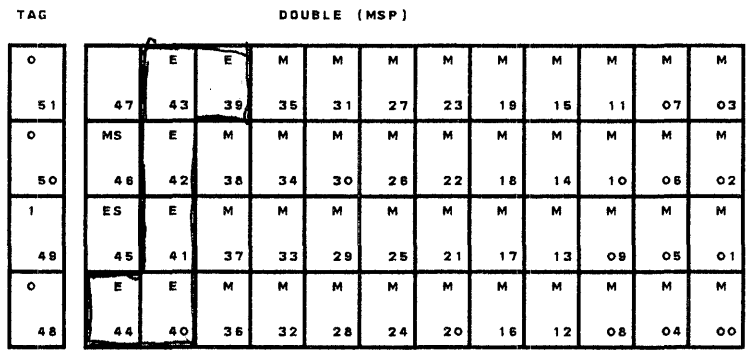
ONE--QUESTION PROBE

If **REAL**s, **INTEGER**s, and **BOOLEAN**s all have the same tag of zero, how does the processor know what type of data it's working with and what to do with it?

different operators

DOUBLE PRECISION OPERANDS

MOST SIGNIFICANT PART -- MSP



TAG 2

46:01 Sign of the **MANTISSA** (0 = positive , 1 = negative)

45:01 Sign of the **EXPONENT** (0 = positive , 1 = negative)

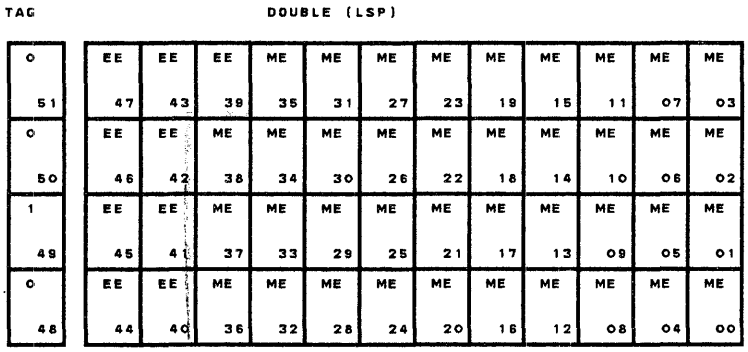
44:06 Low order 6 bits of the **EXPONENT.**

38:39 The integer portion of the **MANTISSA.**

MANTISSA is the magnitude of the number before scaling.

EXPONENT is the power of eight to which the **MANTISSA** is scaled.

LEAST SIGNIFICANT PART -- LSP



TAG	DOUBLE (LSP)											
0	EE	EE	EE	ME	ME	ME	ME	ME	ME	ME	ME	ME
51	47	43	39	35	31	27	23	19	15	11	07	03
0	EE	EE	ME	ME	ME	ME	ME	ME	ME	ME	ME	ME
50	46	42	38	34	30	26	22	18	14	10	06	02
1	EE	EE	ME	ME	ME	ME	ME	ME	ME	ME	ME	ME
49	45	41	37	33	29	25	21	17	13	09	05	01
0	EE	EE	ME	ME	ME	ME	ME	ME	ME	ME	ME	ME
48	44	40	36	32	28	24	20	16	12	08	04	00

TAG 2

44:06 The high order 9 bits of the **EXPONENT**.

38:39 The fractional portion of the **MANTISSA**.

MANTISSA is the magnitude of the number before scaling.

EXPONENT is the power of eight to which the **MANTISSA** is scaled.

The actual value of the **EXPONENT** field is:

```
EXPONENT:= 0
         & MSP.[05:44:06]
         & LSP.[14:47:09]
         ;
```

The maximum absolute value for a **DOUBLE** operand is:
1.9488293820502807912 4469 * (10 ** 29603)
The minimum absolute value, except for zero, is:
1.9385458571375858335564 * (10 ** -28581)

UNINITIALIZED OPERANDS

TAG	UNINITIALIZED OPERANDS											
0 51	47	0 43	0 39	0 35	0 31	0 27	0 23	0 19	0 15	0 11	0 07	0 03
1 50	0 46	0 42	0 38	0 34	0 30	0 26	0 22	0 18	0 14	0 10	0 06	0 02
1 49	0 45	0 41	0 37	0 33	0 29	0 25	0 21	0 17	0 13	0 09	0 05	0 01
0 48	0 44	0 40	0 36	0 32	0 28	0 24	0 20	0 16	0 12	0 08	0 04	0 00

TAG 6

47:48 All bits 0

Can be interpreted by the hardware as a 48 bit vector.

Also used by the compilers for initial setting for some variables, most notably ARRAY REFERENCEs and POINTERs.

This operand is primarily used as a software control word. bit 47 is on

TAG 4 WORDS

TAG	TAG 4 WORD											
0 51	47	0 43	0 39	0 35	0 31	0 27	0 23	0 19	0 15	0 11	0 07	0 03
1 50	0 46	0 42	0 38	0 34	0 30	0 26	0 22	0 18	0 14	0 10	0 06	0 02
0 49	0 45	0 41	0 37	0 33	0 29	0 25	0 21	0 17	0 13	0 09	0 05	0 01
0 48	0 44	0 40	0 36	0 32	0 28	0 24	0 20	0 16	0 12	0 08	0 04	0 00

TAG 4

47:48 All bits 0

This word is used for various software needs:
(i.e. **ON ANYFAULT, ON RESTART,** and **INTERRUPT**).

This word is reserved for applications in future levels of architecture.

LANGUAGE EQUIVALENTS

ALGOL	COBOL68	COBOL74	TYPE	TAG
REAL	COMP-4	REAL (*)	Single Prec	0
INTEGER	COMP-1	N/A	Single Prec	0
INTEGER	COMP	BINARY	Single Prec	0
BOOLEAN	N/A	N/A	Single Prec	0
DOUBLE	COMP-5	DOUBLE (*)	Double Prec	2
EVENT	EVENT	EVENT	Double Prec	2 ← ?
POINTER	N/A	N/A	Uninit Var	6
ARRAY REFERENCE	N/A	N/A	Uninit Var	6

Note: Starred (*) COBOL74 items are new as of Release 3.6.

PROGRAM CODE WORDS

Variable length operator sequences are stored in arrays of Program Code Words.

Contains six 8-bit syllables, numbered zero to five (0 - 5).

TAG	PROGRAM CODE WORDS											
0 51	0 47	0 43	1 39	1 35	2 31	2 27	3 23	3 19	4 15	4 11	5 07	5 03
0 50	0 46	0 42	1 38	1 34	2 30	2 26	3 22	3 18	4 14	4 10	5 06	5 02
1 49	0 45	0 41	1 37	1 33	2 29	2 25	3 21	3 17	4 13	4 09	5 05	5 01
1 48	0 44	0 40	1 36	1 32	2 28	2 24	3 20	3 16	4 12	4 08	5 06	5 00

TAG 3

47:08 Code syllable 0

39:08 Code syllable 1

31:08 Code syllable 2

23:08 Code syllable 3

15:08 Code syllable 4

07:08 Code syllable 5

MEMORY MANAGEMENT

OVERVIEW

Virtual memory management technique that is designed to satisfy memory needs of a program quickly and efficiently.

Memory is organized into two major components:

IN-USE AREAS
AVAILABLE AREAS

Each memory area is preceded and followed by link words which describe the contents and status of that area.

A DESCRIPTOR references the portion of memory containing ~~that contains~~ the data or code.

IN-USE AREAS

These areas have two general types:
- SAVE
- NON-SAVE

SAVE

SAVE ares are required to be resident in memory from time they are allocated to the time they are de-allocated.

The reasons for SAVE areas are varied:
- I/O buffer.
- DIRECT arrays. — IO buffer for direct IO
- Critical MCP or application code.
- Critical MCP structures:
  - TASK
  - STACK
  - FILE INFORMATION BLOCK (FIB)
- Critical application structures.

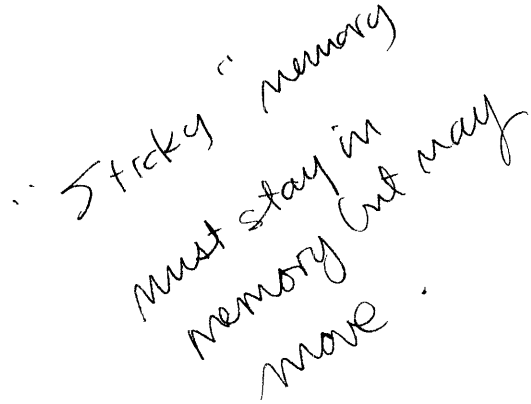
Most SAVE areas must remain at their assigned memory location for extended periods of time.

NON-SAVE OVERLAYABLE

Most NON-SAVE memory ares can be overlayed to disk by the MCP.

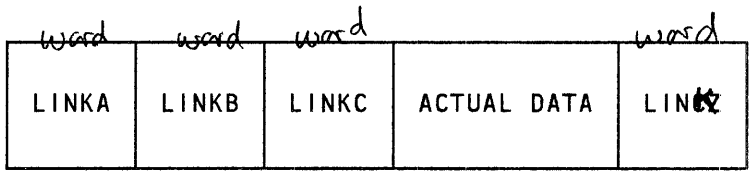
These areas are automatically re-allocated to memory when they are needed.

Some NON-SAVE memory areas must be resident in memory but are allowed to move -- considered to be "sticky" memory. (RESIDENT areas: DCALGOL).



MEMORY LINKS

for INUSE area



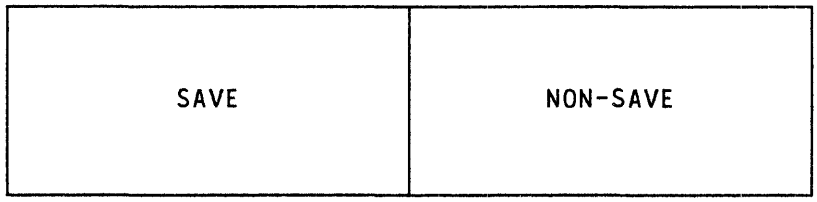
LINKA	Contains address of original descriptor (**MOM**). Contains length of area. in words Contains type of area (**ODDBALLF**).
LINKB	Contains the **STACK** number who owns the **MOM**.
LINKC	Code location where data was allocated.
ACTUAL DATA	Will vary in length for most occurrances.
LINKZ	Similar to **LINKA** for backwards memory chaining.

MEMORY ORGANIZATION

Indiscriminate mixing of **SAVE** and **OVERLAYABLE** areas would cause inefficient use of memory space such as in the case of checkerboarding.

Clustering of **SAVE** and **NON-SAVE** areas helps avoid the checkerboarding problem.

MEMORY ORGANIZATION



AVAILABLE AREAS

2 links in front
2 in back

Organized in two lists to attempt to allocate **SAVE** ares into low memory and **NON-SAVE** into high memory.

SAVE LIST

Is a list of all available memory areas that could be used as **SAVE** memory.

NON--SAVE LIST

Is a list of all available memory areas that could be used as **NON-SAVE** memory.

AVAILABLE LIST STRUCTURES

The **SAVE** and **NON-SAVE** lists are ordered by size where the smallest piece of memory is first.

If there are duplicate areas with the same size (i.e. three areas of 25 words), "side lists" are linked into the primary area in the list.

These "side lists" help reduce the number of duplicate entries the **MCP** has to scan through when looking for memory.

MCP/AS uses a **MOD** function to jump to a location in the list and then proceed from forward, thereby skipping areas of memory it knows to be too small.

DATA DESCRIPTORS

The structures that reference actual memory areas (bounded by memory links).

ORIGINAL DESCRIPTORS

There is only one original descriptor (**MOM**) for an area of memory.

Update access to **MOM** descriptors is restricted since pertinent information would be lost if they were modified.

MOM descriptors control allocation, de-allocation, and overlay of memory segments.

COPY DESCRIPTORS

Primarily is a duplicate descriptor.

May point to a memory segment (same as **MOM**) if resident.

May point to the **MOM** descriptor (if not resident: **ABSENT**).

Built by the compiler when request to access an **ORIGINAL** data segment in a different **SIZE** (**WORDS, EBCDIC, HEX**).

Built where processor operators require updates to a descriptor.

Note: A discussion of **ASD** memory will be included later.

OVERLAY

Memory requests for an area size which can not be satisfied from the available list cause the system to overlay memory which is non-save, in-use areas.

Each task is assigned an overlay file.

Each **MOM** descriptor has space reserved in this file the first time it is made present.

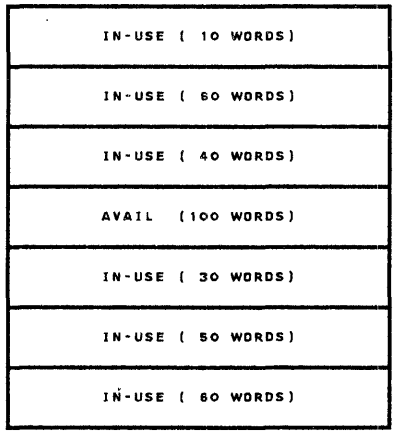
Overlay space is retained until memory segment is deallocated.

DEMAND OVERLAY

Requests for forced overlay occur only when there is not enough contiguous memory available for the size of the request.

From a rotating point in memory (called the **LEFTOFF** pointer), in-use areas are moved or overlayed until there is enough memory to satisfy the requested size.

Movement of data requires search and update of **MOM** and **COPY** descriptors.



If the request was for 300 words of memory:

> All of the above **IN-USE** areas will be moved or overlayed to satisfy the request.
>
> The last area will be split, the remainder added to the appropriate **AVAILABLE** list.

Majority of applications are written in **COBOL**.

COBOL programs tend to be coded with many small data segments (average area may be as small as 60 words).

Large data requests could cause a significant memory management burden.

WORKING SET

WORKING SET is defined as the amount of physical memory required to run a task effectively.

WSSHERIFF

Internal **MCP** process that forces overlays to occur at a predefined rate.

This approach is to prevent bursts of overlays by causing areas not used often to be overlayed.

Increased processor overhead to support **WORKING SET** memory management.

Overlays may occur more often than in **DEMAND** overlay.

WSSHERIFF is initiated when **OVERLAY GOAL** is greater than 0.

MEMORY CONTROLS

NAME	FACTOR
OVERLAY GOAL	1
AVAILMIN	2
FACTOR	3
MEMORY PRIORITY	4

OVERLAY GOAL

Percent of **OVERLAYABLE** memory that is attempted to be overlayed per minute.

AVAILMIN

Active programs are suspended in priority order if the actual **AVAILABLE** memory falls below the **AVAILMIN** percentage of actual memory.

The attempt is to slow down the requests for memory and **OVERLAY** the **OVERLAYABLE** areas of the suspended task.

SAVE areas are not **OVERLAYED.**

FACTOR

This is the percentage of actual memory the scheduling mechanism assumes it has.

Lower than 100% may cause more tasks to become scheduled with potentially less memory management overhead.

Higher than 100% may cause tasks to be executed when they normally should have been scheduled.

Higher **FACTOR** settings may increase overhead to support memory management.

MEMORY PRIORITY

Attempt to add priority as a criteria to the **OVERLAY** decision.

Lower priority tasks may tend to have more **OVERLAY** activity under **WORKING SET** than higher priority tasks.

Time to find overlay areas to meet a given size may be longer under **MEMORY PRIORITY** control.

SWAPPER

This facility allows **TIME-SLICING** to service users in a resource sharing environment.

A portion of memory is reserved for **SWAPPER** to manage.

The bounds for all data segments for a given task are contiguous.

Tasks are overlayed to a file call **SYSTEM/SWAPDISK** when it no longer requires the processor or has exceeded a pre- determined time slice.

A few IO's will occur to overlay or make a program active reducing the number of IO's required to make independent data segments resident.

Programs will conform to the normal **DEMAND** overlay scheme if not enough memory is available within is limited scope.

Tasks requiring little processor utilization are favored.

SWAPPER is not available on **ASD** machines **(MCP/AS)**.

DESCRIPTORS

DATA SEGMENT DESCRIPTORS

Data descriptor is the word type that describes data segments.

A virtual data segment is an array of elements where an element of the array is a single word, a double word pair, or a sub-word character requiring 4 or 8 bits.

TAG — DATA DESCRIPTOR FORMAT

TAG													
0	PR	RD	L	L	L	L	L	A	A	A	A	A	
51	47	43	39	35	31	27	23	19	15	11	07	03	
1	C	SZ	L	L	L	L	L	A	A	A	A	A	
50	46	42	38	34	30	26	22	18	14	10	06	02	
0	0	SZ	L	L	L	L	L	A	A	A	A	A	
49	45	41	37	33	29	25	21	17	13	09	05	01	
1	PG	SZ	L	L	L	L	L	A	A	A	A	A	
48	44	40	36	32	28	24	20	16	12	08	04	00	

TAG 5

47:01 Present bit (0 = absent, 1 = present).

46:01 Copy bit (0 = original, 1 = copy). MOM

45:01 Indexed bit (0 = un-indexed, 1 = indexed).

44:01 Paged bit (0 = non-pages, 1 = paged).

43:01 Read-only bit (0 = read/write, 1 = read-only).

42:03 The type of array element (SIZEF).

0 = Single Precision
1 = Double Precision
2 = Hex
4 = EBCDIC — 3 was 6-bit? BCL

39:20 The number of elements in the array (LENGTHF).

19:20 If the descriptor is:

Present MOM: Address in memory of the array
Absent MOM: See alternate description of 19:20
Present Copy: Address in memory of the array
Absent Copy: Address in memory of MOM

For Absent MOMs, there is a second description of bits 19:20 on the next page.

ABSENT MOM DESCRIPTORS

For non-present (i.e. absent) **MOM** Descriptors, the following is the specification for bits [19:20]:

IF
19:01 (OLAYFILEF) is 1, the data has been overlayed and bits 18:19 are the relative record number in the program's overlay file which contains the actual data
ELSE
IF
18:01 (CODEFILEF) is 1, this is an original descriptor for read-only and code segments and bits 17:18 are the relative record number in the code file of the code segment or read- only array data
ELSE
this is a "virgin" **MOM** and bits 17:18 must be evaluated as follows:

If [17:01] = 0, then bits [16:17] indicate:

0 = **NON-SAVE** array
1 = **SAVE** array
2 = **EVENT** array
>2 = **ARRAY** information table (AIT)

If [17:01] = 1, then bits [16:17] indicate:

0 = **DIRECT** array
1 = **DOPE** vector
>2 = **OWN** array information table (OAT)

LANGUAGE REPRESENTATION ORIGINAL DESCRIPTOR

ALGOL	COBOL68	COBOL74
REAL ARRAY	01 ITEM COMP.	01 ITEM BINARY.
	01 ITEM COMP-1.	N/A
EBCDIC ARRAY	01 ITEM DISPLAY.	01 ITEM DISPLAY.
HEX ARRAY	01 ITEM COMP-2.	N/A

COPY DESCRIPTOR

ALGOL

```
EBCDIC ARRAY                    EA[0:179];
HEX ARRAY              HA[0] = EA;
ARRAY                   A[0] = EA;


EA
     TAG                   5
     COPY BIT              0
     ELEMENT_SIZE          4  (EBCDIC)
     LENGTH              180
     ADDRESS               0  (Absent MOM never used)


HA
     TAG                   5
     COPY BIT              1
     ELEMENT_SIZE          2  (HEX)
     LENGTH              360
     ADDRESS              EA  (COPY points to MOM)


 A   TAG                   5
     COPY BIT              1
     ELEMENT_SIZE          0  (WORD array)
     LENGTH               30
     ADDRESS              EA  (COPY points to MOM)
```

COBOL74

```
01 EA.
     03 ITEM-2                    PIC X(002).
     03 HA            COMP        PIC X(003).
     03 ITEM-4                    PIC X(002).
     03 A             BINARY      PIC 9(011).
     03 ITEM-6                    PIC X(168).
```

```
EA
     TAG                    5
     COPY BIT               0  (MOM descriptor)
     ELEMENT_SIZE           4  (EBCDIC)
     LENGTH               180
     ADDRESS                0  (MOM absent never used)
```

```
HA
     TAG                    5
     COPY BIT               1
     ELEMENT_SIZE           2  (HEX)
     LENGTH               360
     ADDRESS               EA  (COPY points to MOM)
```

```
 A
     TAG                    5
     COPY BIT               1
     ELEMENT_SIZE           0
     LENGTH                30
     ADDRESS               EA  (COPY points to MOM)
```

ONE--QUESTION PROBE

In the example on the prior page, each of the descriptors describe the entire record. However, the individual items are much smaller than that. For example, the data item HA starts at the third EBCDIC character in the record and is three Hex characters in length, yet the descriptor for HA indicates a length of 360 Hex characters. How does the system know what piece(s) of the record to use and what to ignore?

MULTI-DIMENSIONAL ARRAYS

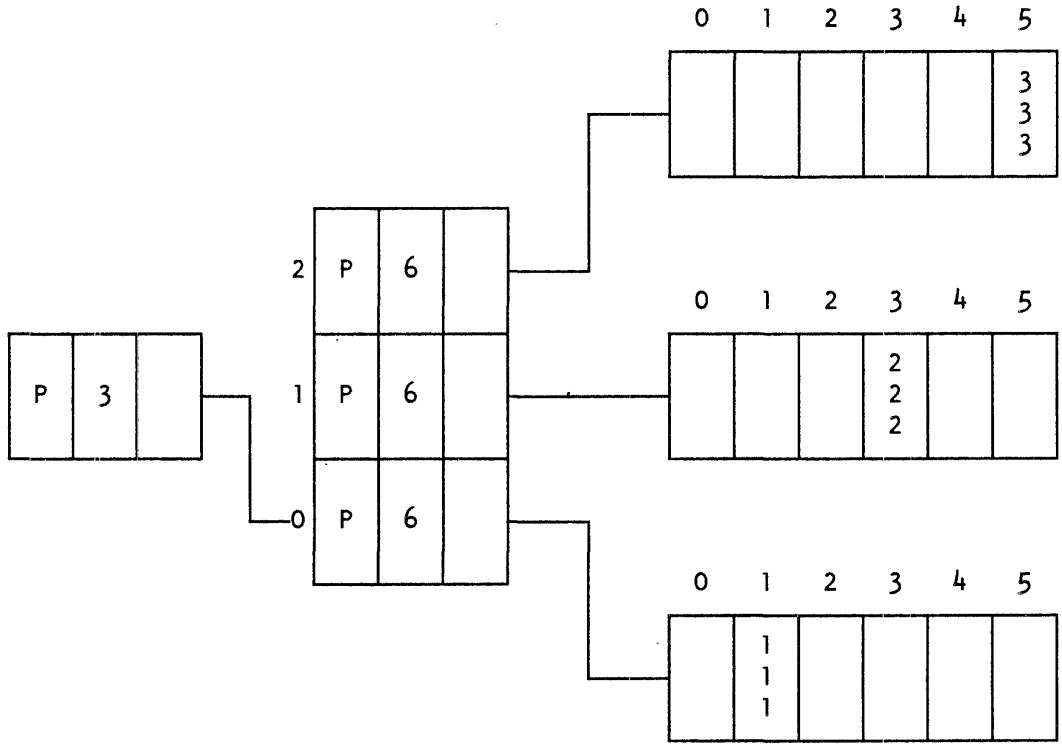
The **ORIGINAL** descriptor (sometimes called a **DOPE**) will point to a vector of **ORIGINAL** descriptors (sometimes called a **DOPE** vector).

This will continue until the last vector has **ORIGINAL** descriptors which actually point to the data segment(s).

Each of these vectors of **ORIGINAL** descriptors will be **SAVE** memory and will have the length of the next dimension.

MULTI-DIMENSIONAL ARRAY DIAGRAM

```
ARRAY A [0:2,0:5];
A [0,1] := 111;
A [1,3] := 222;
A [2,5] := 333;
```



PAGED ARRAYS

PAGED arrays (sometimes called **SEGMENTED**) divide large data segments into multiple data segments.

The structure is the same as a **MULTI-DIMENSIONAL** array except that the **PAGED** bit = **1**.

The size of the individual data segments is **256** words.

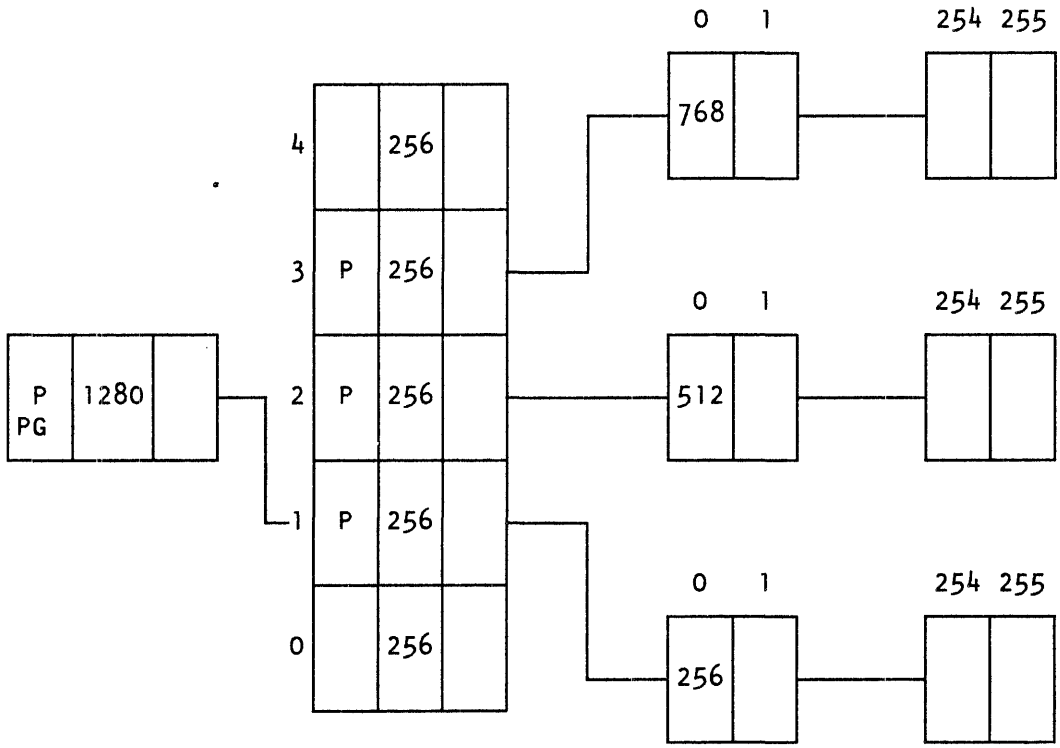
Access to elements of the **PAGED** array is transparent to the program.

Automatic segmentation occurs by default if the array is not declared **LONG** and is greater than **1024** words.

The upper bound which causes truncation can be changed by an **ODT** command.

PAGED ARRAY DIAGRAM

```
ARRAY A [0:1279];
A [256] := 256;
A [512] := 512;
A [768] := 768;
```



INDEXED DATA DESCRIPTORS

Reference an individual element of a data segment.

Must be a **COPY**.

Can not be **PAGED**.

INDEXED WORD DATA DESCRIPTOR

created by index OP

TAG	INDEXED WORD DATA DESCRIPTOR											
0 51	PR 47	RO 43	I 39	I 35	I 31	I 27	I 23	A 19	A 15	A 11	A 07	A 03
1 50	1 46	SZ 42	I 38	I 34	I 30	I 26	I 22	A 18	A 14	A 10	A 06	A 02
0 49	1 45	SZ 41	I 37	I 33	I 29	I 25	I 21	A 17	A 13	A 09	A 05	A 01
1 48	0 44	SZ 40	I 36	I 32	I 28	I 24	I 20	A 16	A 12	A 08	A 04	A 00

TAG 5

47:01 Present bit (0 = absent, 1 = present).

46:01 Copy bit (1 = COPY)

45:01 Indexed bit (1 = indexed).

44:01 Paged bit (0 = non-pages)

43:01 Read-only bit (0 = read/write, 1 = read-only).

42:03 The type of array element (SIZEF).

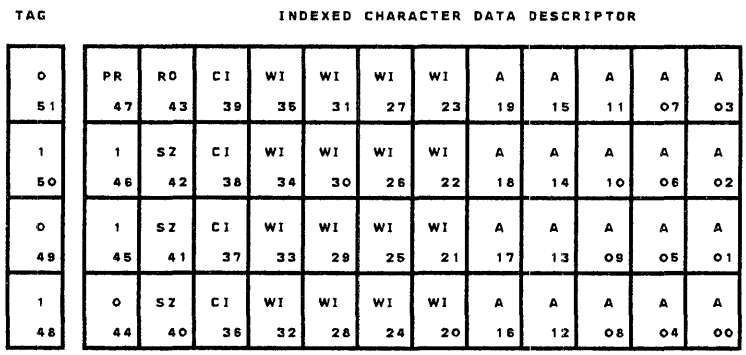
0 = Single Precision INDEXED SINGLE DD.
1 = Double Precision INDEXED DOUBLE DD.

30:20 Index (was displacement count) before index OP

19:20 Address

for char array (16 bits) it is [word index (12 bits) | offset to word (4 bits)]

INDEXED CHARACTER DESCRIPTOR



TAG	INDEXED CHARACTER DATA DESCRIPTOR											
0 51	PR 47	RO 43	CI 39	WI 35	WI 31	WI 27	WI 23	A 19	A 15	A 11	A 07	A 03
1 50	1 46	SZ 42	CI 38	WI 34	WI 30	WI 26	WI 22	A 18	A 14	A 10	A 06	A 02
0 49	1 45	SZ 41	CI 37	WI 33	WI 29	WI 25	WI 21	A 17	A 13	A 09	A 05	A 01
1 48	0 44	SZ 40	CI 36	WI 32	WI 28	WI 24	WI 20	A 16	A 12	A 08	A 04	A 00

TAG 5

47:01 Present bit (0 = absent, 1 = present).

46:01 Copy bit (1 = COPY).

45:01 Indexed bit (1 = indexed).

44:01 Paged bit (0 = non-pages).

43:01 Read-only bit (0 = read/write, 1 = read-only).

42:03 The type of array element (SIZEF).

2 = HEX
4 = EBCDIC

39:04 Character Index in SIZEF units.

35:16 WORD index (regardless of SIZEF).

19:20 Address

LANGUAGE REPRESENTATIONS

```
ARRAY A [0:29];
```

ALGOL	COBOL68	COBOL74
POINTER	N/A	N/A
POINTER(A,0)	N/A	N/A
POINTER(A,4)	N/A	N/A
POINTER(A)	N/A	N/A
POINTER(A,8)	N/A	N/A

CODE SEGMENT DESCRIPTOR

TAG	CODE SEGMENT DESCRIPTOR											
0 51	PR 47		0 38	0 35	SL 31	SL 27	SL 23	A 19	A 15	A 11	A 07	A 03
[illegible] 50	C 46	SZ 42	0 38	0 34	SL 30	SL 26	SL 22	A 18	A 14	A 10	A 06	A 02
1[illegible] 49		SZ 41	0 37	0 33	SL 29	SL 25	SL 21	A 17	A 13	A 09	A 05	A 01
1 48	0 44	SZ 40	0 36	SL 32	SL 28	SL 24	SL 20	A 16	A 12	A 08	A 04	A 00

TAG 3

47:01 Present bit (0 = absent, 1 = present).

46:01 Copy bit (0 = original, 1 = copy).

42:03 SIZEF must be zero (i.e. WORDs).

39:07 Must be zero

32:13 The number of code words in the segment.

19:20 Present:
Memory address of the base word of the data segment.

Absent Copy:
Address of the original descriptor.

If the descriptor is an Absent MOM, bit [19:01] will be zero and bits [18:19] take on new meaning:

18:01 If 1, the code segment has never been touched;
if 0, the code segment has been touched.

17:18 Address of the code segment in the code file.

SOFTWARE CONTROL WORD -- SCW

A BLOCK (in ALGOL) is defined as a **BEGIN...END** pair with declarations.

The **SCW** is used by the system to terminate a **BLOCK** which may have present **ORIGINAL** descriptors or other structures which reference entities outside of this stack.

BLOCKEXIT is the **MCP** software procedure which returns the data segments to the appropriate available memory list upon attempt to **EXIT** a **BLOCK.**

The **SCW** has a mask which designates the type of memory areas which may need to be returned.

TAG | SOFTWARE CONTROL WORD (SCW)

0 51	1 47	0 43	0 39	0 35	0 31	0 27	0 23	M 19	M 15	M 11	PC 07	PC 03
1 50	0 46	0 42	0 38	0 34	0 30	0 26	0 22	M 18	M 14	0 10	PC 06	PC 02
1 49	0 45	0 41	0 37	0 33	0 29	0 25	0 21	M 17	M 13	PC 09	PC 05	PC 01
0 48	0 44	0 40	0 36	0 32	0 28	0 24	0 20	M 16	M 12	PC 08	PC 04	PC 00

TAG 6

47:01 Bit = 1.

19:09 Mask field defining element types in BLOCK.

17:01 Non local GO TO.

16:01 DIRECT array.

15:01 FAULT (e.g. ON ANYFAULT).

14:01 INTERRUPT.

13:01 FILE.

12:01 Multi-dimensional array(s).

11:01 Single-dimensional array(s).

09:10 Count of dependent processes which have this block as its CRITICAL BLOCK.

ACTUAL SEGMENT DESCRIPTORS

ACTUAL SEGMENT DESCRIPTOR MEMORY

ASD memory is part of the MCP/AS (Advanced System) product.

Available only on Burroughs A Series computers.

A3 and A15 require hardware modifications to run MCP/AS.

A9 and A10 do not require hardware modifications to run MCP/AS.

Advantages:

- Expands monolithic memory size to 4 GW (2**32 = 24 g-bytes).
  - SUBSYSTEM concept goes away.
  - SWAPPER goes away.
- Reduces stack searching.
  - All descriptors point to the ASD table.
- Reduced MCP and user complexity.
- Addition of new resource-saving features.

Disadvantages:

- Additional reference for each memory access.
- Hardware upgrade on some machines.

Major benefit will be to users with memory problems.

ASD TABLE

Table size (i.e. number of entries) settable at Cold Start time and via the new ODT command ASD.

Each entry in the ASD table is four words long and consists of pieces of information about memory such as:

- o whether the array is present in memory or not
- o its addresses, both actual and virtual
- o its length
- o whether it's been changed (the "dirty" bit)
- o stack number of MOM descriptor
- o if it's a dope vector or not

ASD VERSUS NON-ASD DESCRIPTORS

Descriptors in the non-ASD environment are of only one type: Actual.

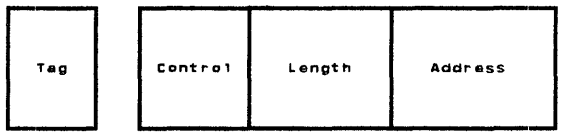
In an ASD environment, there are two types of descriptors:

Virtual (what used to be Actual on non-ASD)

Actual (the four-word ASD table entry)

Non-ASD Actual descriptors take the following basic form:

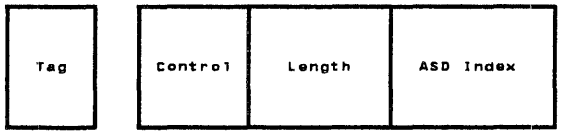
Non-ASD Actual Descriptor



[51:04] Tag of 5
[47:08] Control
[39:20] Length
[19:20] Address

ASD Virtual descriptors take the following basic form:

ASD Virtual Descriptor

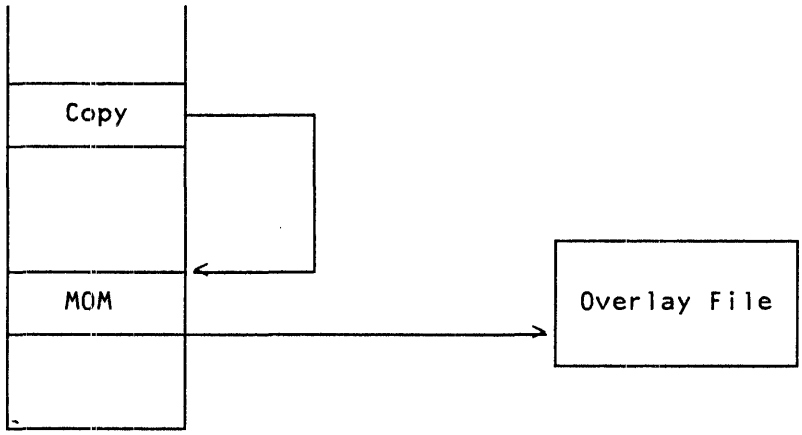


[51:04] Tag of 5
[47:08] Control
[39:20] Length
[19:20] ASD Index

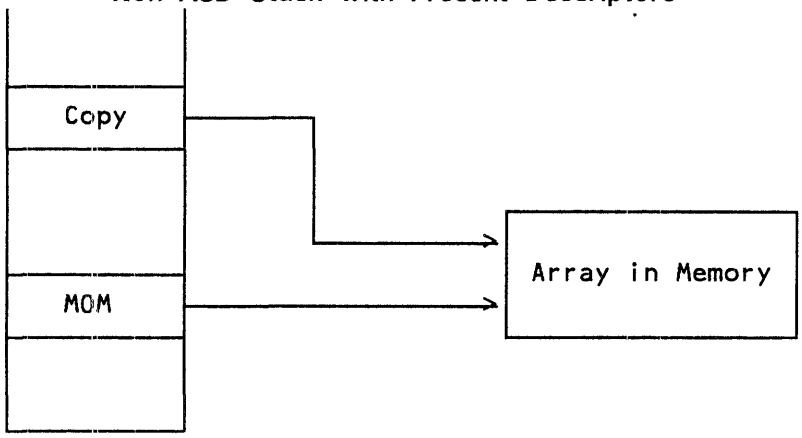
NON-ASD DESCRIPTORS

Non-ASD Descriptors are handled as follows:

Non-ASD Stack with Non-Present Descriptors



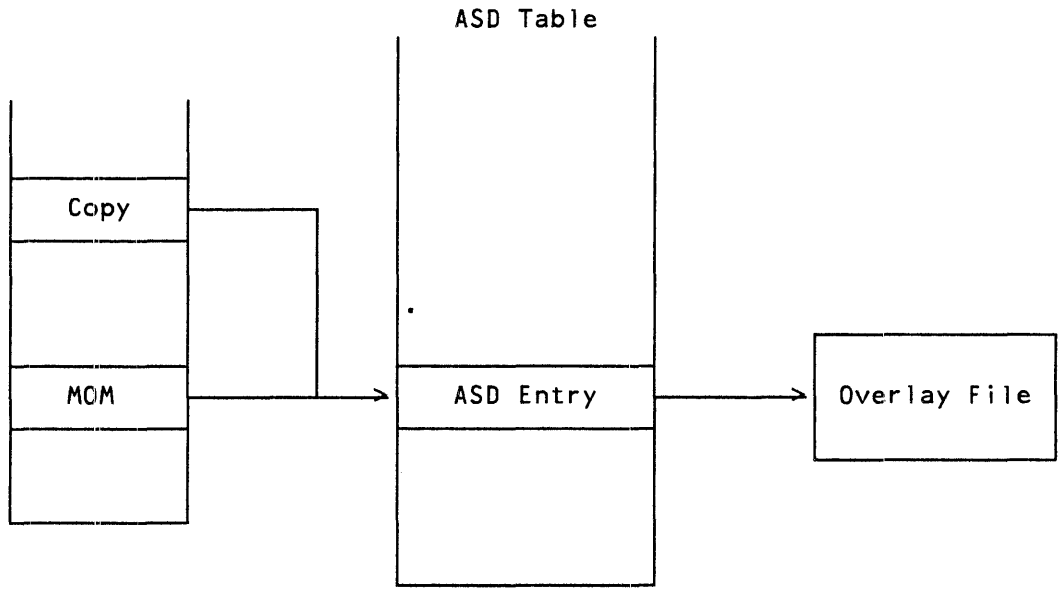
Non-ASD Stack with Present Descriptors



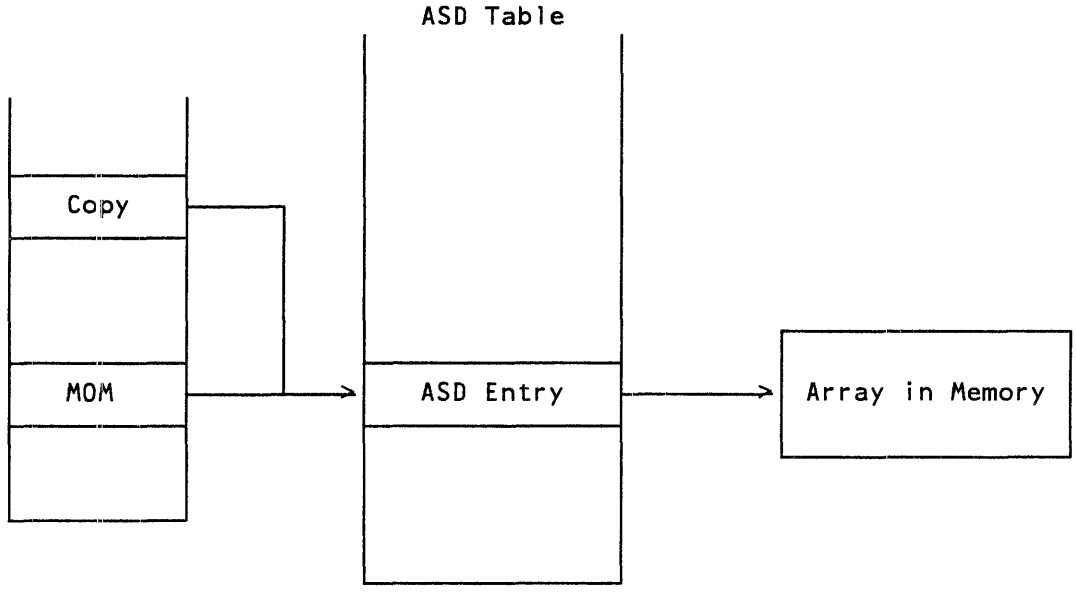
ASD DESCRIPTORS

ASD Descriptors are handled as follows:

ASD Stack with Non-Present Descriptors



ASD Stack with Present Descriptors



REFERENCE WORDS

ADDRESS COUPLES

A pair of indices (Lex Level, Offset) that reference a word in the current addressing environment.

Lex Level represents a LEXICAL region which is in the current addressing environment.

Offset is a the number of words from an Activation Record in the current addressing environment.

Examples:

(2,4)
(4,22)

Address Couples are generated in one of two forms:

FIXED FENCED

The Lex level and Offset are fixed as to their maximum ranges.

VARIABLE FENCE

The Lex level and Offset ranges depend on the current Lex Level at the time of the execution of the reference.

INDIRECT REFERENCE WORD -- IRW

NON A-series

TAG — INDIRECT REFERENCE WORD (IRW)

0 51	47	43	39	35	31	27	23	19	15	A 11	A 07	A 03
0 50	0 46	42	38	34	30	26	22	18	14	A 10	A 06	A 02
0 49	45	41	37	33	29	25	21	17	A 13	A 09	A 05	A 01
1 48	44	40	36	32	28	24	20	16	A 12	A 08	A 04	A 00

46:01 Bit = 0

13:14 Lex level and Offset fields (variable fence)

The least significant bit of the Lex Level starts at bit 13 and continues to the right up to the variable fence.

The least significant bit of the Offset starts at bit 0 and continues to the left up to the variable fence.

But where's the fence?

Number of bits valid in LEX level depends on current Lex Level at the time of the execution of the reference.

FLOATING FENCE

Problem:

One field [13:14] that contains two subfields: Lex Level and Offset.

Both fields can vary in size.

How does one determine where the "fence" (or dividing line) is?

Solution:

Take the currently-running Lex Level (not what is being referenced).

Determine how many bits are needed to represent that Lex Level.

That's how big the lex level field is.

The remaining bits are for the Offset.

Table for Determining Where the "Fence" Is

LL	LEX BITS	OFFSET BITS	MAX OFFSET
0	1	13	8191
1	1	13	8191
2	2	12	4095
3	2	12	4095
4	3	11	2047
5	3	11	2047
6	3	11	2047
7	3	11	2047
8	4	10	1023
9	4	10	1023
10	4	10	1023
11	4	10	1023
12	4	10	1023
13	4	10	1023
14	4	10	1023
15	4	10	1023

ONE--QUESTION PROBE

What problems or limitations are inherent in the above "floating fence" concept?

IRW EXAMPLE

TAG — INDIRECT REFERENCE WORD (IRW)

EXAMPLE

0 51	47	43	39	35	31	27	23	19	15	0 11	0 07	0 03
0 50	0 46	42	38	34	30	26	22	18	14	0 10	0 06	1 02
0 49	45	41	37	33	29	25	21	17	0 13	0 09	0 05	0 01
1 48	44	40	36	32	28	24	20	16	1 12	0 08	0 04	1 00
1									1	0	0	5

Assume LL = 3.

The above IRW 1005 is actually a reference to (2,5).

NORMAL INDIRECT REFERENCE WORD -- NIRW

Valid only on **E-MODE** machines.

Fixed fence.

LEX level range 0-15.

INDEX range always 0-1023, without regard to Lex Level.

Easier to read in dumps, too.

NORMAL INDIRECT REFERENCE WORD (NIRW)

TAG												
0									LL	I	I	I
51	47	43	39	35	31	27	23	19	15	11	07	03
0								0	LL	I	I	I
50	46	42	38	34	30	26	22	18	14	10	06	02
0									LL	I	I	I
49	45	41	37	33	29	25	21	17	13	09	05	01
1									LL	I	I	I
48	44	40	36	32	28	24	20	16	12	08	04	00

18:01 Bit = 0 denotes **NIRW**.

15:04 Fixed **LEX** level field.

11:12 Fixed **INDEX** field.

NIRW EXAMPLE

TAG NORMAL INDIRECT REFERENCE WORD (NIRW)

EXAMPLE

0										0	0	0
51	47	43	39	35	31	27	23	19	15	11	07	03
0								0		0	0	1
50	46	42	38	34	30	26	22	18	14	10	06	02
0									1	0	0	0
49	45	41	37	33	29	25	21	17	13	09	05	01
1									0	0	0	1
48	44	40	36	32	28	24	20	16	12	08	04	00
1									2	0	0	5

The above **NIRW 2005** is actually a reference to **(2,5)**.

E-MODE machines convert **IRWs** to **NIRWs**.

STUFFED INDIRECT REFERENCE WORD -- SIRW

Like a **NIRW** because it references a location in an addressing environment.

Points to the same location regardless of the state of the current lexical addressing environment.

Can reference anywhere in memory.

Can reference an item even if it's not within the scope of the stack's addressing environment.

E-MODE SIRW

TAG — STUFFED INDIRECT REFERENCE WORD (SIRW)

0	SN	SN	SN	D	D	D	D			OF	OF	OF
51	47	43	39	35	31	27	23	19	15	11	07	03
0	SN	SN	SN	D	D	D	D	1		OF	OF	OF
50	46	42	38	34	30	26	22	18	14	10	06	02
0	SN	SN	SN	D	D	D	D			OF	OF	OF
49	45	41	37	33	29	25	21	17	13	09	05	01
1	SN	SN	SN	D	D	D	D		OF	OF	OF	OF
48	44	40	36	32	28	24	20	16	12	08	04	00

47:12 The Stack Number of the **STACK** containing the referenced location.

35:16 The **DISPLACEMENT** from the base of the stack to the base of the Activation Record.

18:1 Bit = 1 denotes SIRW.

12:13 The **OFFSET** from the base of the activation record to the referenced location.

NON E-MODE SIRW

45:10 The Stack Number of the STACK containing the referenced location.

46:01 Bit = 1 denotes SIRW.

SIRW EXAMPLE

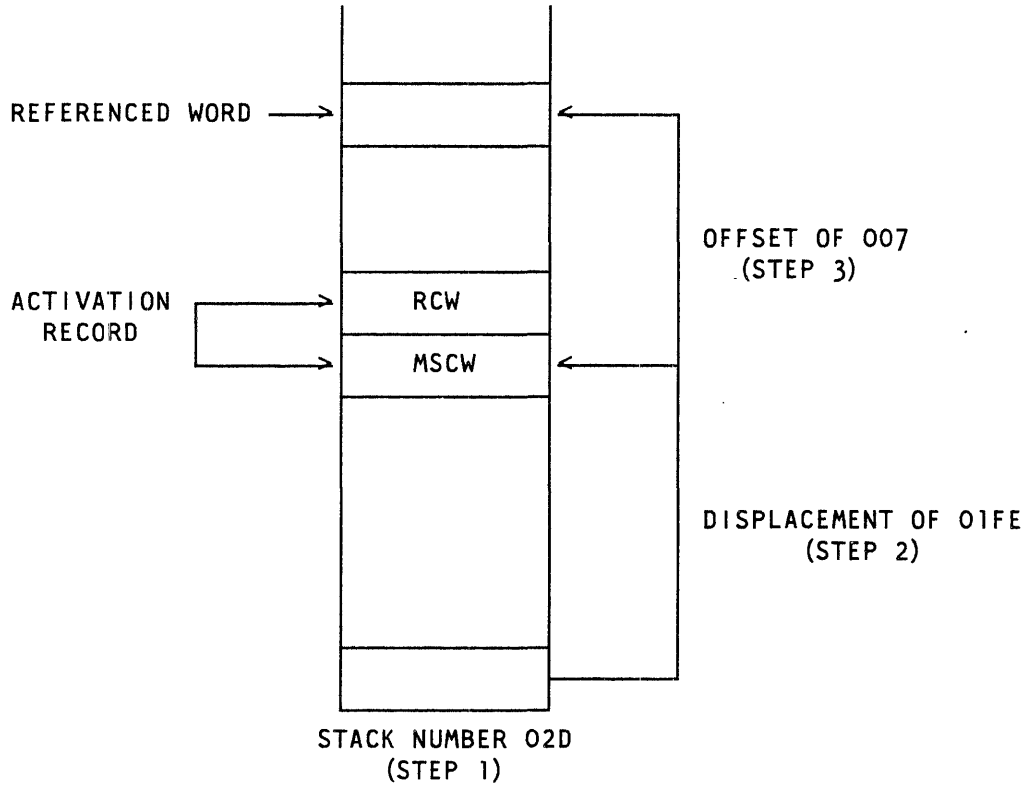
Given that an SIRW contains the following information:

47:12 Stack Number 02D.

35:16 Displacement of 01FE.

12:13 Offset of 007.

The following diagram illustrates the SIRW mechanism:



PROGRAM CONTROL WORD -- PCW

Points to a code syllable in a code segment.

Normally used for procedure entry.

E-MODE

TAG	PROGRAM CONTROL WORD (PCW)											
0	SN	SN	SN	PSI	PWI	PWI	PWI	CS	LL	SDI	SDI	SDI
51	47	43	39	35	31	27	23	19	15	11	07	03
1	SN	SN	SN	PSI	PWI	PWI	PWI	0	LL	SDI	SDI	SDI
50	46	42	38	34	30	26	22	18	14	10	06	02
1	SN	SN	SN	PSI	PWI	PWI	PWI	LL	SDLL	SDI	SDI	SDI
49	45	41	37	33	29	25	21	17	13	09	05	01
1	SN	SN	SN	PWI	PWI	PWI	PWI	LL	SDI	SDI	SDI	SDI
48	44	40	36	32	28	24	20	16	12	08	04	00

TAG 7

47:12 SNR value when MPCW (Make PCW) is executed.

35:03 The syllable index into the code word.

PROGRAM SYLLABLE INDEX (PSI).

32:13 The word index into the code segment.

PROGRAM WORD INDEX (PWI).

19:01 CONTROL STATE (1=CONTROL state, 0=NORMAL state)

CONTROL STATE : no external interrupts.
NORMAL STATE : external interrupts allowed.

newp declaration

18:01 Must be zero.

17:04 The Lexical Level for the new activation record.

13:01 The SEGMENT DICTIONARY LEX LEVEL (SDLL).

D[0]: Bit = 0 (MCP code segments).
D[1]: Bit = 1 (USER code segment).

12:13 The SEGMENT DICTIONARY INDEX (SDI).
Code segment descriptor is at address D[SDLL] + SDI or as an address couple: (SDLL,SDI).

NON E-MODE

45:10 SNR value when MPCW executed.

18:05 The lex level for the new activation record.

BASIC STACK ARCHITECTURE

STACK

SAVE data segment.

Maintains a "last in first out" (LIFO) structure.

Contains program data or references to data (data could be code).

Defines program environments.

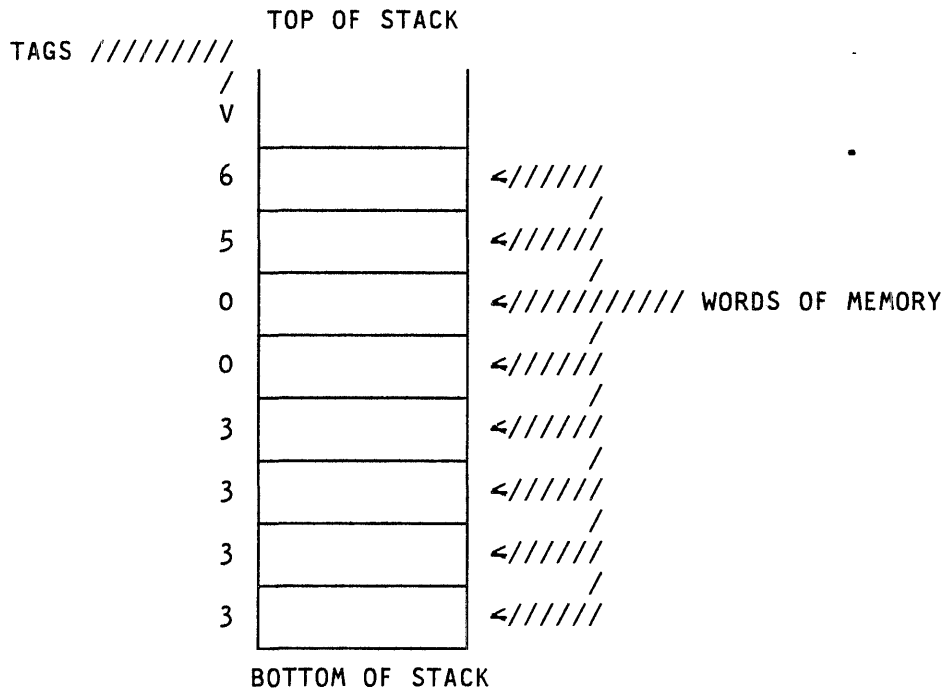
Physical top of STACK (LOSR [Limit of Stack Register]).

Logical top of STACK (S Register).

There are multiple PSEUDO STACK structures.

FILE INFORMATION BLOCKS.

PROCESS INFORMATION BLOCKS.



PROCESS STACK

Each execution of a code file has its own **PROCESS** stack.

Can be indirectly shared by other **PROCESS** stacks.

Data unique to this run of the program is kept in the Process Stack.

EXPRESSION STACK is the last active portion of the **STACK** where expressions are evaluated.

Commonly (and erroneously) called the **D[2]** stack.

At any instant in time, shows the exact state of a program, including:

- all procedures that have been entered
- the value(s) of all variables

CODE SEGMENT DICTIONARY

Executable code is contained in segments defined by descriptors located in the **CODE SEGMENT DICTIONARY**, sometimes called the **SEGMENT DICTIONARY**.

May contain descriptors to **READ-ONLY** arrays.

May contain **SINGLE-** or **DOUBLE-**precision operands.

Items directly or indirectly in the **CODE SEGMENT DICTIONARY** are never modified by an application program.

Each application has at least one Segment Dictionary.

A segment of code may only be referenced by one **CODE SEGMENT DICTIONARY**.

Multiple **PROCESS** stacks can share **CODE SEGMENT DICTIONARIES** and its associated code.

STATEMENTS, COMPOUND STATEMENTS, AND BLOCKS

STATEMENT

A **STATEMENT** is a simple statement without a **BEGIN...END** pair surrounding it.

```
X := 3                    READ (INFILE,14,INBUF)
```

COMPOUND STATEMENT

Zero of more **STATEMENT**s bounded by a **BEGIN...END** pair and having no declarations.

```
BEGIN          BEGIN          BEGIN
END            X := 3;        Y := 2;
               END            THRU 10 DO;
                              END
```

BLOCK

A **COMPOUND STATEMENT** with declarations.

```
BEGIN          BEGIN          BEGIN
REAL X;        REAL A,B,C;    DEFINE NEWSEGMENT = #;
END            A := 2;        END
               B := 3;
               END
```

Every block gets its own code segment.

Every block gets its own Segment Descriptor in the Segment Dictionary.

These facts are noted in the compile listing with both an Address Couple for the Segment Descriptor and in the Code File Address.

INTERNAL PROCESSOR REGISTERS

Each processor contains one set of special purpose registers.

Some registers contain memory addresses for program environments (**ENVIRONMENT** registers).

Other registers pertinent to execution and efficient indirect access to data.

Most of these registers contain a physical memory address which is use as a base for indirect access to data or data segment references.

Maintained as an array of registers.

ENVIRONMENT REGISTERS

Addressing environments of the executing code stream consists of a set of local addressing spaces contained within stacks.

Referred to as **ACTIVATION RECORDS, LEXICAL REGIONS,** or **DISPLAY REGISTERS.**

Each region is given an environment number.

The environments range from **0** to **15** (older hardware can go higher).

These environments are designated as **D0 - D15** or **D[0] - D[15].**

The current addressing environments is a linked list of **LEXICAL REGIONS.**

NON A-Series

D0

Memory address of the most global **ACTIVATION RECORD** (MCP stack).

D1

Primarily the memory address of the **CODE SEGMENT DICTIONARY ACTIVATION** record.

May point to **MCP LEXICAL** regions.

May address other pseudo **STACK** structures as **LEXICAL** regions.

FILE INFORMATION BLOCK.

PROGRAM INFORMATION BLOCK.

D2

Primarily memory address of applications outer block **LEXICAL** region.

May also point to **MCP** nested **LEXICAL** regions.

D3 -- D15

The current hardware supports these **LEXICAL** regions.

Older hardware may address environments up to **D[31]**.

Current hardware reference these Environments indirectly through **ACTIVATION** record chaining.

on A Series only seen CURRENT

A, B, X, AND Y REGISTERS

The A and B registers are considered to be the two "top of stack registers".

Whenever two operands are operated on, the values, or their addresses must be in the two top of stack registers.

The X and Y registers are considered extensions of the A and B registers, respectively.

> For example, if a double variable is put on the top of the stack, the MSP may go in the A or B register and the LSP would go in the respective extension register. If the MSP was put in the A register, the LSP would be put in the X register.

AROFF AND BROFF REGISTERS

The AROFF and BROFF registers indicate whether the A and B registers contain valid data, respectively.

When information is placed in a register, the appropriate xROFF register is set. When the data in the register is invalidated, such as by a STOD operator, the xROFF register is reset.

LL REGISTER

The lexical level of the topmost **ACTIVATION** record in the current addressing environment.

DLL REGISTER

A series

The memory address of the base of the topmost **ACTIVATION** record.

Head of the lexical chain.

F REGISTER

Defines the head of the historical chain (the most current activation record).

S REGISTER

Memory address of the last valid item in the topmost **ACTIVATION** record.

Logical top of stack.

BOTTOM OF STACK REGISTER -- BOSR

Memory address of the base of the **STACK**.

LIMIT OF STACK REGISTER -- LOSR

Memory address of the bottom element in the stack as defined by its entry in the **STACK VECTOR ARRAY**.

STACK NUMBER REGISTER -- SNR

Index into the STACK VECTOR ARRAY.

Used by the hardware to index into the STACK VECTOR ARRAY.

PROGRAM SYLLABLE INDEX -- PSI

Current code syllable pointer.

PROGRAM WORD INDEX -- PWI

Current word index of the active segment descriptor.

SEGMENT DICTIONARY INDEX -- SDI

Index into the segment dictionary of the code's descriptor.

PROGRAM BASE REGISTER -- PBR

Memory address of the first word of the current code segment.

MCP STACK

Contains data, indirect data references and code segment references for the MCP.

Outer BLOCK of the MCP.

Sometimes referred to as the D [0] stack.

This environment is visible to all STACKs present in the system.

The highest in the hierarchy of data addressing environments.

Contains some physical structures for the hardware.

The D[0] register points near the base of the MCP stack.

STACK VECTOR ARRAY

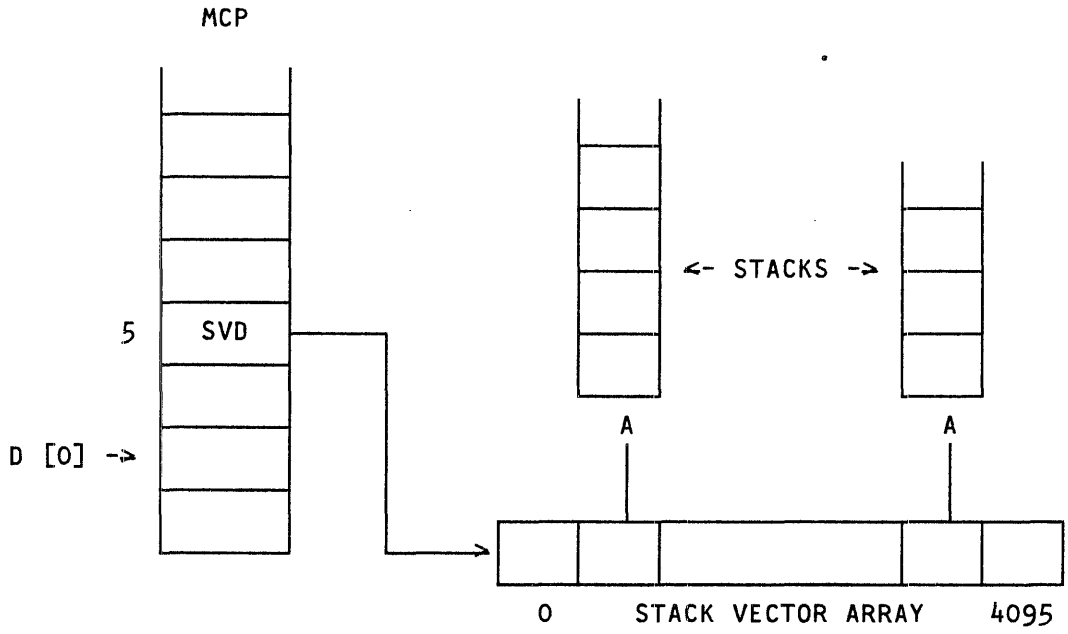
Vector (array) of active and available **STACK** entries in the system.

Each word in the Stack Vector may be a data descriptor which may point to a stack.

Maximum number of stacks depends on the total memory available (current maximum 4096).

The **STACK VECTOR DESCRIPTOR (SVD)** is declared in the **MCP** addressing environment.

SVD is located at **D [0] + 2** (i.e. Address Couple (0,2)).



PROCEDURE ENTRY

Entering a procedure is a four-step process that the compiler must generate code for:

```
MKST       (Insert a Mark Stack Control Word)
NAMC       (Generate an IRW to a PCW)
<optional parameter placement>
ENTR       (Actually enter the procedure)
```

We will omit the discussion of parameter passing until later.

Before executing the ENTR operator, the top of the stack has:

An incomplete MSCW.
An IRW to the PCW of the procedure to be entered.

When the ENTR operator is executed:

The IRW is turned into an RCW.
The MSCW is completed and marked as "entered".
D Registers are updated as per the PCW.
The processor starts executing where the PCW dictated.

PROCEDURE EXIT

A procedure is exited when the EXIT or RETN operator is executed. (The RETN operator will be discussed later.)

When an EXIT operator is executed:

The S register is set to F - 1.
The F register is set to point to the prior MSCW by following the History Link in the current MSCW.
The RCW is evaluated to determine, among other things, where to start executing code.
D registers are updated by following the Lex Level Links from the new MSCW.

COBOL PROCEDURE ENTRY AND EXIT

As COBOL is not a block structured language, it does not handle procedure entry and exit (i.e. PERFORMs) the same way that block structured languages do.

When the COBOL compiler encounters a PERFORM statement, it determines where the exit point is for the PERFORM.

A "serial number" is assigned to every possible exit point.

Entering a procedure (doing the PERFORM) consists of the compiler placing on top of the stack a PCW of where to come back to and then the "serial number" of the exit point. A "go to" is done to enter the procedure.

At every exit point, code exists to see if the serial number left on top of the stack matches the serial number for the exit point you are at.

If the "serial numbers" don't match, the program falls through to the next statement.

If the "serial numbers" do match, the PCW left on the stack is used to do a "go to" back to the PERFORM.

The concept of a BAD GO TO does not exist in COBOL.

PROGRAM EXAMPLES

```
      BEGIN
      REAL
          R                                    (2,2)
         ,R2                                   (2,3)
          ;
      INTEGER
          I                                    (2,4)
          ;
      DOUBLE
          D                                    (2,5)
          ;
      ARRAY
          AR [0:9]                             (2,6)
          ;
      DOUBLE ARRAY
          DA [0:9]                             (2,7)
          ;
      EBCDIC ARRAY
          EA [0:9]                             (2,9)
          ;
      HEX ARRAY
          HA [0:9]                             (2,A)
          ;
      POINTER
          P_UNINITIALIZED                      (2,B)
         ,P_INITIALIZED                        (2,C)
          ;
      ARRAY REFERENCE
          AR_REF_UNINITIALIZED                 (2,D)
          ;

      P_INITIALIZED := EA [9];

------>

      END OF PROGRAM.
```

```
        BEGIN
        REAL
            R                                          (2,2)
           ;
        ARRAY
            A [0:9]                                    (2,3)
           ;
        EBCDIC ARRAY
            AR [0] = A                                 (2,4)
           ;
        POINTER
            GP                                         (2,5)
           ;
        PROCEDURE P;                                   (2,6)
            BEGIN
            REAL
                R2                                     (3,2)
               ;
            ARRAY
                A2 [0:9]                               (3,3)
               ;
            POINTER
                PA                                     (3,4)
               ,PA2                                    (3,5)
               ;

            PA := POINTER (A  [2]) + 3;
            PA2:= POINTER (A2 [2]) + 3;

    --------->

            END;
        P;

        END.
..endfixed
```

```
BEGIN
REAL
    R                                          (2,2)
   ;
ARRAY
    A [0:9]                                    (2,3)
   ;
EBCDIC ARRAY
    AR [0] = A                                 (2,4)
   ;
POINTER
    GP                                         (2,5)
   ;
PROCEDURE P;                                   (2,6)
    BEGIN
    REAL
        R2                                     (3,2)
       ;
    ARRAY
        A2 [0:9]                               (3,3)
       ;
    POINTER
        PA                                     (3,4)
       ,PA2                                    (3,5)
       ;
    PROCEDURE  P2;                             (3,6)
        BEGIN
        ARRAY      A3 [0:9];                   (4,2)
        POINTER    PA3;                        (4,3)

        PA := POINTER (A [2])  + 3;
        PA2:= POINTER (A2 [2]) + 3;
        PA3:= POINTER (A3 [2]) + 3;

------------->

        END;
    P2;
    END;
P;
END OF PROGRAM.
```

```
      BEGIN
      REAL
          R                                          (2,2)
          ;
      ARRAY
          A [0:9]                                    (2,3)
          ;
      EBCDIC ARRAY
          AR [0] = A                                 (2,4)
          ;
      POINTER
          GP                                         (2,5)
          ;
      PROCEDURE GPPROC;                              (2,6)
          BEGIN
          REAL R2;                                   (3,2)

--------->

          END;
      PROCEDURE P;                                   (2,7)
          BEGIN

          REAL
              R2                                     (3,2)
              ;
          ARRAY
              A2 [0:9]                               (3,3)
              ;
          POINTER
              PA                                     (3,4)
             ,PA2                                    (3,5)
              ;
          PROCEDURE P2;                              (3,6)
              BEGIN
              ARRAY     A3 [0:9];                    (4,2)
              POINTER   PA3;                         (4,3)

              PA := POINTER (A  [2])  + 3;
              PA2:= POINTER (A2 [2]) + 3;
              PA3:= POINTER (A3 [2]) + 3;
              GPPROC;

              END;
          R2 := 75;
          P2;
          END;
      R := 50;
      P;
      END OF PROGRAM.
```

(B) (A) return point

STACK LINKAGE WORDS

MARK STACK CONTROL WORD -- MSCW

To place information in the stack as to where prior MSCWs are as well as what "D" register should point here.

E-MODE

A-Series

TAG	MARK STACK CONTROL WORD (MSCW)											
0	SN	SN	SN	D	D	D	D	RS	LL	HL	HL	HL
51	47	43	39	35	31	27	23	19	15	11	07	03
0	SN	SN	SN	D	D	D	D	E	LL	HL	HL	HL
50	46	42	38	34	30	26	22	18	14	10	06	02
1	SN	SN	SN	D	D	D	D	LL	HL	HL	HL	HL
49	45	41	37	33	29	25	21	17	13	09	05	01
1	SN	SN	SN	D	D	D	D	LL	HL	HL	HL	HL
48	44	40	36	32	28	24	20	16	12	08	04	00

TAG 3

47:12 STACK of previous Lex Level's Activation Record.

35:16 The DISPLACEMENT field.

18:01 The ENTERED bit (0 = not entered, 1 = entered).

17:04 The LEX Level which this activation record runs.

13:14 The HISTORY between this MSCW and the prior MSCW.

Bits 47:12 and 35:16 are collectively called the "Lex Level Linkage". They point to the prior Lex Level's Activation Record. For example, if we're running at Lex Level 4, the Lex Level Linkage would point to the Activation Record for Lex Level 3. This may or may not be in the same stack.

Bits 13:14 are called the "History Linkage". They point to the prior Activation Record's MSCW in this stack without regard to Lex Levels.

displacement
back within stack

NON E-MODE

TAG MARK STACK CONTROL WORD

0	DS	SN	SN	D	D	D	D	RS	LL	HL	HL	HL
51	47	43	39	35	31	27	23	19	15	11	07	03
0	E	SN	SN	D	D	D	D	LL	LL	HL	HL	HL
50	46	42	38	34	30	26	22	18	14	10	06	02
1	SN	SN	SN	D	D	D	D	LL	HL	HL	HL	HL
49	45	41	37	33	29	25	21	17	13	09	05	01
1	SN	SN	SN	D	D	D	D	LL	HL	HL	HL	HL
48	44	40	36	32	28	24	20	16	12	08	04	00

TAG 3

47:01 Previous Lex Level's Activation Record is in a DIFFERENT STACK.

46:01 The entered bit (0 = not entered, 1 = entered).

45:10 STACK number of prior Lex Level's Activation Record.

35:16 DISPLACEMENT from BOSR to MSCW of Prior Lex Level.

19:01 Position in operator flow to restart: 0 = restart from beginning, 1 = restart from next syllable.

18:05 The LEX level which the activation record runs.

13:14 The HISTORY LINK.

MSCW LEX LEVEL VERSUS HISTORY LINKAGES

LEX LEVEL LINKAGE

The Stack Number/Displacement fields together constitute the Lex Level Linkage.

The Lex Level Linkage identifies where the prior Lex Level's D register should point.

For example, if D [3] points at the current MSCW, the Lex Level Linkage can be used to locate the MSCW where D [2] points. This may or may not be in the same stack.

HISTORY LINKAGE

The History Link field points to the prior MSCW in this stack.

No consideration is given as to Lex Levels for the History Linkage.

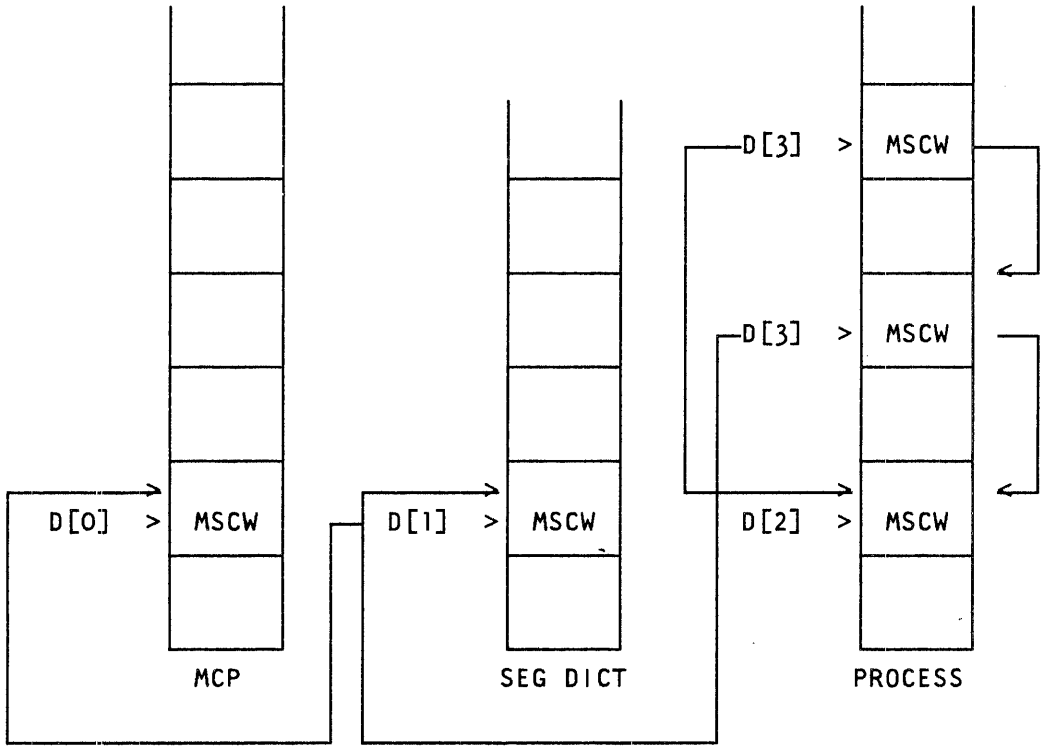
WHY THE TWO

The two linkages are mainly used for procedure exit.

The History Linkage is used to know how far to cut back the stack when the program exits a procedure.

The Lex Level Linkage is used to re-establish the addressing environment to what it was prior to entering the procedure.

LEX LEVEL VERSUS HISTORY LINK DIAGRAM



Lexical Links are shown on the left sides of the stacks.

History Links are shown on the right sides of the stacks.

RETURN CONTROL WORD -- RCW

To tell the hardware where to continue executing code when this procedure is exited.

Note that an RCW tells where to go -- not necessarilly where we came from.

E-MODE RCW

TAG	RETURN CONTROL WORD											
0	EX	RS	EO	PSI	PWI	PWI	PWI	CS	LL	SDI	SDI	SDI
51	47	43	39	35	31	27	23	19	15	11	07	03
0	OF			PSI	PWI	PWI	PWI	0	LL	SDI	SDI	SDI
50	46	42	38	34	30	26	22	18	14	10	06	02
1	TF	BE		PSI	PWI	PWI	PWI	LL	SDLL	SDI	SDI	SDI
49	45	41	37	33	29	25	21	17	13	09	05	01
1	FL	EO		PWI	PWI	PWI	PWI	LL	SDI	SDI	SDI	SDI
48	44	40	36	32	28	24	20	16	12	08	04	00

TAG 3

47:01 External sign flip-flop.

46:01 Overflow flip-flop.

45:01 True/false flip-flop.

44:01 Float flip-flop.

43:01 Restart indicator (0 = initial, 1 = restart state)

41:01 Arms BLOCKEXIT interrupt from EXIT or RETN
0 = disarmed, 1 = armed

40:02 Optimization of EXIT/RETN.

35:03 The syllable code stream pointer.
PROGRAM SYLLABLE INDEX (PSI).

32:13 Word index into the code segment. PROGRAM WORD INDEX (PWI).

19:01 CONTROL STATE
1 = CONTROL STATE: no external interrupts,
0 = NORMAL STATE: external interrupts allowed.

18:01 Must be zero.

17:04 The lexical level for the new activation record.

13:01 The SEGMENT DICTIONARY LEX LEVEL (SDLL).
0: D [0] (MCP code segment)
1: D [1] (USER code segment)

12:13 The SEGMENT DICTIONARY INDEX (SDI).
Points to the code segment descriptor.

NON E-MODE

The following bits are not valid:

43:01 RS

41:01 BE

40:02 EO

The following bits are not included:

42:01 True/false occupied flip-flop.

ACTIVATION RECORD

A combination of **MSCW** and **RCW** that is generated as a result of entering a procedure.

An Activation Record indicates things about this procedure such as:

The Lex Level (and hence what **D** register points at it) the procedure runs at.

The location of the prior Lex Level's Activation Record (the Lex Level Linkage).

The location of the prior **MSCW** in this stack (the History Linkage).

Where to go when this procedure is **EXIT**ed (provided by the **RCW**).

TOP OF STACK CONTROL WORD -- TOSCW

Goes at the bottom of a Process stack when the stack is inactive.

It allows the processor to set up some of its registers such as S and F when it goes to execute this stack.

When the Process stack is active, the processor id (a number from zero to eight) is in the bottom of the stack.

E-MODE

TOP OF STACK CONTROL WORD

TAG												
0				SH	SH	SH	SH			SF	SF	SF
51	47	43	39	35	31	27	23	19	15	11	07	03
0				SH	SH	SH	SH			SF	SF	SF
50	46	42	38	34	30	26	22	18	14	10	06	02
1				SH	SH	SH	SH		SF	SF	SF	SF
49	45	41	37	33	29	25	21	17	13	09	05	01
1				SH	SH	SH	SH		SF	SF	SF	SF
48	44	40	36	32	28	24	20	16	12	08	04	00

TAG 3

35:16 Logical Stack Height (SH). S := BOSR + SH.

13:14 S to F Displacement (SF). F := S - SF.

NON E-MODE

The following fields have been deleted:

47:01 ES

46:01 OF

45:01 TF

44:01 FL

42:01 TFOF

41:01 C:Compare flip-flop.

19:01 CS

18:05 LL

The above deleted fields can be obtained from the other STACK linkage words.

PROGRAM INITIALIZATION

BASE OF STACK

Contains the fixed portion of the stack built by the MCP.

Contains a descriptor to the program's TASKFILE.

Contains a descriptor to the array information table (AIT) used for multi-dimensioned arrays.

Contains a descriptor to the OWN information table (OIT) used for arrays declared OWN.

Dummy MSCW.

Contains overlay file information pointers.

FIRST EXECUTABLE PCW -- FEP

PCW located in the segment dictionary which represents the first syllable of execution in the program.

Normally points to the stack building code of the outer block.

Location of **FEP** is pointed to by the first record of the code file usually called **SEGMENT ZERO** or **BLOCK ZERO.**

Normally **ALGOL** location is at **(1,2).**

Floats for **COBOL** (after last variable entered in the segment dictionary).

TASK INITIALIZATION PROCESS

1. MCP searches for the code file.
2. MCP validates code file kind.
3. SEGMENT ZERO of code file is read into memory.
4. SEGMENT DICTIONARY size determined.
5. STACK number selected for SEGMENT DICTIONARY.
6. Space acquired for the SEGMENT DICTIONARY.
7. STACK VECTOR ARRAY descriptor updated.
8. Compiler-built SEGMENT DICTIONARY read from code file to memory starting at MSCW slot after fixed BASE.
9. STACK number selected for PROCESS stack.
10. Fixed BASE built.
11. Dummy MSCW, RCW built called DUMMY RUN (used as platform for execution of NORMALEOJ).
12. MSCW RCW pair built return set to execute NORMALEOJ.
13. MSCW RCW spare built to be used as user RCW exit.
14. MSCW RCW pair built for entry into NORMALBOJ.
15. Rest of stack filled with 4"BADBADBADBAD".
16. STACK inserted into queue for tasks waiting for the processor (READYQ) in priority order.
17. Stack selected by processor.
18. EXITS into NORMALBOJ.
19. NORMALBOJ performs remaining initialization tasks.
20. NORMALBOJ installs FEP into USER RCW previously built.
21. NORMALBOJ exits into user code.
22. Upon completion of user task, the processor EXITS into NORMALEOJ.
23. NORMALEOJ performs task termination duties.

PARAMETERS

PARAMETER PASSING

Parameters are a formalized way of giving selected information to and receiving selected information back from a procedure.

Parameters to a procedure appear first in the stack before any local variables.

Sequence of events for entering a procedure with parameters is:

Mark the stack with an **MSCW**

Create an **IRW** to the **PCW** location where the procedure we are entering is.

Place actual parameter(s) on top of stack

Enter procedure.

CLASSIFICATIONS OF PARAMETERS

FORMAL

Parameters which are described in the procedure header.

Treated as local variables to the procedure body.

Parameters are assigned the first local index cell(s) prior to declarations of the procedure body.

ACTUAL

Parameters actually passed to the procedure in invocation statement.

ACTUAL parameters must match FORMAL parameters in number and type.

```
BEGIN
REAL R;

PROCEDURE P (W);
  REAL W;          % FORMAL PARAMETER W
    BEGIN
    W := 52;
    END OF PROCEDURE P;

P (R);                 % ACTUAL PARAMETER R
END OF PROGRAM.
```

REFERENCE TYPES

BY VALUE

Actual expression is evaluated once prior to procedure call.

Any reference to the associated **FORMAL** parameter within the procedure body only modifies the local procedure copy and does not affect the **ACTUAL** parameter.

BY REFERENCE

A reference to the **ACTUAL** parameter is passed **(SIRW)**.

Any use of the associated **FORMAL** parameter causes interrogation or modification to the **ACTUAL** parameter passed.

BY NAME

Each use of the associated **FORMAL** parameter causes re-calculation of the **ACTUAL** reference.

Re-calculation causes a procedure entry called "accidental entry".

Sometimes called **"THUNK"**.

Expressions (i.e. **3 + X**) can be passed, but only interrogation is allowed if the result does not produce a reference.

Any attempt to store into a non reference producing **ACTUAL** parameter would cause program fault **(INVALID OP)**.

A further description will follow in a few pages.

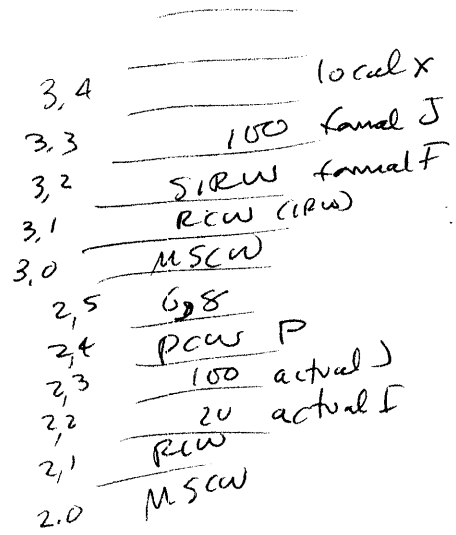
EXAMPLES

```
BEGIN
REAL
    ACTUAL_I                          (2,2)
   ,ACTUAL_J                          (2,3)
   ;

PROCEDURE P (FORMAL_I, FORMAL_J);
  VALUE                 FORMAL_J;
  REAL
      FORMAL_I                        (3,2)
     ,FORMAL_J                        (3,3)
     ;
    BEGIN
    REAL LOCAL_X;                     (3,4)
    FORMAL_I := 52;
    FORMAL_J := ACTUAL_I + 2;
    END OF PROCEDURE P;

% Main line of program.
ACTUAL_I := 20;
ACTUAL_J := 100;

P (ACTUAL_I, ACTUAL_J);
END OF PROGRAM.
```



THUNK AND ACCIDENTAL ENTRY

Passing an expression as a parameter causes the expression to be re-evaluated at every reference in the "passed-to" procedure.

How does this re-evaluation occur?

Upon detecting that an expression is being passed "by name", the compiler generates a small procedure that will do the re-evaluation.

Then, instead of passing a value or data address to the procedure, the compiler generates code which passes an **SIRW** to the **PCW** for the procedure that does the re-evaluation.

Whenever the called procedure asks for the value or address of the "by name" parameter, the hardware automatically invokes the procedure to do the re-evaluation.

This automatic procedure invocation by the hardware is where the name "accidental entry" is derived from. It is also called a "thunk".

BY-NAME PARAMETER EXAMPLE

Consider the following program example:

```
BEGIN
REAL Y;

PROCEDURE A (B);
  REAL B;
    BEGIN
    B := 3;
    Y := 2;
    B := 5;
    END OF PROCEDURE A;

REAL ARAY X [0:9];

% Main line of program.
A (X[Y]);
END OF PROGRAM.
```

What has this program just done?

TYPED PROCEDURES

Called "functions".

Is really nothing more than a procedure that returns a value.

Very similar to the Intrinsic functions such as **SIN**, **COS**, **TAN**, and **TIME**.

When entering a procedure, an extra location is placed in the stack for the "procedure value".

> The compiler notes this if **$ SET STACK** is set.
> The procedure value comes after any parameters.
> The procedure value comes before any local variables.

Instead of doing a normal **EXIT** operator to exit a procedure, the compiler will generate a **RETN** (Return) operator. This operator exits the procedure, but leaves whatever value was on top of the stack as the value of the procedure. It is the compiler's responsibility to make sure the procedure value is on top of the stack when the **RETN** operator is executed.

In fact, this is how "thunks" are done. The compiler generates a small typed procedure that returns the result of the expression that is re-evaluated at every usage.

STACK REVIEW

BASIC STACK REVIEW

Review PROGRAMDUMP for:

JSD/CONCEPTS/LL2

Find location of address couples.

Find location of SDI:PWI:PSI. (003:0000:1)

Find length of each segment.

X IS SEGMENT 00004.

X (004) LENGTH IS 8 WORDS.

Notice outer block identification is BLOCK#1.

This is because there is no explicit name like a procedure has.

PROGRAMDUMP

HEADER

MIX numbers.
Machine
BOSR location
Local BOX location.

Program name.

MCP Field Release.Patch Release.Cycle
MCP name.
INTRINSICS

SYSTEM SERIAL.
HOSTNAME.

Cause of dump.
    PROGRAM REQUESTED
    FAULT TERMINATION
    DSED TERMINATION
    SPO REQUESTED
RCW history.
    Line number (notice they are absent: no LINEINFO)
    SDI:PWI:PSI

PROGRAMDUMP options.

BODY

LOSR

LOSR - BOSR = Relative offset

Items by column:

Offset
Address couple
Environment Locations (D register settings).
TAG
Stack word
Word type (OP, RCW, SEG, CODE)
Word description

CODE

SEG DESC is the code segment descriptor (SDI) for each RCW.

CODE represents a moving window of actual code.

>01808EABDF60< represents **SDI:PWI** for code lexical region will return to.

SEGMENT dictionary **DUMP** is helpful if code that appears is not enough to analyze code history that caused the failure.

OPTION = CODE

Locate the **PWI:PSI** for each RCW.

JSD/CONCEPTS/LL3

Build the stack (at PROGRAMDUMP).

Include:

SEGMENT DICTIONARY stack.
- BASE
- Program code segments
- First Executable PCW (FEP).

PROCESS stack.
- BASE
- Stack building code for Outer Block
- Stack for local procedure X.
- S
- F
- BOSR
- LOSR
- MSCW
  - History linkage.
  - Activation record linkage.
- RCW

Review PROGRAMDUMP.

Find code syllable each RCW is pointing to.

JSD/CONCEPTS/LL4

Build the stack with the following:

Include:

SEGMENT DICTIONARY stack.
BASE
Program code segments
First Executable PCW (FEP).

PROCESS stack.
BASE
Stack building code for Outer Block
Stack for local procedures.
S
F
BOSR
LOSR
MSCW
History linkage.
Activation record linkage.
RCW

PROBE

Given the following program:

```
BEGIN
REAL VAR1,VAR2;
DOUBLE VAR3;

PROCEDURE D (D1,D2,D3);
  VALUE      D1,D2,D3;
  REAL       D1,D2,D3;
    PROGRAMDUMP;

PROCEDURE A (A1,A2);
  VALUE      A1,A2;
  REAL       A1,A2;
    BEGIN

    ARRAY X [0:10];

    PROCEDURE C (C1);
      REAL       C1;
        D (C1,VAR1,VAR2);

    PROCEDURE B (B1);
      VALUE      B1;
      REAL       B1;
        C (B1);

    B (A1);
    END;

VAR1 := 1;
VAR2 := 18;
A (VAR1,VAR2);

END OF PROGRAM.
```

Build the stacks at the point in which the program forced a program dump. Include the following:

SEGMENT DICTIONARY stack.
- **BASE**
- Program code segment descriptors
- First Executable **PCW (FEP)**

PROCESS stack.
- **BASE**
- Stack building code for Outer Block
- Stack for local procedures
- **S**
- **F**
- **BOSR**
- **LOSR**
- **MSCW**
  - History linkage.
  - Activation record linkage.
- **RCW**

DUMPANALYZER

MEMORY DUMPS

OVERVIEW

Memory dumps are a snapshot of memory.

Multi-processing is discontinued during dumping the contents of memory.

A **MEMDUMP** routine within the **MCP** executes in a portion of memory that is reserved for dumping without destroying the state of usable memory.

Media for output is usually **TAPE.**

Dump to disk is also possible.

CAUSES

Operator requested using the **DUMP ODT** command.

Operator requested using the **??DP** primitive.

System generated because of:

Potential software failures.

Potential hardware failures.

General diagnostic.

Reset **TERMINATE; MCP** run-time option (any failure such as **INVALID INDEX** or **SEG ARRAY ERROR** will cause a dump).

INTRODUCTION

DUMPANALYZER produces user-specified subsets of information from a memory dump.

Analyzes information according to parameters given or default.

MODES

INTERACTIVE

Commands are processed separately as it is entered by the user.

Can specify where output is to be sent (printer or remote).

STANDARD

Commands entered completely before processing.

Output always directed to the printer.

SAVED DUMPS

Stored for future processing without requiring original memory dump input TAPE.

Faster future processing.

EXECUTION

Normal tape input.

```
RUN *SYSTEM/DUMPANALYZER;
```

Saved dump.

```
RUN *SYSTEM/DUMPANALYZER;
  FILE TAPEIN (KIND=DISK,TITLE=MEMORY/DUMP/TITLE);
```

Select run mode: **STANDARD** or **INTERACTIVE**

Memory dump tape is processed with the following responses:

Initial options display.

Initializing.

Reading <dump file name>

Initializing stack info.

Lineinfo/names.

Enter requests...

BASIC CONSTRUCTS

Numbers default is hexadecimal.

<simple address>

<absolute>

4F3A2

<simple location>

STK 4A3

PIB A6

STK 682 + 7

RV M [47AC2]

<multiple address>

More than one memory address.

4AF3 FOR A6

3BAC TO 3BCB

STK 32 LOSR TO BOSR

<simple value>

M [4A32]

DEC 100

Partial words are available.

COMMANDS

AREAS

Prints contents of memory areas.

Areas can be searched by type or size.

AREAS SIZE 512

AREAS ODDBALL FIBMARK

BOX

Designates local box to use for some memory commands.

DC

Causes a full data communications analysis to be printed.

DEADLOCK

Causes information to be printed regarding stacks that hold locks or waiting for a lock.

FIB

Analyzes a FIB at a given address.

```
FIB AT 4A3DE
```

HDR

Causes an analysis of the disk file header stack.

HELP

Provides information about **DUMPANALYZER** commands.

```
HELP AREAS

HELP HELP

HELP <number>
```

IO

Invokes input/output analysis of all peripherals.

IOCB

Causes an I/O control block to be printed.

LINKS

Prints the address and contents of each link of a memory area.

LOCKS

Reports on the status of soft (PROCURED) locks.

MD

Dumps the contents of a group of address in memory with no analysis.

```
MD 6DE43 FOR A6
```

NAMES

Causes the entire list of MCP names and D [0] relative address.

Names appear in numeric and alphabetic sequences.

PIB

Prints the contents of a PIB.

PRINTER

Routes output to the printer instead of the terminal.

PRINTVALUE/PV

Displays the specified <simple value> in several possible forms.

QUEUE

Displays DCALGOL queues.

RELEASE

Closes the current print file.

RELX

Causes the current print file to be closed and printed (using **SYSTEM/BACKUP**).

REMOTE

Routes output to the terminal rather than the printer.

SAVE

Causes the memory dump to be stored on disk for future use.

MCP names and memory contents are retained releasing the need of having the current MCP code file and original memory dump tape.

SAVE "MEMORY/DUMP"

SEARCH

Provides the ability to check each word in memory for a specified pattern of bits.

Can search including TAG.

```
SEARCH MOMDESC RANGE 3DE9 FOR 6F
```

Command which initiates a memory search for a specific **PATTERN** optionally using a **MASK**.

MASK

Used to mask fields for searching.

PATTERN

Bit configuration to search for.

STACK

Causes the contents of a stack to be interpreted and printed.

STOP

Terminates execution.

SUMMARY

Provides the list of stacks in the machine and the status of the stacks at the time the dump was take.

WHERE

Displays the D[0] relative location of <MCP global names>.

```
WHERE BLOCKEXIT
```

Response: 00A BLOCKEXIT

WHO

Displays the MCP global name for a D[0] relative address.

```
WHO 00A
```

Response: 00A BLOCKEXIT

ASD CHANGES

DUMPANALYZER now allows the displaying of memory pointed to by an ASD entry by using the "via" option:

MD via 4425 for 20

(will print out the twenty words of memory pointed to by ASD entry 4425)

ASDNUMBER <asd #> [EXPAND]

The above command will display the ASD entry requested.

ADVANCED STACK ARCHITECTURE

SHARED GLOBAL ENVIRONMENTS

OVERVIEW

Internal procedures may be initiated as a separate stack executing in step (**SYNCHRONOUS**) or in parallel (**ASYNCHRONOUS**).

The external task(s) is(are) dependent upon the initiator (or parent).

The block which is designated as the **CRITICAL BLOCK** must not attempt to exit before its dependent children terminate. If this happens, a **CRITICAL BLOCK EXIT** error is issued.

The scope of visibility of variables remains the same if the procedure was invoked in the caller.

Global variables may be interrogated or modified by the caller and all dependent processes.

Resource protection (**LOCKs**) is the means to coordinate access to shared variables.

EXAMPLE

What will be the value of the variable I at EOT?

```
BEGIN
REAL I;
TASK TSK;

PROCEDURE P;
     BEGIN
     REAL X;
     X := 24;
     I := 52;
     END OF PROCEDURE P;

I := 22;
PROCESS P [TSK];
I := 75;

WHILE TSK.STATUS GEQ 0 DO
    WAITANDRESET (MYSELF.EXCEPTIONEVENT);

END OF PROGRAM.
```

INTRINSIC INTERFACE

OVERVIEW

Intrinsics are a set of general utility routines which can be invoked as if they where procedures declared in each programs environment.

The environmental software comes with a set of these routines which appear in the **GENERALSUPPORT** library.

The user has the option to create his own utility routine library (**INTRINSICS**).

This interface was available prior to the implementation of libraries.

LINKAGE PROCESS

1. The first call to an intrinsic (i.e SQRT) causes a slot to be reserved in the program's **SEGMENT DICTIONARY.**

```
I := SQRT(9);
                    (1,5) = SQRT
```

2. This slot contains an invalid word required for procedure entry (**PCW**).

```
5 070000 000001  <--- Intrinsic Number
```

3. The code to execute the above call to **SQRT** would be:

ALGOL:

```
SQRT (9)
```

CODE:

```
MKST
NAMC (1,5)
LT8  9 <--------- Parameter to the SQRT Intrinsic
ENTR
```

4. The **ENTR** would cause an **INVALID OPERATOR** interrupt to to occur due to the fact that it is trying to enter a data descriptor instead of a **PCW.**

Note: This type of interrupt (entering a tag 5 word with an Element Size of 7) is called a "Binding Request".

5. The hardware interrupt routine would recognize the **TAG** of 5 and the **ELEMENT** size of 7 as an **INTRINSIC** call.

6. The address field contains the numeric mnemonic representation of **SQRT.**

7. The **MCP** then changes the invalid **TAG** 5 word to a **SIRW** pointing to the location of the **PCW** in the **GENERAL SUPPORT LIBRARY** or the **INTRINSIC** stack.

8. The **INTRINSIC** stack contains data, **PCWS** and **SEGMENT** descriptors (much like the **MCP** stack).

9. The intrinsic routines run at **LEX** level 2.

MCP INTERFACE

OVERVIEW

1. Each call to an MCP procedure (i.e PROGRAMDUMP) causes a slot to be reserved in the programs SEGMENT DICTIONARY.

```
PROGRAMDUMP (ARRAYS);
                    (1,5) = MCP reference
```

2. This slot contains an invalid word required for procedure entry (PCW).

```
5 070000 FFE017
```

3. The code to execute the above call to PROGRAMDUMP would be:

```
MKST
NAMC (1,5)
<OPTIONS Parameter>
ENTR
```

4. The ENTR would cause an INVALID OPERATOR interrupt to to occur due to it finding a data descriptor instead of a PCW.

5. The hardware interrupt routine would recognize the TAG of 5, the ELEMENT size of 7 and the FFE in the length and address fields as an MCP procedure call.

6. The address field contains the mnemonic representation of PROGRAMDUMP.

7. The MCP then changes the invalid TAG 5 word to a SIRW pointing to the location of the PCW in the MCP stack.

8. The MCP stack contains data, PCWS and SEGMENT descriptors.

9. The MCP intrinsic routines run at LEX level 1 since they are actually declared at LEX level 0.

10. The ENTR operator is re-executed and the MCP procedure is invoked since the target is now a valid PCW.

11. Actually, all MCP or INTRINSIC references are resolved in NORMALBOJ.

PASSING PROCEDURES AS PARAMETERS

OVERVIEW

Procedures may also be passed as parameters to other procedures.

Procedures may be passed to:

Internal procedures that are invoked.

Internal procedures that are executed as dependent processes.

Separately compiled programs.

Activation record environment depend on the location in which the procedure passed as a parameter is **DECLARED.**

There may be multiple **D[2]** references in this complex stack linkage environment.

EXAMPLE

Main Program

```
BEGIN
REAL R;                                    (2,2)
TASK TSK;                                  (2,3)

PROCEDURE P;                               (2,4)
    PROGRAMDUMP;

PROCEDURE EXT(PROC);                       (2,5)
  PROCEDURE PROC ();                       (3,2)
    FORMAL;
  EXTERNAL;

REPLACE TSK.NAME BY "OBJECT/EXT.";
PROCESS EXT (P) [TSK];

WHILE TSK.STATUS GEQ 0 DO
    WAITANDRESET (MYSELF.EXCEPTIONEVENT);

END OF PROGRAM.
```

OBJECT/EXT

```
$ LEVEL 2

PROCEDURE EXT_PROC (P1);
  PROCEDURE P1 ();                                 (2,2)
      FORMAL;
    BEGIN
    REAL X;                                        (2,3)
    P1;
    END OF PROCEDURE EXT_PROC AND PROGRAM.
```

LIBRARY INTERFACE

OVERVIEW

This type of linkage essentially is the same as passing a procedure as a parameter.

The syntax to invoke the library procedure is different.

The caller and the library are usually separate programs.

LINKAGE PROCESS

THE CALLER

1. The compiler builds a **TEMPLATE** which describe the following:

   - o attributes of the library
   - o **TITLE** of the code file
   - o Procedures which should appear in the library
   - o The count and type of the parameters of the library routine.

2. The compiler builds an invalid **PCW** at the location where the procedure in the library is declared in the caller.

```
PROCEDURE P_IN_LIB;                    (2,5)
    LIBRARY LIB;


    5 670001 4808E9                    (2,5)
```

3. The caller invokes the procedure with the following code:

```
MKST
NAMC (2,5)
ENTR
```

4. The **ENTR** operator would cause an **INVALID OPERATOR** interrupt because the target is not a **PCW**.

5. The **MCP** hardware interrupt routine would recognize that this is a library call because:

   TAG 5
   INDEX bit on
   ELEMENT size 7

6. The **ADDRESS** field of this word contains the memory address of the actual location of the **TEMPLATE**.

7. The **LENGTH** field contains an index into the **TEMPLATE** for the procedure which we are trying to invoke.

8. An **MCP** library linkage routine is call which causes the actual library to be automatically invoked.

9. After the linkage is made the **MCP** routine inserted an **SIRW** at (2,5) which points to the actual **PCW** for the library procedure.

10. The **ENTR** operator is re-executed.

THE LIBRARY

1. The library begins executing as a normal program.

2. For each procedure referenced in the **EXPORT** declaration the compiler inserts into a library **DIRECTORY** the name of the procedure, the number and types of parameters, and an **SIRW** to the actual **PCW** for the procedure.

3. When the library program executes a **FREEZE**, an **MCP** procedure is called to make this program a **LIBRARY**.

    - o Shrinks the stack.

    - o Looks around for any task looking for this program as a library.

4. When a caller is found the library procedure which it wants to invoke is searched for in the libraries **DIRECTORY**.

5. The count and type of parameters are verified in both the callers **TEMPLATE** and the libraries **DICTIONARY**.

6. If the parameters match the libraries **SIRW** for the actual procedure is placed in the callers slot reserved for the procedure.

7. All references to procedures which the caller wants in a library are resolved at upon the first reference.

EXAMPLE

CALLER

```
BEGIN
REAL X;                                          (2,2)
LIBRARY LIB (TITLE = "OBJECT/LIB.");             (2,3)

PROCEDURE P_IN_LIB;
    LIBRARY LIB;

P_IN_LIB;

END OF PROGRAM.
```

OBJECT/LIB

```
BEGIN

PROCEDURE GLOBAL_P;                              (2,2)
    PROGRAMDUMP;

PROCEDURE P_IN_LIB;                              (2,3)
    BEGIN
    REAL R;                                      (3,2)
    GLOBAL_P;
    END OF PROCEDURE P_IN_LIB;

EXPORT P_IN_LIB;

FREEZE (TEMPORARY);

END OF PROGRAM.
```

PROGRAM INFORMATION BLOCK -- PIB

OVERVIEW

The PIB contains TASK attribute information.

Any reference to TASK attributes causes D[01] to point into the PIB structure.

The PIB is treated like a stack so that access to attributes is accomplished by normal variable addressing since the activation record pointed to by D[01] contains all of the PIB information.

FILE INFORMATION BLOCK -- FIB

OVERVIEW

The FIB contains directly or indirectly all the elements of the file structure.

Many of these elements are FILE attributes.

Referencing the FILE structure causes D [01] to point within the FIB.

The FIB is treated like a stack so that access to attributes is accomplished by normal variable addressing since the activation record pointed to by D [01] contains all of the FIB information.

CODE FILE CONSTRUCTION

INTRODUCTION

The following subjects will be reviewed in appendix B "Code File Construction" and with program listings.

Code File Layout

Segment Zero

Segment Dictionary

File Parameter Blocks (FPBs)

Compiler/Intrinsic Interface

Binding

Lineinfo

OPERATING SYSTEM INTERFACE

CODE FILE LAYOUT

Review the text in the appendix.

Demonstrate actual **SEGMENT ZERO** in dumpall listing of:

CODESTRUCTURE/TEST/STACK

SEGMENT ZERO

Review the text in the appendix.

Build the following of the **SEGMENT DICTIONARY.**

Fixed portion of the stack **(BASE).**

Actually perform **READWITHTAGS.**

First location is the **MSCW.**

Use the dumpall listing of:

CODESTRUCTURE/TEST/STACK

Build the **SEGMENT DICTIONARY** from the dumpall list.

SEGMENT DICTIONARY

Once you have built the **SEGMENT DICTIONARY**, relate all stack items to address couples in the compile listing.

PROBE

Using the dumpall listing:

CODESTRUCTURE/TEST/STACK

1. Build the complete **SEGMENT DICTIONARY** including the **BASE** and analyze each of the fields of each of the words.
2. Locate and determine the value of the fields of the **FEP**.
3. Find the code syllable pointed to by the **FEP**.
4. Find the syllable pointed to by **PCWS** given to you by the instructor.

FILE PARAMETER BLOCKS -- FPB

Review the text of the appendix.

Review the **DUMPALL** listing.

Records 1 and 2 (0 relative) are **FPBS.**

COMPILER/INTRINSIC INTERFACE

Compiler places invalid descriptors (SIZEF = 7) in the Segment Dictionary.

MCP changes them to SIRWs to the procedure's actual PCW.

BINDING

OVERVIEW

The process of merging separately-compiled procedures into a main host creating a single executing unit.

BINDING also allows inter-language procedures to be combined and invoked.

The **BINDER** is a compiler who is responsible for merging of the separate code pieces into one.

This facility was developed before libraries were implemen- ted.

Many sites still use the binder.

The **MCP** is built with separate inter-language modules which is bound to the **MCP** host producing a running **MCP** code file.

GLOBAL DECLARATIONS FOR BINDER

The languages provide the capability to reference GLOBAL variables when compiling a separate unit.

ALGOL

```
[

 REAL GLOBAL_R1;
 PROCEDURE GLOBAL_PROCEDURE;
 FILE GLOBAL_FILE;

]
```

COBOL

```
01 GLOBAL_RECORD GLOBAL.
     03 GLOBAL-ELEM PICTURE X(100).
```

EXTERNAL DECLARATIONS FOR BINDER

The syntax for declaring an external procedure which is to be bound is:

ALGOL

```
BEGIN

PROCEDURE BOUNDPROC;
    EXTERNAL;  % Note that the body of the
               % procedure is replaced by the
               % word EXTERNAL.

BOUNDPROC;     % A normal invocation.

END;
```

COBOL74

```
DECLARATIVES.

BOUNDPROC.
     USE AS EXTERNAL PROCEDURE.

END DECLARATIVES.

PERFORM BOUNDPROC.
```

BINDER SYNTAX

The syntax for binding includes:

```
HOST IS OBJECT/HOST;
BIND BOUNDPROC FROM OBJECT/BOUNDPROC;
```

The binder must know about the names and address couples in both the host and the separate routine to resolve the differences.

If the **BINDER** does not find a reference in the **HOST** that was declared **GLOBAL** in the routine to be bound, the **BINDER** will add this item to the **HOSTS** outer block declarations.

Internal procedures may be replaced using the **BINDER**.

The resultant code file has the **FILEKIND** of **BOUNDCODE**.

BOUNDCODE files may be used as **HOSTs**.

BINDER CONVENTIONS

See Appendix B, Code File Information, of the Student Appendix.

For **BINDER** to be able to do its job, it must be able to find out about all of a program's declarations.

These declarations are encoded and kept in the code file (unless the compiler dollar option **NOBINDINFO** is set).

UNIVERSAL CLASS

Is used to determine the rough category of tokens in the code file.

This could be things such as **INFORMATION KEYWORD, DATA DESCRIPTOR, PCW, FILE, TASK,** etc.

SUB CLASS

Within a given Universal Class, what more specifically is being discussed.

For example, for a Universal Class of 2 [two] (48-bit Operand), some of the possible Sub Classes recognized are:

0 Single Precision Operand (INTEGER)
1 Single Precision Operand (REAL)
2 Single Precision Operand (BOOLEAN)
3 WORD Variable
4 REFERENCE Variable

PROGRAM DESCRIPTION

A structure in the code file that defines all procedures in the program and all external references, such as intrinsics and global variables.

PROCEDURE DIRECTORY

A structure in the code file that lists all of the procedures in a program, regardless of their lex level.

EXTERNAL DIRECTORY

A directory of what items are outside of this program, such as intrinsics.

LOCAL DIRECTORY

For each procedure, there is a Local Directory enumerating the items declared within that procedure.

These items could be **ARRAY**s, **REAL**s, **INTEGER**s, and even other **PROCEDURE**s.

PRINTBINDINFO

Review example in the Printed Listings.

Documented in the System Software Support Manual.

Mainly, reads a code file and generates its declarations, without any executable code.

FPB/PPB RUN TIME

Review the File Parameter Block and Program Parameter Blocks in the code file.

LINEINFO

This information relates **SDI:PWI:PSI** locations to program sequence numbers.

Used to when analyzing **RCWs** when producing a **PROGRAM DUMP.**

Controlled by the following compiler dollar option:

```
$ SET LINEINFO
```

Defaults to **SET** when compiling under **CANDE.**

LINE DICTIONARY

This item is the **READONLY** descriptor referenced by the **SDI 1 (1,1)**.

Each word of the **LINE DICTIONARY** contains the relative record in the code file of the beginning of the sequence information.

The **LINE DICTIONARY** is indexed by the **SDI** of the code pointer we are trying to find the sequence number for.

SDI:0	SDI:1	SDI:2	SDI:3
000000000000	000000000000	000000000000	000000000002

Record 2 of the code file contains the sequence number information for **SDI 3**.

FORMAT OF SEQUENCE RECORDS

Word 0:

[39:20] Total character count of entire sequence number record starting at word 1.

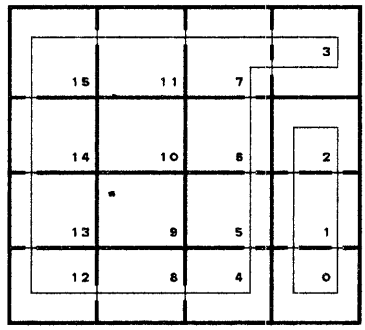
[19:20] Back link to previous sequence information for the same SDI.

Word 1:

2 bytes PWI:PSI for the beginning of the code for the following sequence number.

[15:12] PWI

[02:03] PSI



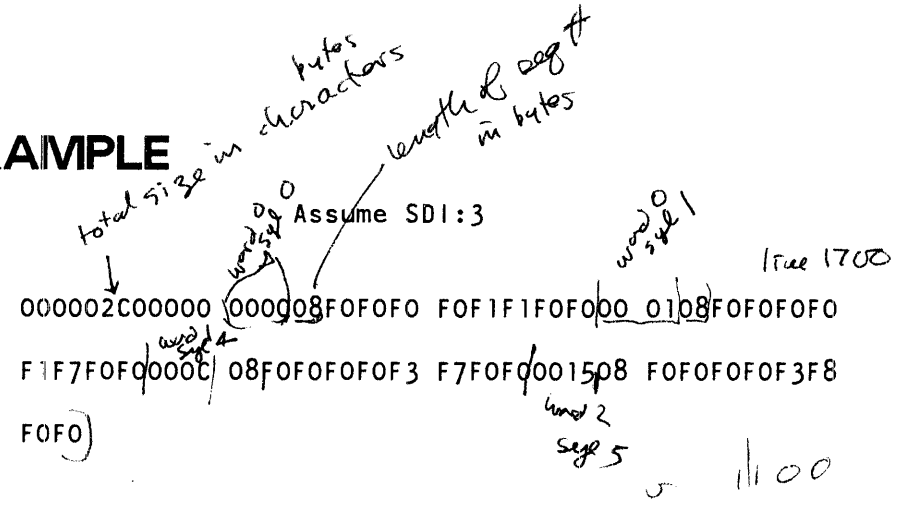
1 byte Binary length of the sequence number.

SSSSSS Actual sequence number.

Word 2-N:

Word 1 format continues until end of sequence information.

EXAMPLE



Total length 2C (44) characters.

3:0:0 starts sequence 1100

3:0:1 starts sequence 1700

3:1:4 starts sequence 3700

3:2:5 starts sequence 3800

If an RCW contains 3:1:2, what line would it reference?

PROBE

Using dumpall listing for:

CODESTRUCTURE/LINEDICTIONARY

1. Build **SEGMENT DICTIONARY.**
2. Find **FEP.**
3. Find syllable pointed to by **FEP.**
4. FInd line numbers for instructor given **SDI:PWI:PSI** sequences.
5. Build **3** items of the **EXTERNAL** directory.
6. Locate **1** procedure directory record.
7. Locate and process **5** items from a **LOCAL** directory.
8. Identify if there is any run time **FPB/PPB** information.

MACHINE OPERATOR SET

REVERSE POLISH NOTATION

OVERVIEW

Arithmetic or logical notational system using only operands and operators arranged in sequence or strings.

Eliminates the need for intermediate storage for multiple computations.

Well suited to stack architecture.

Example:

A + B + C

Reverse Polish Notation

AB+C+ ABC ++ A + (B + C)

Polish Notation

++ABC

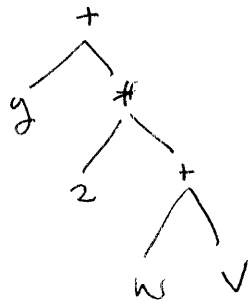
EXAMPLES

Expression	Reverse Polish String
7 * (B + C)	7 B C + *
Y + 2 * (W + V)	Y 2 W V + * +
A AND B AND NOT C	A B AND C NOT AND

7 BC + *

Y 2 W V + * +

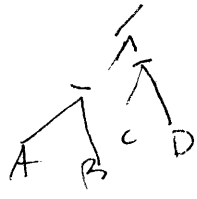
A B and C not and



A / B - C

AB/C-

(A - B) / (C - D)



AB - CD - /

REFERENCE GENERATION OPERATORS

NAMC 40-7F

Name call operator transforms an address-couple in the code stream into:

NON-EMODE: an IRW

E-MODE: an NIRW

The address couple given in the code stream is variable fence.

Examples:

5002	NAMC (2,2)
7003	NAMC (3,3)
4017	NAMC (0,17)
6005	NAMC (1,5)

LNMC 958C

Long name call is equivalent to **NAMC** except that its parameter is a fixed fence.

STFF AF

The stuff operator converts an **IRW** or an **NIRW** into an **SIRW**.

INDX A6

The index operator applies an integer index to an un-indexed data descriptor and leaves on top of the stack an Indexed Data Descriptor to the specified element.

If the data descriptor is a word data descriptor, the result is an Indexed Word Data Descriptor.

If the data descriptor is a Character Data Descriptor, the result is a Pointer.

IRW chaining will occur, if required.

INXA E7

The index by means of address couple operator is functionally equivalent to the **INDX** operator except that the address couple follows the **INXA** operator as a fixed fence.

EMODE
only,
E7 on 5700
was implemented
on EMODE

OCRX 9585

The Occurs Index operator computes an offset into a record.

It is primarily used by **COBOL** for array handling and **ALGOL** for **CASE** statements.

The two top-of-stack items must be:

Occurs Index Word (Index Control Word)

Index

The format of an Occurs Index Word is:

```
[47:16] Width Coefficient (width of each occurrance)
[31:16] Upper Bound (one-relative)
[15:16] Offset Coefficient (from beginning of record)
```

OCRX calculates a zero-relative offset from the beginning of the record which can be used by the **INDX** (or similar) operator.

The formula used by **OCRX** is:

```
Relative Index = Offset + (Index - 1) * Width
```

As the **OCRX** operator assumes one-relative indices, if the Index value is less than one or greater than the Upper Bound, an **INVALID INDEX** interrupt will occur.

MPCW BF

The make PCW constructs a PCW at the top of the stack from a six syllable parameter in the code stream.

The TAG is changed to a 7.

The SNR is also inserted into the STKNRF.

The parameter to the MPCW operator must start on a word boundary.

Any syllables after the MPCW and before the parameter are ignored (usually filled with Hex "FF"s).

READ EVALUATION OPERATORS

VALC 00-3F

The value call operator evaluates a reference chain whose head is an address couple parameter.

The end result must be an operand.

If a PCW must be evaluated, accidental procedure entry is performed.

The parameter is a variable fence address couple.

EXAMPLES:

```
1002        VALC (2,2)
3003        VALC (3,3)
2002        VALC (1,2)
```

LVLC 958D

The long value call is equivalent to the VALC except that its parameter is a fixed-fence address couple.

NXLV AD

The index and load value operator performs an **INDX** operation to produce an Indexed Word Data Descriptor and then evaluates the Indexed word data descriptor to fetch an operand.

IRW chaining can occur.

NXVA EF

The index and load value by means of fixed fence address couple parameter is functionally the same as the **NXLV** operator.

NXLN A5

The index and load name operator performs an **INDEX** operation to produce a Indexed Word Data Descriptor and then evaluates to fetch an un-indexed data descriptor.

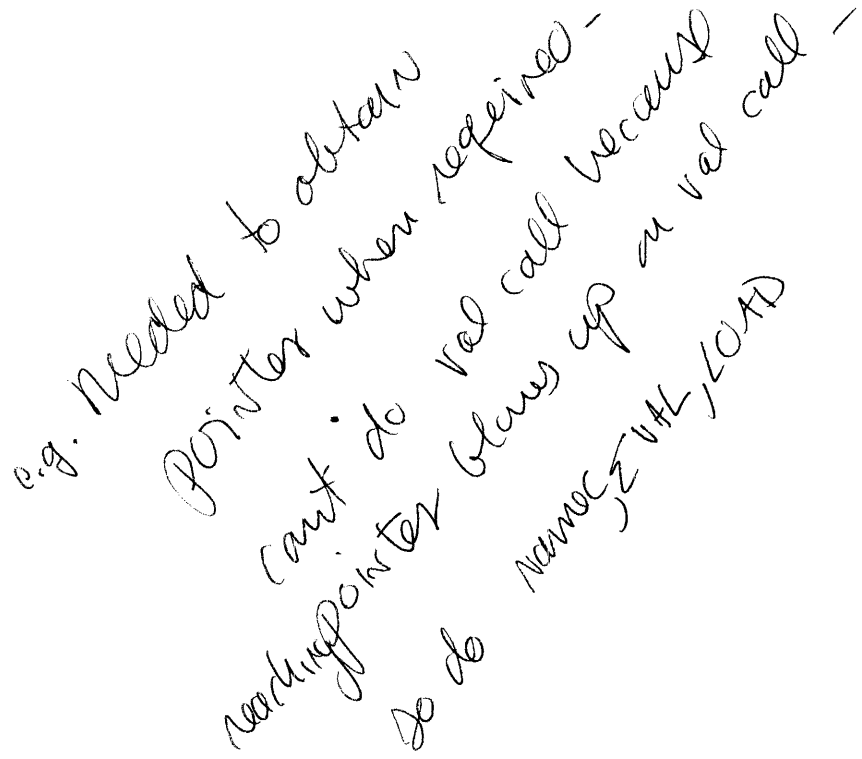
Copy bit action occurs.

EVAL AC

The evaluate operator is used to evaluate a reference chain in order to locate some target and then leave on top of the stack the reference whose evaluation produced the target.

This operator is used when the actual target is needed for execution of another operator (String transfer).

NEWP uses this operator prior to passing a parameter by reference to guarantee the **SIRW** points to the target.



LOAD BD

The load operator fetches a target and places it on top of the stack.

If the reference is a **NIRW (IRW NON-EMODE)** the target is fetched and placed on top of the stack.

If the reference is a **Word** or a **Double Indexed Data Descriptor** the target is fetched and placed on top of the stack.

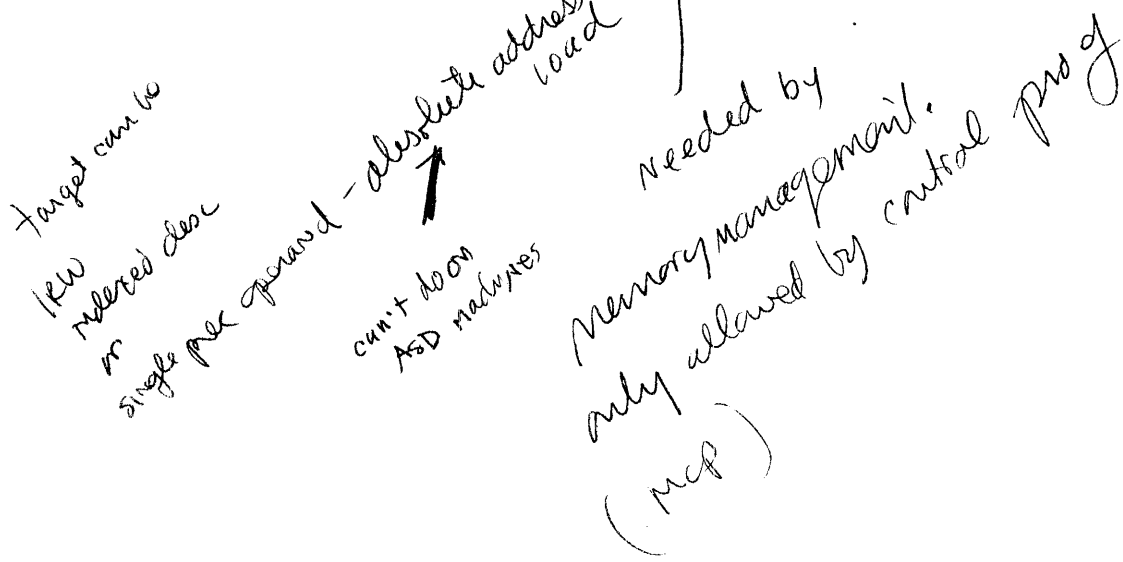
If the target is a Data Descriptor, copy bit action will occur.

LODT 95BC

The load transparent operator performs a **LOAD** operation.

The reference can be a 20 bit address reference.

If the target is a Data Descriptor, no copy bit action occurs (an original descriptor is fetched and place on top of the stack).



STORE EVALUATION OPERATORS

NON-EMODE

STOD B8

The store delete operator places the operand at the top of the stack at the reference pointer.

The order of the operand and reference pointer may vary. I will switch them if needed.

The reference pointer may be an **NIRW** (**IRW** for **NON-EMODE**) or an Indexed Data Descriptor.

IRW chaining may occur.

If the target referenced is a **PCW** accidental procedure entry occurs producing a new reference.

Both the reference and the operand are deleted from the top of the stack.

If the target location has the memory protect bit on, an interrupt occurs.

STON B9

The store non-delete operator performs a **STOD** function, except that at completion of the operation the original operand is left on stop of the stack.

The reference is deleted from the top of the stack.

If the target location has the memory protect bit on, an interrupt occurs.

STAD F6

ENODE

The store delete by means of address couple is functionally equivalent to the STOD operator except that the reference is a fixed-fence address couple parameter.

Both the reference and the operand are deleted from the top of the stack.

If the target location has the memory protect bit on, an interrupt occurs.

STAN F7

The store non-delete by means of address couple is functionally equivalent to the STON operator except that the reference is a fixed-fenced address couple parameter.

The reference is deleted from the top of the stack.

If the target location has the memory protect bit on, an interrupt occurs.

OVERWRITE OPERATORS

used to store into pointers

OVRD BA

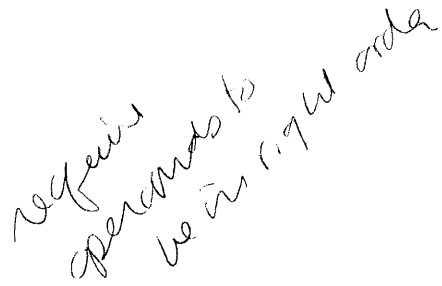
An overwrite delete essentially performs the **STOD** operator, except that no interrupt occurs if the target has the memory protect bit on (i.e. has an odd tag).

The reference and the operand are deleted from the top of the stack.

OVRN BB

The overwrite non-delete functional performs a **OVRD** operator except that the operand is left on top of the stack.

The reference is deleted from the top of the stack.



COMPUTATIONAL OPERATORS

ADD 80

The add operator performs an arithmetic ADD operation on the top 2 operands on top of the stack.

The result is left on the top of the stack.

Program Segment:

```
REAL
    R1                          (2,2)
   ,R2                          (2,3)
   ;

R1 := R1 + R2;
```

Code String:

Non-EMODE:

```
   1002         1003       80     5002        B8
 VALC (2,2)   VALC (2,3)  ADD   NAMC (2,2)  STOD
```

EMODE:

```
   1002         1003       80     F62002
 VALC (2,2)   VALC (2,3)  ADD      STAD
```

SUBT 81

The subtract operator takes the numeric value of the top item and algebraically subtracts is from the numeric value of the second item in the stack.

The result is rounded and left on top of the stack.

Program Segment:

```
REAL
    R1                          (2,2)
   ,R2                          (2,3)
   ,R3                          (2,4)
   ;

R1 := R2 - R3;
```

Code String:

Non-EMODE:

1003	1004	81	5002	B8
VALC (2,3)	VALC (2,4)	SUBT	NAMC (2,2)	STOD

EMODE:

~~1002~~ 1003	~~1003~~ 1004	81	F62002
VALC (2,2)	VALC (2,3)	SUBT	STAD

MULT 82

The multiply operator algebraically multiplies the top two items in the stack.

The result is rounded and left on top of the stack.

DIVD 83

The divide operator algebraically divides the numeric value of the second item by the numeric value of the first item in the stack.

The result is rounded and left on top of the stack.

NTGR 87

INTEGER function in ALGOL.

The integerize rounded operator converts the operand on top of the stack to an integer by rounding and leaving the result on top of the stack.

NTIA 86

ENTIER function in ALGOL.

Integerize truncated returns the greatest integer value less than or equal to the operand.

DIFFERENCE BETWEEN NTGR AND NTIA

There is a small but significant difference between the two operators. Effectively, NTGR adds 0.5 to the value and then does an NTIA.

The following table should help to illustrate:

Value	NTGR	NTIA
3.2	3	3
3.5	4	3
-3.2	-3	-4

COMPLEX ARITHMETIC EXAMPLE

Program Segment:

```
BEGIN
REAL
    R1                                  (2,2)
   ,R2                                  (2,3)
   ,R3                                  (2,4)
   ;

INTEGER
    I1                                  (2,5)
   ,I2                                  (2,6)
   ;

I1 := R1 + R2 * I2 / R3;
```

Reverse Polish Notation String:

```
R1 R2 I2 * R3 / + I1 :=
```

Code String:

Non EMODE:

```
  1002        1003        1005        82
VALC (2,2)  VALC (2,3)  VALC (2,5)  MULT

  1004        83   80    87     5005        B8
VALC (2,4)  DIVD  ADD  NTGR  NAMC (2,5)  STOD
```

EMODE:

```
  1002        1003        1005        82
VALC (2,2)  VALC (2,3)  VALC (2,5)  MULT

  1004        83   80    87     F62005
VALC (2,4)  DIVD  ADD  NTGR      STAD
```

IDIV 84

The integer divide operator causes the numeric value of the second item in the stack to be divided by the numeric value of the first item in the stack.

The fractional part of the quotient is discarded.

The integer part is left on top of the stack.

```
R := R1 DIV R2;
```

Examples:

```
8 DIV 2 = 4
8 DIV 3 = 2
8 DIV 5 = 1
```

RDIV 85

The remainder divide causes the numeric value of the second item in the stack to be divided by the first item in the stack.

The integer quotient with remainder is generated.

The remainder is left on top of the stack.

```
R := R1 MOD R2;
```

Examples:

```
8 MOD 2 = 0
8 MOD 3 = 2
8 MOD 5 = 3
```

AMIN 9588

The arithmetic minimum compares the numerical values of the top two items on top of the stack.

The lesser of the two items is left on top of the stack.

AMAX 958A

The arithmetic maximum compares the numerical values of the top two items in the stack.

The greater of the two items is left on top of the stack.

LOGICAL OPERATORS

Logical operators use one or two items on top of the stack as 48 or 96 bit vectors.

These items may be of any type.

The logical operation is applied in parallel to each bit of the vector.

The logical value of the result depends only on bit zero.

TRUE = 1

FALSE = 0

LNOT 92

The **LOGICAL NOT** operator causes all bits of the item on top of the stack to be complemented.

The **TAG** is unchanged.

Program Segment:

```
REAL
    R1                                  (2,2)
   ;
BOOLEAN
    B1                                  (2,3)
   ;

R1 :=
    1"10111010111011000010001110000111000000011011 0010";
B1 := NOT BOOLEAN (R1);
```

Code Segment:

```
        VALC (2,2) LNOT NAMC (2,3) STOD
```

Result:

```
Before 1"10111010111011000010001110000111000000011011 0010";
After  1"01000101000101111101110001111000111111100100 1101";
```

The logical value of **B1** is **TRUE.**

Program Segment:

```
REAL
    R1                                  (2,2)
   ,R2                                  (2,3)
   ;

R1 := FALSE; % A WORD OF ALL ZEROS

R2 := REAL (NOT R1);
```

Code String:

```
    ZERO  NAMC (R1)  STON  VALC (2,2) LNOT NAMC (2,3) STOD
```

Result:

```
Before: 1"00000000000000000000000000000000000000000000"
After:  1"11111111111111111111111111111111111111111111"
```

The logical value of R2 is **TRUE.**

The above example is functionally the same as the commonly used:

```
R2 := REAL (NOT FALSE);
```

LAND 90

Logically ANDs the two items on top of the stack.

The result is left on top of the stack.

The TAG is double-precision if the result is double.

The TAG is the same as the TAG of the second item in the stack.

The truth table for LAND is as follows:

AND	0	1
0	0	0
1	0	1

Program Segment:

```
REAL
    R1
   ,R2
   ,R3
   ;

R1 :=
   1"0100001001111111010101010001111110000111100011101";
R2 :=
   1"1010001010000101010001111111000101010000111000110";


R3 := REAL (BOOLEAN (R1) AND BOOLEAN (R2));
```

Code String:

```
VALC (R1)  VALC (R2)  LAND  NAMC (R3)  STOD
```

Results:

```
R1:  1"0100001001111111010101010001111110000111100011101"
R2:  1"1010001010000101010001111111000101010000111000110"
      ------------------------------------------------
After: 0000001000000101010001010000001010000000110000100
```

The logical value of R3 is FALSE.

LOR 91

Logically ORs the top two items in the stack.

The result is left on top of the stack.

If the result is double-precision the TAG is double.

Otherwise the TAG is the same as the second item in the stack.

The truth table for LOR is as follows:

OR	0	1
0	0	1
1	1	1

Program Segment:

```
REAL
    R1
   ,R2
   ,R3
   ;

R1 :=
   1"0100001001111111010101010001111110000111100011101";
R2 :=
   1"1010001010000101010001111111000101010000111000110";

R3 := REAL (BOOLEAN (R1) OR BOOLEAN (R2));
```

Code String:

```
VALC (R1)  VALC (R2)  LOR  NAMC (R3)  STOD
```

Results:

```
R1:   1"0100001001111111010101010001111110000111100011101"
R2:   1"1010001010000101010001111111000101010000111000110"
       ---------------------------------------------------
After: 1110001011111111101010111111111111010011111001111
```

The logical value of R3 is TRUE.

LEQV 93

The result of the logical equivalence of the top two items on the stack is performed.

The result is left on top of the stack.

If the result is double-precision the **TAG** is double.

Otherwise the **TAG** is the same as the second item on the stack.

The truth table for **LEQV** is as follows:

EQV	0	1
0	1	0
1	0	1

Program Segment:

```
REAL
    R1
   ,R2
   ,R3
   ;

R1 :=
   1"0100001001111111010101010001111110000111100011101";
R2 :=
   1"1010001010000101010001111111000101010000111000110";

R3 := REAL (BOOLEAN (R1) EQV BOOLEAN (R2));
```

Code String:

```
VALC (R1)   VALC (R2)   LEQV   NAMC (R3)   STOD
```

Result:

```
R1:    1"0100001001111111010101010001111110000111100011101"
R2:    1"1010001010000101010001111111000101010000111000110"
         ---------------------------------------------------
After: 0001111100000101111011010100000010110110011010100
```

The logical value of R3 is FALSE.

EXCLUSIVE OR

NOT EQV

There is no processor operator that performs the **EXCLUSIVE OR** operation.

The following simulates this function using the logical operators already discussed.

The truth table for **EXCLUSIVE OR** is as follows:

XOR	0	1
0	0	1
1	1	0

Program Segment:

```
REAL
     R1
    ,R2
    ,R3
    ;

R1 :=
   1"0100001001111111010101010001111110000111100011101";
R2 :=
   1"1010001010000101010001111111000101010000111000110";

R3 := REAL (NOT (BOOLEAN (R1) EQV BOOLEAN (R2)));
```

Code String:

```
VALC (R1)   VALC (R2)   LEQV LNOT   NAMC (R3)   STOD
```

Results:

```
R1:       1"0100001001111111010101010001111110000111100011101"
R2:       1"1010001010000101010001111111000101010000111000110"
     EQV   ----------------------------------------------------
Temp:      0001111100000101111011010000001011011001101101 00
    LNOT   ----------------------------------------------------
After:     1110000011111010000100101111110100100110010010 11
```

The logical value of R3 is TRUE.

RELATIONAL OPERATORS

The relational operators algebraically compare the numeric value of the top two operands on top of the stack.

The result is left on top of the stack.

The form of the result is a **BOOLEAN.**

BIT ZERO

1 = TRUE

0 = FALSE

All other bits of the result are 0. & irrelevant anyway

SAME 94

algol IS (algol ISNT IS a NOT)

The logical **SAME** operators actually performs the function of logical operators.

The top two items in the stack are compared bit by bit for equal.

All bits are compared in parallel.

The result is left on top of the stack.

The result is a **BOOLEAN.**

If all bits are the same, the result is **TRUE.**

If all bits are not the same, the result is **FALSE.**

Program Segment:

```
REAL
    R1
   ,R2
   ;
BOOLEAN
    B1
   ;

R1 :=
   1"0100001001111111010101010001111110000111100011101";
R2 :=
   1"1010001010000101010001111111000101010000111000110";

B1 := R1 IS R2;
```

Code Stream:

```
VALC (R1)   VALC (R2)   SAME   NAMC (B1)   STOD
```

Result:

```
R1:    1"0100001001111111010101010001111110000111100011101"
R2:    1"1010001010000101010001111111000101010000111000110"
SAME   ----------------------------------------------------
After:  00000000000000000000000000000000000000000000000000
```

The value of B1 is FALSE.

LESS 88

LESS leaves **TRUE** result if the second from top-of-stack operand is arithmetically less than the top of stack operand and a **FALSE** result otherwise.

```
R1 LSS R2
```

LSEQ 8B

LSEQ leaves **TRUE** result if the second from top-of-stack operand is arithmetically less than or equal to the top of stack operand and a **FALSE** result otherwise.

```
R1 LEQ R2
```

EQUL 8C

EQUL leaves TRUE result if the second from top-of-stack operand is arithmetically equal to the top of stack operand and a FALSE result otherwise.

```
R1 EQL R2
```

NEQL 8D

NEQL leaves TRUE result if the second from top-of-stack operand is arithmetically not equal to the top of stack operand and a FALSE result otherwise.

```
R1 NEQ R2
```

GREQ 89

GREQ leaves TRUE result if the second from top-of-stack operand is arithmetically greater than or equal to the top of stack operand and a FALSE result otherwise.

```
R1 GEQ R2
```

GRTR 8A

GRTR leaves TRUE result if the second from top-of-stack operand is arithmetically greater than the top of stack operand and a FALSE result otherwise.

```
R1 GTR R2
```

LITERAL OPERATORS

Places a single precision constant on top of the stack.

ZERO B0

ZERO leaves on the top of the stack a single-precision word with all bits initialized to zero.

Program Segment:

```
R1 := 0;
```

Code Stream:

```
ZERO  NAMC (R1)  STOD
```

ONE B1

ONE leaves on the top of the stack a 1-bit integer equal to 1.

Program Segment:

```
R1 := 1;
```

Code Stream:

```
ONE  NAMC (R1)  STOD
```

LT8 B2

Insert 8 bit literal; leaves on top of the stack an 8-bit integer that is a copy of its one-syllable parameter.

Program Segment:

```
R1 := 100;                    % R1 = (2,2)
```

Code Stream:

```
    B264        5002      B8

LT8 (100)   NAMC (R1) STOD
```

LT16 B3

Insert 16 bit literal; leaves on top of the stack a 16-bit integer that is a copy of its two-syllable parameter.

Program Segment:

```
R1 := 4096;                   % R1 = (2,2)
```

Code Stream:

```
    B31000        5002       B8

LT16 (4096)   NAMC (R1)   STOD
```

LT48 BE

Insert 48 bit literal leaves on the top of the stack a single-precision operand that is a copy of its six-syllable parameter.

Similar to the **MPCW** operator.

The parameter (i.e. the 48 bit literal) must start on a word boundary.

All syllables are ignored between the **LT48** operator and its parameter.

Program Segment:

```
R1 := "ABCDEF";                    % R1 = (2,2)
```

Code Stream:

```
BEFFFFFFFFFF C1C2C3C4C5C6          5002      B8
   LT48 ("ABCDEF")                 NAMC (R1) STOD
```

BRANCHING OPERATORS

Provide for altering the processor's sequence through the code stream.

Branches may be conditional or un-conditional.

May be static or dynamic.

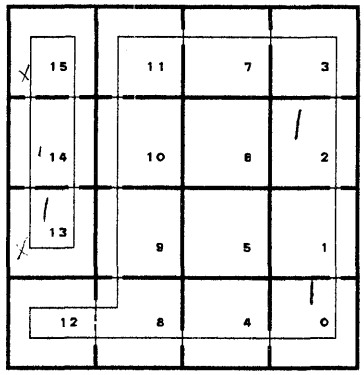
STATIC BRANCHES

A two-syllable parameter designates the PWI:PSI within the current segment.

Branching may only occur within the same code-segment (SDI).

[15:03] = PSI

[12:13] = PWI



used by ALGOL for branches

Example:

Value	PWI	:	PSI
6005	5	:	3

BRUN A2

Processor registers PSI and PWI are set from the parameter.

The code-stream transfers to the next operator in that code segment.

```
GO TO XIT
```

BRTR A1

The top-of-stack item is interpreted as a **BOOLEAN** value.

If the logical value is **FALSE:**

PWI:PSI are set to the next operator.

Sequential processing continues.

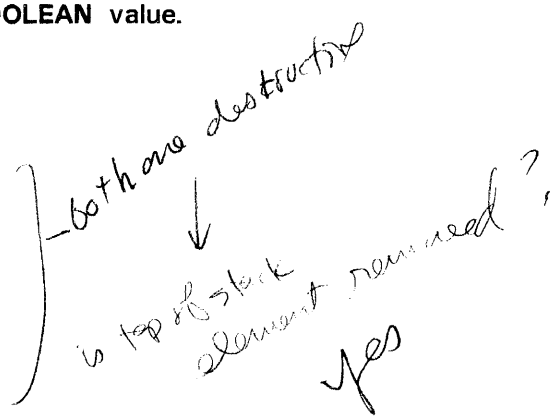
Fall through.

If the logical value is **TRUE:**

PWI:PSI are set to parameter values.

The processor begins execution at that point in the code-stream.

Actual branch occurs.



BRFL. A0

The top-of-stack item is interpreted as a **BOOLEAN** value.

If the logical value is **TRUE:**

PWI:PSI are set to the next operator.

Sequential processing continues.

Fall through.

If the logical value is **FALSE:**

PWI:PSI are set to parameter values.

The processor begins execution at that point in the code-stream.

Actual branch occurs.

STATIC BRANCH EXAMPLES

Program Segment:

```
BEGIN
REAL
    R1                                  (2,2)
   ,R2                                  (2,3)
   ,R3                                  (2,4)
   ;

BOOLEAN
    B                                   (2,5)
   ;

R1 := 25;
R2 := 35;

IF (B := R1 GTR R2)
  THEN
    R1 := R2
  ELSE
    R1 := R3;

END.
```

Code Stream:

```
                    IF (B := R1 GTR R2) THEN

        VALC (R1)   VALC (R2)   GRTR   NAMC (B)   STON BRFL 7:3
3:4:2     1002        1003       8A      5005       B9    A06007
%-------------------------------------------------------------

                         R1 := R2

        VALC (R2)   NAMC (R1)   STOD   BRUN 8:2
3:6:1     1003        5002       B8     A24008
%-------------------------------------------------------------

                    ELSE
                         R1 := R3;

        VALC (R3)   NAMC (R1)   STOD
3:7:3     1004        5002       B8
%-------------------------------------------------------------
3:8:2
```

Program Segment:

```
REAL
    R1                                  (2,2)
   ,R2                                  (2,3)
   ,R3                                  (2,4)
   ;

BOOLEAN
    B                                   (2,5)
   ;

R1 := 25;
R2 := 35;

IF NOT (B := R1 GTR R2)
  THEN
    R1 := R2
  ELSE
    R1 := R3;

END.
```

Code Stream:

```
                IF NOT (B := R1 GTR R2) THEN

        VALC (R1)   VALC (R2)   GRTR   NAMC (B)   STON BRTR 7:3
3:4:2     1002        1003       8A      5005      B9    A06007
%-------------------------------------------------------------

                          R1 := R2

        VALC (R2)   NAMC (R1)   STOD   BRUN 8:2
3:6:1     1003        5002       B8     A24008
%-------------------------------------------------------------

                      ELSE

                           R1 := R3;

        VALC (R3)   NAMC (R1)   STOD
3:7:3     1004        5002       B8
%-------------------------------------------------------------
3:8:2
```

DYNAMIC BRANCHES

The top-of-stack item contains the code-stream pointers.

The branch may with the current segment or a different segment.

IRW chaining will occur if the item on top of the stack is an **NIRW** (IRW on non-**EMODE**).

If the target is a **PCW**:

SDI:PWI:PSI are set from the **PCW**.

The **LEX** level in the **PCW** must be the same as the current **LEX** level (LL).

The control state indicator is ignored.

DYNAMIC BRANCH TARGETS

If the target is an operand:

Integerized.

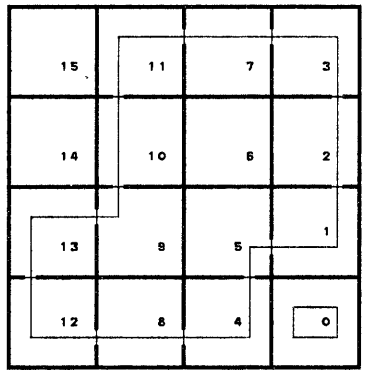
Rounded.

Uses a 14 bit integer.

```
[15:02] = Ignored

[13:13] = PWI (note awkward shape of field)

[00:01] = PSI indicator:
          If this bit is on,
              PSI := 3,
          else
              PSI := 0
```



Example:

```
Value   PWI : PSI
0015     A     3
```

The operand form is typically used in the CASE statement.

DBUN AA unconditional

DBTR A9 true

DBFL A8 false

DYNAMIC BRANCH EXAMPLE

```
77 I PIC 9 (11) BINARY.                              I = (2,5)

PAR-A-SEC       SECTION.
PAR-A.                                               3:1:5
     PERFORM PAR-B-SEC.                              3:1:5

          MPCW BF    3:4:0                          3:1:5
          LT8  B2    04                             3:3:0
          NAMC       501A        (2,1A)             3:3:2
          DBUN AA                                   3:3:5
     STOP RUN.                                      3:4:0

PAR-B-SEC       SECTION.
                                          PCW = 4:3:5 @ (2,1A)
PAR-B.                                              4:3:5
     MOVE 1 TO I.                                   4:3:5

          ONE  B1                                   4:3:5
          NAMC (I)   5005        (2,5)              4:4:0
          STOD B8                                   4:4:2


          DUPL B7                                   4:4:3
          LT8  B2    04                             4:4:4
          SAME 94                                   4:4:5
          BRFL A0    5:5                            4:5:0
          DLET B5                                   4:5:3
          DBUN AA                                   4:5:4
                                                    4:5:5
```

If the **PERFORM** target paragraph was in the same section, a **BRUN** would have been used.

WORD MANIPULATION OPERATORS

Provide the capability to alter any partial field of a word in the stack.

Basic elements.

Destination word.

Source word.

Number of bits to transfer.

Field bits.

Destination start bit ("to" bit).

Source start bit ("from" bit).

The altered, destination item is left on top of the stack.

If the source is double-precision the second word (LSP) is discarded.

If the destination is double-precision the first word is altered, the second word remains unchanged.

Static and dynamic operators.

Static operators obtain the following values from the parameter:

Source start bit.

Destination start bit.

Number of bits to transfer.

Dynamic obtain its these values from items on top of the stack.

BSET 96

Bit set initializes a destination bit to 1.

The destination bit is specified by the parameter.

The destination is the word on top of the stack.

The result is left on top of the stack.

Program Segment:

```
R1.[36:1] := 1;                % R1 = (2,F)
```

Code Stream:

```
  100F        9624        500F        B8
VALC (R1)   BSET (36)   NAMC (R1)   STOD
```

DBST 97

Dynamic bit set is functionally the same as the BSET operator.

The destination bit is the item on top of the stack.

The destination word is the second item on top of the stack.

The result is left on top of the stack.

Program Segment:

```
R1.[BIT:1] := 1;          % R1 = (2,2F), BIT = (3,7)
```

Code Stream:

```
  102F        3007        97      502F        B8
VALC (R1)   VALC (BIT)   DBST   NAMC (R1)   STOD
```

BRST 9E

Bit reset initializes a destination bit to 0.

The destination bit is specified by the parameter.

The destination is the word on top of the stack.

The result is left on top of the stack.

DBRS 9F

Dynamic bit is functionally the same as the BRST operator.

The destination bit is the item on top of the stack.

The destination word is the second item on top of the stack.

The result is left on top of the stack.

ISOL 9A

Field isolate:

Initializes a single precision destination word to zero.

Sets its low order field from a field in the source.

The source start bit is the first parameter.

The number of bits to isolate is the second parameter.

The destination start bit is calculated as follows:

destination start bit = number of bits to transfer - 1.

Program Segment:

```
I.[47:08]          % I = (2,7)
```

Code Stream:

```
VALC (1)   ISOL 47:08
  1007       9A2F08
```

```
Source start bit:        47
Destination start bit:   07
Number of bits:          08
```

The result is left on top of the stack.

DISO 9B

The dynamic field isolate is functionally the same as the ISOL operator.

The initial stack state is as follows:

Number of bits.

Source start bit.

Source item.

The result is left on top of the stack.

Program Segment:

```
REAL
    R1                                  (2,2)
   ,R2                                  (2,3)
   ,R3                                  (2,4)
   ;

R1 := 100;
R2 := 47;
R3 := 08;

R1 := R1.[R2:R3];
```

Code Stream:

VALC (R1)	VALC (R2)	VALC (R3)	DISO	NAMC (R1)	STOD
1002	1003	1004	9B	5002	B8

INSR 9C

The field insert sets a field of the destination from the low order field of the source.

The initial stack state:

Source item.

Destination item.

The destination start bit is the first parameter.

The number of bits is the second parameter.

The source start bit is calculated as follows:

Source start bit = number of bits - 1

Program Segment:

```
REAL
    R1                                    (2,2)
   ,R2                                    (2,3)
   ;

R1.[47:08] := R2;
```

Code Stream:

VALC (R1)	VALC (R2)	INSR 47:08	NAMC (R1)	STOD
1002	1003	9C2F08	5002	B8

Source start bit: 07

Destination start bit: 47

Number of bits: 08

The result is left on top of the stack.

DINS 9D

The dynamic field insert is functionally the same as the INSR operator.

The initial stack state is as follows:

Source item.

Number of bits.

Destination start bit.

Destination item.

The result is left on top of the stack.

Program Segment:

```
REAL
    R1                                  (2,2)
   ,R2                                  (2;3)
   ,R3                                  (2,4)
   ;

R1 := 100;
R2 := 47;
R3 := 08;

R1.[R2:R3] := R1;
```

Code Stream:

```
VALC (R1)   VALC (R2)   VALC (R3)   VALC (R1)

  1002        1003        1004        1002

DINS  NAMC (R1)  STOD

 9B     5002      B8
```

FLTR 98

Field transfer sets a field of the destination from a field of the source.

The initial stack state.

Source item.

Destination item.

Parameter values.

Destination start bit.

Source start bit.

Number of bits.

The result is left on top of the stack.

Program Segment:

```
REAL
    R1                                  (2,2)
   ,R2                                  (2,3)
   ,R3                                  (2,4)
   ;

R1 := R2 & R3 [47:35:08];
```

Code Stream:

VALC (R2)	VALC (R3)	FLTR 47:35:08	NAMC (R1)	STOD
1003	1004	982F2308	5002	B8

DFTR 99

The dynamic field transfer is functionally the same as the FLTR operator, except the parameters are located on top of the stack.

The initial stack state.

Number of bits.

Source start bit.

Destination start bit.

Source item.

Destination item.

The result is left on top of the stack.

Program Segment:

```
REAL
    R1                                  (2,2)
   ,R2                                  (2,3)
   ,R3                                  (2,4)
   ,R4                                  (2,5)
   ,R5                                  (2,6)
   ,R6                                  (2,7)
   ;

R1 := R2 & R3 [R4:R5:R6];
```

Code Stream:

```
VALC (R2)   VALC (R3)   VALC (R4)   VALC (R5)   VALC (R6)

  1003        1004        1005        1006        1007


DFTR  NAMC (R1)  STOD

 99     5002      B8
```

STACK STRUCTURE

Provide procedure entry and exit.

Sets, saves, and restores processor state components.

Maintains linkage of activation records, both historical and lexical.

Display update required upon procedure exit or entry.

Display update is terminated when looping through decreasing lexical levels by the following:

The new value for D[i] is the same as the current.

The environment number (i) is less than the previous value of LL.

MKST AE

Mark stack builds an inactive MSCW (Mark Stack Control Word) on top of the stack.

Inserted at the head of the historical chain.

The difference between the current location of the MSCW and F is calculated (S + 1 - F).

This difference is inserted in the HISTORY_LINK field of the MSCW.

MKSN DF

The mark-stack bound to name-call is functionally equivalent to the MKST operator.

The operator immediately following the MKSN must be a NAMC.

Observance to the MKSN rules is critical.

E-MODE operator.

ENTR AB

The enter operator completes the procedure entry process.

Assumes prior execution of MKST.

Inactive MSCW must be at F.

F + 1 must be a PCW after IRW chaining.

Completes the MSCW.

Inserts MSCW at the head of the appropriate lexical chain.

The LL field is filled with the LL field of the PCW.

The MSCW is marked as entered.

Constructs an RCW.

Saves the current processor code-stream pointer.

Saves BOOLEAN accumulators.

LL set to the current LL prior to procedure entry.

CS (control state) set to the value in the PCW.

Initializes processor state for the procedure being entered.

Code-stream pointer.

Addressing environment.

Transfers control to the code-stream pointers set from the PCW.

EXIT A3

The EXIT deletes the topmost activation record from the stack.

Returns execution to the prior activation record.

S is set to D[LL] - 1.

F is set to the location referenced by:

Current F - MSCW.historylink

The following registers are set from the fields in the RCW:

LL

Boolean processor accumulators: TFFF, OFFF

CS (control state)

Processor code-stream pointers (SDI:PWI:PSI)

Control is transferred to the code-stream pointed to by the RCW.

RETN A7

The **RETURN** operator is exactly the save as **EXIT**, except:

The top-of-stack item is retained.

The returned item is placed onto the top of the stack after the top most activation record is deleted.

MVST 95AF

switch stacks

The move to stack changes the processor's site of activity by deactivating the current stack and activating a destina- tion stack.

A new memory addressing environment is established.

A TOSCW is stored at the base of the inactive stack.

TOSCW of the destination stack used to find the height (S) and the start of the historical chain (F).

The TOSCW is sufficient by itself to activate the stack.

Single precision top_of_stack item used as the destination stack number.

environmental num.

```
[23:12]   Destination environment number.
[11:12]   Destination stack number.
```

Restores the stack state.

BOSR := Stack descriptor.address.

LOSR := Stack descriptor.length + BOSR.

S := TOSCW.stack_height + BOSR.

F := S - TOSCW.SF_displacement.

Updates the LEXICAL environment state.

POINTER OPERATORS

Deal with sequences of word or character elements in an array.

Operations include:

- Scanning
- Comparing
- Transferring
- Editing

Most pointer operations require initial stack arguments that specify:

- Length
- Source
- Destination

Source can be an operand.

Source and destination pointers can be updated in most operations.

Length can be updated to reflect the number of items left to process.

UNCONDITIONAL TRANSFER

TUND E6

Transfer Characters Unconditional Delete transfers a specified number of characters.

The initial stack state:

Length

Source

Destination

No results are left on the stack.

TUNU EE

Transfer characters unconditional update functionally equivalent to the TUND operator, except:

Updated source and destination pointers are left on the stack.

No updated length because operation is unconditional.

Compiler generates DLET if one of the pointers is not required to be updated.

TWSD D3

Transfer words delete performs an unconditional transfer, except:

Length is in units of words.

Actual transfer is word at a time.

No result left on top of the stack.

TWSU DB

Transfer words update performs is functionally the same as the TWSD operator except:

The updated source and destination pointers are left on top of the stack.

EXAMPLES

Program Segment:

```
REPLACE P BY P2 FOR COUNT;
```

Code Stream:

```
NAMC (P)  LOAD  NAMC (P2) LOAD  VALC (COUNT)  TUND
```

Program Segment:

```
REPLACE P:P BY P2:P2 FOR COUNT;
```

Code Stream:

```
NAMC (P)  LOAD  NAMC (P2)  LOAD  VALC (COUNT)  TUNU
NAMC (P2)  OVRD  NAMC (P)  OVRD
```

Program Segment:

```
REPLACE P:P BY P2 FOR COUNT WORDS;
```

Code Stream:

```
NAMC (P)  LOAD  NAMC (P2)  LOAD  VALC (COUNT)  TWSU
DLET  NAMC (P) OVRD
```

SCAN OPERATORS

Character-relative scan operators sequentially compare each source character to the delimiter character.

The initial stack state.

Delimiter. right justify - single target char.

Length.

Source.

DELETE SCAN OPERATORS

The following operators leave no results on the stack:

SGTD (95F2) (scan while greater delete)

SGED (95F1) (scan while greater or equal delete)

SEQD (95F4) (scan while equal delete)

SNED (95F5) (scan while not equal delete)

SLED (95F3) (scan while less than or equal delete)

SLSD (95F0) (scan while less than delete)

never used. what good is it if you get NO returned info

UPDATE SCAN OPERATORS

The update scan operators are the same as the delete scan operators except:

Updated length left on the stack.

Updated source pointer left on top of the stack.

EXAMPLES

Program Segment:

```
SCAN P FOR 10 WHILE = " ";
```

Code Stream:

```
NAMC (P)   LOAD   LT8 (10)   LT8 (" ")   SEQD
```

Program Segment:

```
SCAN P:P FOR COUNT:10 UNTIL = " ";
```

Code Stream:

```
NAMC (P)   LOAD   LT8 (10)   LT8 (" ")   SNEU

NAMC (COUNT)   STOD   NAMC (P)   OVRD
```

CHARACTER TRANSFER OPERATORS

Character-relative transfer sequentially compares each source character to the delimiter character.

Each source character that satisfies the relation is transferred to the destination.

Initial stack state:

Delimiter

Length

Source

Destination

CHARACTER TRANSFER DELETE

The following operators leave no results on the stack:

TGTD (E2) (transfer while greater delete)

TGED (E1) (transfer while greater than or equal)

TEQD (E4) (transfer while equal)

TNED (E5) (transfer while not equal delete)

TLED (EB) (transfer while less than or equal)

TLSD (E0) (transfer while less than delete)

CHARACTER TRANSFER UPDATE

The character transfer operators perform a characters transfer except:

The updated length is left on top of the stack.

The update source is left on the stack.

The updated destination is left on the stack.

EXAMPLES

Program Source:

```
REPLACE P BY P2 FOR 10 WHILE NEQ " ";
```

Code Stream:

```
NAMC (P)  LOAD  NAMC (P2)  LOAD  LT8 (10)  LT8 (" ")  TNED
```

Program Source:

```
REPLACE P BY P2:P2 FOR 10 UNTIL NEQ " ";
```

Code Stream:

```
NAMC (P)  LOAD  NAMC (P2)  LOAD  LT8 (10)  LT8 (" ") TEQU
DLET  NAMC (P2)  OVRD  DLET
```

CHARACTER COMPARE OPERATORS

Character-sequence compare apply a relational comparison of each source.

The TFFF boolean accumulator is set to 1 if the relation is satisfied and 0 if the relation fails.

The initial stack state.

Length

Source1

Source2

Boolean result left in TFFF in hardware, must read RTFF

CHARACTER COMPARE DELETE

The following operators terminate when the actual relation is determined.

No result is left on top of the stack.

CGTD (F2) (compare characters greater delete)

CGED (F1) (compare characters greater than or equal delete)

CEQD (F4) (compare characters equal delete)

CNED (F5) (compare characters not equal delete)

CLED (F3) (compare characters less than or equal delete)

CLSD (F0) (compare characters less than delete)

CHARACTER COMPARE UPDATE

The character-sequence compare update operators perform the same as the character-sequence compare delete operators except:

Both source pointers are update and left on the stack.

for not length

EXAMPLES

Program Segment:

```
B := P GTR P2 FOR 3;
```

Code Stream:

```
NAMC (P)  LOAD  NAMC (P2)  LOAD  LT8 (3)  CGTD
RTFF  NAMC (B)  STOD
```

Program Segment:

```
B := P:P GTR P2 FOR 3;
```

Code Stream:

```
NAMC (P)  LOAD  NAMC (P2)  LOAD  LT8 (3)  CGTU
DLET  NAMC (P)  OVRD  RTFF  NAMC (B)  STOD
```

EXAMPLES OF COMPILER-GENERATED CODE

Review the following listing analyzing as many of the constructs that time permits.

CODESTRING/POINTERS

CODESTRING/STRINGEX

CODESTRING/SCANPOINTERS

CODESTRING/TRANSLATE

CODESTRING/ACCIDENTAL

CODESTRING/GOTOSOLVER

CODESTRING/PASSBYNAME

CODESTRING/TYPEDPROCEDURE

CODESTRING/PERFORMS

CODESTRING/NESTEDPERFORMS

CODESTRING/WORDTYPES

CODESTRING/LOWERBOUNDS

CODESTRING/CASE

CODESTRING/TADS

INTERRUPTS

OVERVIEW

An interrupt is an automatic invocation of an operating system procedure.

May be invoked by:

Operators (operator dependent)

Between operators (external interrupts)

Any time (alarm interrupts).

An interrupt causes an MCP procedure (HARDWAREINTERRUPT) whose PCW, or an SIRW chain, is located a D[0] + 3.

The following is the steps to interrupt entry:

Invoke MKST.

Place an NIRW (IRW non-E-MODE) to D[0] + 3.

Place a variable numbers of words (depending on hardware type) on the stack.

Invoke ENTR.

Interrupts are divided into three classes:

ODI

An operator dependent interrupt is invoked directly by the current operator to request an MCP service required by the operator or to report a programming or operator fault.

Alarm

An Alarm interrupt is triggered by hardware fault detection during operator execution.

External

An External interrupt is invoked between operators to report events that are independent of the executing code.

The optional parameters include:

P1 parameter (also called interrupt ID parameter).

P2 parameter.

Contents vary depending on type on interrupt.

OPERATOR DEPENDENT

MCP SERVICE

Requests for an MCP service that is an extension of the hardware operators.

PRESENCE BIT

Used by operators to gain access to a data array or program code-segment that is not present in memory.

Occurs if the presence bit in the data descriptor = 0.

The P2 parameter is a copy of the data descriptor that caused the interrupt.

If the descriptor is a COPY descriptor, the address filed contains the memory location of the original data descriptor.

A presence bit interrupt is not generated if the descriptor is a COPY and the original descriptor is present in memory.
(NOTE: Not true on some earlier hardware.)

PAGED ARRAY

Used by pointer operators to indicate an attempt to access beyond the end of the array or page.

Attempt to access a word that has the memory protect bit = 1.

If the data array is not segmented an error is generated.

If the data array is segmented (virtual):

If at the end of the data segment

Error condition (Seg Array Err or Paged Array Err)

Otherwise

Next array page made resident in memory

Pointer updated on the stack.

Return from the interrupt procedure to resume the operator on the next page of the array.

BINDING REQUEST

Generated when IRW chaining produces a data descriptor that has an SIZEF = 7.

Except EVAL.

Interpreted by the software.

STACK OVERFLOW

Attempt to add an item on the stack passed the limit.

S greater than LOSR.

MCP attempts to extend the stack and resume processing.

BLOCK EXIT

EXIT and RETN operators can generate this interrupt.

Interrupt generated when attempt to deallocate an activation record that has the block_exit bit of the RCW = 1.

(NOTE: A-Series only)

ERROR REPORTING

Programming, compiler or operator faults.

INVALID OP

Generated by execution of **NVLD** (invalid operator).

No other operator generates this interrupt.

Other interrupts that are similar in nature:

UNDEFINED OPERATOR

INVALID STACK ARGUMENT

INVALID ARGUMENT VALUE

INVALID CODE PARAMETER

INVALID REFERENCE

INVALID REFERENCE CHAIN

INVALID OBJECT

When an application program encounters an **INVALID OP** message, it is usually due to an application bug such as not having initialized a variable properly.

INVALID INDEX

Attempt to access an array element not within a valid index range for that array.

MEMORY PROTECT

Attempt to write into a memory location that has a memory protected word.

DIVIDE BY ZERO

Generated by arithmetic divide operators if the numeric interpretation of the top-of-stack operand (the divisor) is zero.

INTEGER OVERFLOW

Indicates that an operand required to have an integer value cannot be represented as an integer.

ALARM

Alarm interrupts are triggered by hardware fault detection.

INVALID ADDRESS

UNCORRECTABLE MEMORY ERROR

LOOP TIMER

HARDWARE ERROR

EXTERNAL

Reports events that are independent of the executing code stream.

If in Control State, the **EXTERNAL** interrupt is normally queued until return to Normal State.

IO FINISH

Caused by the IO subsystem to report a physical IO completion.

Not necessarily generated for each physical IO complete.

INTERVAL TIMER

Periodically set by MCP processor control procedures to enforce task priority.

Prevents lower priority CPU-bound tasks from maintaining control of the processor.

System relies on interrupt driven mechanism.

PROGRAMMATIC FAULT HANDLING

Programs can handle various faults and interrupts if they so desire.

This is done through the use of the **ON** <fault list> statement in **ALGOL**.

A program has the ability to continue running after detecting the fault, or terminating.

HISTORY information can be obtained to be used in the fault processing, if desired.

Example:

```
BEGIN
REAL ARRAY A [0:9];
REAL NDX;
ON INVALIDINDEX,
    BEGIN
    DISPLAY ("INVALID INDEX WHEN NDX WAS:  " CAT
             STRING (NDX,*));
    END;
WHILE TRUE DO
    BEGIN
    A [NDX] := NDX;
    NDX := * + 1;
    END;
END;
```

The above program would issue the message:

```
INVALID INDEX WHEN NDX WAS:  10
```

PROGRAMMATIC INTERRUPTS

It is possible to write software interrupts that can be entered dynamically.

In other words, at program compilation time, there may or may not be an explicit call on the **INTERRUPT**.

INTERRUPTs look just like **PROCEDURE**s, and can be called just like procedures, or they can be **ATTACH**ed to an **EVENT**.

If **ATTACH**ed to an **EVENT**, whenever the **EVENT** is **CAUSE**d, the **INTERRUPT** is dynamically entered.

Example:

```
BEGIN
FILE F;
REAL RECORD_CNTR;

INTERRUPT DISPLAY_RECORD_CNTR;
    BEGIN
    DISPLAY ("RECORDS READ = " CAT
             STRING (RECORD_CNTR,*));
    END OF INTERRUPT DISPLAY_RECORD_CNTR;

ATTACH DISPLAY_RECORD_CNTR TO MYSELF.EXCEPTIONEVENT;
WHILE NOT READ (F) DO RECORD_CNTR := * + 1;
END OF PROGRAM.
```

While this program is running, doing a ?<mix nbr>HI will **CAUSE** the event **MYSELF.EXCEPTIONEVENT**. The **MCP** will automatically call the **INTERRUPT** **DISPLAY_RECORD_CNTR** which will in turn **DISPLAY** the number of records read from file **F**.

When **DISPLAY_RECORD_CNTR** exits, the program will resume processing where it was.

Unless the **INTERRUPT** informs the program that it has been called, the program doesn't even know that an interrupt occurred.

PROBLEM ANALYSIS TECHNIQUES

Recommended steps in using program dump to help resolve program failure.

Observe MCP release.cycle.patch levels.

Identify system by using system serial number.

Notice local box (relate to SUBSYSTEM).

Review RCW history.

Record MCP history locations.

May be useful to determine exactly what MCP was doing.

Find first USER RCW.

Notice if this activation record history belongs to another environment.

Could be LIBRARY call.

Is this lexical region MCP procedure HARDWARE INTERRUPT.

Identify syllable that failed.

Record P2 parameter.

May be useful later.

Reverse from that syllable until location of a statement end syllable.

STOD

TUND

EXIT

Proceed from the next syllable forward until the operator that failed.

Build temporary stack of results of re-executing the operators.

Continue passed operator that failed until statement end.

Observe stack cells between next two MSCWs.

Top stack items could be residual cells from operator that failed.

PAGED ARRAY error.

Updated length.

Updated source pointer.

Updated destination pointer.

INVALID INDEX.

Actual index.

P2 is descriptor we are trying to index.

PROGRAM FAILURE REVIEW

Review program listings/dumps as outlined by the instructor.

FILE INFORMATION BLOCKS

FIB STRUCTURE

Label equation block.

Disk file headers.

    Layout in appendix.

Buffers.

IOCB.

Logical record area.

Transaction count.

Physical IO count.

    READS

    WRITES

SYSTEM SOFTWARE COMPILATION

OVERVIEW

Periodic requirement to compile environmental software.

Install patches.

Install flashes.

Generate compiled listings.

Burroughs provides most of the symbolics with software release tapes.

Can react quickly to install recommended software modifications intended to correct environmental software problems.

System provided **WORK FLOW JOB** which can compile most all or selected software components.

Refer to appendix for source listing of **WFL** job.

Documented in **A-SERIES SYSTEM SUPPORT**.

PATCHESFOR

PATCHESFOR/ <software item>.

For each <software item> present, the WFL job will execute a software compilation.

OPTIONS

Module selection options.

These options are **BOOLEAN** variables.

Valid values:

TRUE

FALSE

```
COMPILE_ALL:=TRUE;
```

COMPILE ALL

Option allows all of the system software to be compiled regardless if a **PATCHESFOR** files is present.

FALSE means selective compilation.

SKIP IF NO SYMBOL

Software modules are skipped if their corresponding symbol files are not present.

Can be combined with **COMPILE_ALL** option.

REQUIRED FILES

WFL/COMPILE/SOFTWARE

SYSTEM/DCALGOL

PATCHESFOR/ <each software item to compile>

Symbolic file for compilation

DATABASE/PROPERTIES

GENERATED FILES

Code file (for each successful compilation)

COMPLETED/<software item>

Symbolic files (only if needed).

PROCESS

Generate **PATCHESFOR/**<software item> patch files.

SYSTEM/PATCH format.

Load symbolic.

START WFL job.

Respond to **ODT** input requests:

Only required if job not run under privileged usercode.

Insure successful compile:

COMPLETED/<software item> resident.

Refer to listing for errors (if any).

Correct and restart process.

A SERIES PROCESSOR OPERATORS

BURROUGHS A SERIES PROCESSOR OPERATORS

Here is a detailed description of the columns used in each of the tables that appear on the following pages in this section:

Mne	This is the Mnemonic code given the operator.
Mode	The "Mode" of the operator: <blank> is Primary Mode e is Edit Mode; t is Table Edit Mode (not a true mode); v is Variant Mode.
Op	The Hexadecimal code for the operator.
Syls	Number of code syllables for the total instruction.
Function	A description of the function of the operator.
References	Page numbers in the Burroughs A-Series System Architecture Reference Manual, Volume 2 (Form 5014954, April, 1985) where the operators are documented.

OPERATORS LISTED BY MNEMONIC NAME

Mne	Mode	Op	Syls	Function	References
ADD		80	1	Add	3-007 B-07
AMAX	v	958A	2	Arithmetic Maximum	3-009 B-07
AMIN	v	9588	2	Arithmetic Minimum	3-009 B-07
ASRT	v	9580	3	Assert	3-101 B-08
BCD	v	9577	3	Binary Convert to Decimal	3-016 B-08
BRFL		A0	3	Branch False	3-049 B-08
BRTR		A1	3	Branch True	3-049 B-09
BRST		9E	2	Bit Reset	3-022 B-09
BRUN		A2	3	Branch Unconditional	3-049 B-09
BSET		96	2	Bit Set	3-022 B-09
CBON	v	95BB	2	Count Binary Ones	3-021 B-09
CEQD		F4	1	Compare Chars Equal Delete	3-081 B-10
CEQU		FC	1	Compare Chars Equal Update	3-081 B-10
CGED		F1	1	Compare Chars Greater or Equal Delete	3-081 B-10
CGEU		F9	1	Compare Chars Greater or Equal Update	3-081 B-11
CGTD		F2	1	Compare Chars Greater Delete	3-081 B-11
CGTU		FA	1	Compare Chars Greater Update	3-081 B-11
CHSN		8E	1	Change Sign	3-025 B-11
CLED		F3	1	Compare Chars Less or Equal Delete	3-081 B-11
CLEU		FB	1	Compare Chars Less or Equal Update	3-081 B-11
CLSD		F0	1	Compare Chars Less Delete	3-081 B-12
CLSU		F8	1	Compare Chars Less Update	3-081 B-12
CNED		F5	1	Compare Chars Not Equal Delete	3-081 B-12
CNEU		FD	1	Compare Chars Not Equal Update	3-081 B-12
CUIO	v	954C	2	Communicate with Universal I/O	3-099 B-12
DBCD	v	957F	2	Dynamic Binary Convert to Decimal	3-017 B-13
DBFL		A8	1	Dynamic Branch False	3-051 B-13
DBRS		9F	1	Dynamic Bit Reset	3-022 B-13
DBST		97	1	Dynamic Bit Set	3-022 B-14
DBTR		A9	1	Dynamic Branch True	3-051 B-14
DBUN		AA	1	Dynamic Branch Unconditional	3-050 B-14
DEXI	v	9547	2	Disable External Interrupts	3-068 B-14
DFTR		99	1	Dynamic Field Transfer	3-024 B-15
DINS		9D	1	Dynamic Field Insert	3-024 B-15
DISO		9B	1	Dynamic Field Isolate	3-023 B-16
DIVD		83	1	Divide	3-008 B-16
DLAY	v	95F6	3	Delay	3-100 B-16
DLET		B5	1	Delete Top-of-stack	3-064 B-17
DRNT	v	9583	2	Dynamic Range Test	3-010 B-17
DSLF		C1	1	Dynamic Scale Left	3-013 B-17
DSRF		C7	1	Dynamic Scale Right Final	3-015 B-18
DSRR		C9	1	Dynamic Scale Right Rounded	3-015 B-18
DSRS		C5	1	Dynamic Scale Right Save	3-014 B-18
DSRT		C3	1	Dynamic Scale Right Truncate	3-014 B-19
DUPL		B7	1	Duplicate Top-of-stack	3-065 B-19
EEXI	v	9546	2	Enable External Interrupts	3-068 B-19
ENDE	e	DE	1	End Edit	3-099 B-19
ENDF	e	D5	3	End Float	3-097 B-20

Mne	Mode	Op	Syls	Function	References
ENTR		AB	1	Enter	3-054 B-21
EQUL		8C	1	Equal To	3-009 B-22
EVAL		AC	1	Evaluate	3-038 B-22
EXCH		B6	1	Exchange Top-of-stack	3-064 B-22
EXIT		A3	1	Exit	3-059 B-23
EXPU		DD	1	Execute Sngl Edit Op, Sngl Ptr Update	3-094 B-24
EXSD		D2	1	Execute Single Edit Operator Delete	3-094 B-24
EXSU		DA	1	Execute Single Edit Operator Update	3-094 B-25
FLTR		98	4	Field Transfer	3-024 B-25
GREQ		89	1	Greater Than or Equal To	3-009 B-25
GRTR		8A	1	Greater Than	3-009 B-25
HALT	v	95DF	2	Conditional Processor Halt	3-101 B-26
HALT	e	DF	1	Conditional Processor Halt	3-101 B-26
ICLD	v	9575	2	Input Convert Left-Signed Delete	3-089 B-26
ICRD	v	9576	2	Input Convert Right-Signed Delete	3-089 B-26
ICUD		A4	1	Input Convert Unsigned Delete	3-089 B-26
ICVD		CA	1	Input Convert Delete	3-089 B-27
ICVU		CB	1	Input Convert Update	3-089 B-27
IDIV		84	1	Integer Divide	3-008 B-27
IDLE	v	9544	2	Idle Until Interrupt	3-099 B-27
IMKS		CF	1	Insert Mark Stack	3-054 B-28
INDX		A6	1	Index	3-032 B-29
INOP	e	D8	1	Insert Overpunch	3-096 B-30
INSC	e	DD	3	Insert Conditional	3-096 B-30
INSC	t	DD	4	Insert Conditional	3-096 B-30
INSG	e	D9	3	Insert Display Sign	3-096 B-31
INSR		9C	3	Field Insert	3-023 B-31
INSU	e	DC	2	Insert Unconditional	3-096 B-31
INSU	t	DC	3	Insert Unconditional	3-096 B-31
INXA		E7	3	Index Via Address-Couple	3-033 B-32
ISOL		9A	3	Field Isolate	3-023 B-32
JOIN	v	9542	2	Set Two Singles to Double	3-020 B-33
LAND		90	1	Logical And	3-017 B-33
LEQV		93	1	Logical Equivalence	3-018 B-33
LESS		88	1	Less Than	3-009 B-33
LKID	v	95B3	2	Read Interlock Status	3-048 B-34
LLLU	v	95BD	2	Linked List Lookup	3-071 B-34
LNMC	v	958C	4	Long Name Call	3-031 B-35
LNOT		92	1	Logical Not	3-017 B-35
LOAD		BD	1	Load	3-039 B-35
LODT		BC	1	Load Transparent	3-039 B-36
LODT	v	95BC	2	Load Transparent	3-039 B-36
LOG2	v	958B	2	Leading One Test	3-021 B-36
LOK	v	95B0	2	Lock Interlock	3-047 B-36
LOKC	v	95B1	2	Conditional Lock Interlock	3-048 B-37
LOR		91	1	Logical Or	3-018 B-37
LSEQ		8B	1	Less Than or Equal To	3-009 B-37
LT8		B2	2	Insert 8-Bit Literal	3-018 B-37
LT16		B3	3	Insert 16-Bit Literal	3-018 B-37
LT48		BE	7-12	Insert 48-Bit Literal	3-019 B-38

Mne	Mode	Op	Syls	Function	References
LVLC	v	958D	4	Long Value Call	3-036 B-38
MCHR	e	D7	1	Move Chars	3-097 B-38
MCHR	t	D7	2	Move Chars	3-097 B-38
MFLT	e	D1	5	Move with Float	3-098 B-38
MFLT	t	D1	5	Move with Float	3-098 B-38
MINS	e	D0	2	Move with Insert	3-098 B-39
MINS	t	D0	3	Move with Insert	3-098 B-39
MKSN		DF	1	Mark-Stack Bound to Name-Call	3-053 B-39
MKST		AE	1	Mark Stack	3-052 B-39
MPCW		BF	7-12	Make PCW	3-034 B-40
MULT		82	1	Multiply	3-007 B-40
MULX		8F	1	Extended Multiply	3-008 B-40
MVNU	e	D6	1	Move Numeric Unconditional	3-097 B-41
MVNU	t	D6	2	Move Numeric Unconditional	3-097 B-41
MVST	v	95AF	2	Move to Stack	3-062 B-42
NAMC		40-7F	2	Name Call	3-031 B-43
NEQL		8D	1	Not Equal To	3-009 B-43
NOOP		FE	1	No Operation	3-100 B-43
NOOP	v	95FE	2	No Operation	3-100 B-43
NORM	v	958E	2	Normalize	3-008 B-43
NTGD	v	9587	2	Integerize Double-Precision Rounded	3-012 B-44
NTGR		87	1	Integerize Rounded	3-011 B-44
NTIA		86	1	Integerize Truncated	3-011 B-44
NTTD	v	9586	2	Integerize Double-Precision Truncated	3-011 B-44
NVLD		FF	1	Invalid Operator	3-101 B-45
NVLD	v	95FF	2	Invalid Operator	3-101 B-45
NXLN		A5	1	Index and Load Name	3-037 B-45
NXLV		AD	1	Index and Load Value	3-036 B-46
NXVA		EF	3	Index and Load Value Via Addr-Couple	3-037 B-47
OCRX	v	9585	2	Occurs Index	3-025 B-47
ONE		B1	1	Insert Literal One	3-018 B-48
OVRD		BA	1	Overwrite Delete	3-043 B-48
OVRN		BB	1	Overwrite Non-Delete	3-043 B-48
PACD		D1	1	Pack Delete	3-086 B-48
PACU		D9	1	Pack Update	3-086 B-49
PAUS	v	9584	2	Pause Until Interrupt	3-099 B-49
PKLD	v	9573	2	Pack Left-Signed	3-086 B-49
PKRD	v	9574	2	Pack Right-Signed	3-086 B-49
PKUD	v	9572	2	Pack Unsigned	3-086 B-50
PUSH		B4	1	Push Working Stack Onto Activation Rec	3-100 B-50
RDIV		85	1	Remainder Divide	3-008 B-50
RDLK	v	95BA	2	Read Lock	3-044 B-51
REMC	v	9592	2	Read External Memory Control	3-099 B-51
RETN		A7	1	Return	3-061 B-52
RIPS	v	9598	2	Read Internal Processor State	3-067 B-52
RNGT	v	9582	4	Range Test	3-010 B-53
ROFF		D7	1	Read and Reset Overflow Flip-Flop	3-070 B-53
RPRR	v	95B8	2	Read Processor Register	3-067 B-53
RSDN	v	95B7	2	Rotate Stack Down	3-065 B-54
RSNR	v	9581	2	Read Stack Number	3-066 B-54

Mne	Mode	Op	Syls	Function	References
RSTF	e	D4	1	Reset Float Flip-Flop	3-099 B-54
RSUP	v	95B6	2	Rotate Stack Up	3-065 B-54
RTAG	v	95B5	2	Read Tag	3-021 B-55
RTFF		DE	1	Read True-False Flip-Flop	3-066 B-55
RTOD	v	95A7	2	Read Time of Day Clock	3-067 B-55
RUNI	v	9541	2	Indicate Running	3-070 B-55
SAME		94	1	Logical Equality	3-018 B-56
SCLF		C0	2	Scale Left	3-013 B-56
SCRF		C6	2	Scale Right Final	3-015 B-56
SCRR		C8	2	Scale Right Rounded	3-015 B-57
SCRS		C4	2	Scale Right Save	3-014 B-57
SCRT		C2	2	Scale Right Truncate	3-014 B-57
SEQD	v	95F4	2	Scan While Equal Delete	3-079 B-58
SEQU	v	95FC	2	Scan While Equal Update	3-079 B-58
SFDC	e	DA	1	Skip Forward Destination Chars	3-079 B-58
SFDC	t	DA	2	Skip Forward Destination Chars	3-095 B-58
SFSC	e	D2	1	Skip Forward Source Chars	3-095 B-59
SFSC	t	D2	2	Skip Forward Source Chars	3-095 B-59
SGED	v	95F1	2	Scan While Greater or Equal Delete	3-079 B-59
SGEU	v	95F9	2	Scan While Greater or Equal Update	3-079 B-59
SGTD	v	95F2	2	Scan While Greater Delete	3-079 B-59
SGTU	v	95FA	2	Scan While Greater Update	3-079 B-59
SHOW	v	95DE	2	Primitive Display	3-091 B-60
SINT	v	9545	1	Set Interval Timer	3-068 B-60
SISO		D5	1	String Isolate	3-083 B-61
SLED	v	95F3	2	Scan While Less or Equal Delete	3-079 B-61
SLEU	v	95FB	2	Scan While Less or Equal Update	3-079 B-61
SLSD	v	95F0	2	Scan While Less Delete	3-079 B-62
SLSU	v	95F8	2	Scan While Less Update	3-079 B-62
SNED	v	95F5	2	Scan While Not Equal Delete	3-079 B-62
SNEU	v	95FD	2	Scan While Not Equal Update	3-079 B-62
SNGL		CD	1	Set to Single-Precision Rounded	3-011 B-62
SNGT		CC	1	Set to Single-Precision Truncated	3-011 B-63
SPLT	v	9543	2	Set Double to Two Singles	3-020 B-63
SPRR	v	95B9	2	Set Processor Register	3-069 B-64
SRCH	v	95BE	2	Masked Search for Equal	3-072 B-64
SRDC	e	DB	1	Skip Reverse Destination Chars	3-095 B-64
SRDC	t	DB	2	Skip Reverse Destination Chars	3-095 B-64
SRSC	e	D3	1	Skip Reverse Source Chars	3-095 B-65
SRSC	t	D3	2	Skip Reverse Source Chars	3-095 B-65
STAD		F6	3	Store Delete Via Address-Couple	3-042 B-65
STAG	v	95B4	2	Set Tag	3-019 B-66
STAN		F7	3	Store Non-Delete Via Address-Couple	3-042 B-66
STFF		AF	1	Stuff	3-032 B-66
STOD		B8	1	Store Delete	3-042 B-67
STON		B9	1	Store Non-Delete	3-042 B-68
STOP	v	95BF	2	Unconditional Processor Halt	3-101 B-68
SUBT		81	1	Subtract	3-007 B-68
SWFD	v	95D4	2	Scan While False Delete	3-082 B-69
SWFU	v	95DC	2	Scan While False Update	3-082 B-69

Mne	Mode	Op	Syls	Function	References
SWTD	v	95D5	2	Scan While True Delete	3-082 B-69
SWTU	v	95DD	2	Scan While True Update	3-082 B-70
SXSN		D6	1	Set External Sign Flip-Flop	3-068 B-70
TEED		DO	1	Table Enter Edit Delete	3-093 B-70
TEEU		D8	1	Table Enter Edit Update	3-093 B-71
TEQD		E4	1	Transfer While Equal Delete	3-080 B-71
TEQU		EC	1	Transfer While Equal Update	3-080 B-72
TGED		E1	1	Transfer While Greater or Equal Delete	3-080 B-72
TGEU		E9	1	Transfer While Greater or Equal Update	3-080 B-72
TGTD		E2	1	Transfer While Greater Delete	3-080 B-72
TGTU		EA	1	Transfer While Greater Update	3-080 B-72
TLED		E3	1	Transfer While Less or Equal Delete	3-080 B-72
TLEU		EB	1	Transfer While Less or Equal Update	3-080 B-73
TLSD		EO	1	Transfer While Less Delete	3-080 B-73
TLSU		E8	1	Transfer While Less Update	3-080 B-73
TNED		E5	1	Transfer While Not Equal Delete	3-080 B-73
TNEU		ED	1	Transfer While Not Equal Update	3-080 B-73
TRNS	v	95D7	2	Translate	3-084 B-74
TUND		E6	1	Transfer Chars Unconditional Delete	3-078 B-75
TUNU		EE	1	Transfer Chars Unconditional Update	3-078 B-75
TWFD	v	95D2	2	Transfer While False Delete	3-083 B-76
TWFU	v	95DA	2	Transfer While False Update	3-083 B-76
TWOD		D4	1	Transfer Words Overwrite Delete	3-090 B-77
TWOU		DC	1	Transfer Words Overwrite Update	3-090 B-77
TWSD		D3	1	Transfer Words Delete	3-090 B-78
TWSU		DB	1	Transfer Words Update	3-090 B-78
TWTD	v	95D3	2	Transfer While True Delete	3-083 B-79
TWTU	v	95DB	2	Transfer While True Update	3-083 B-79
UNLK	v	95B2	2	Unlock Interlock	3-047 B-79
UPLD	v	9570	2	Unpack Left-Signed Delete	3-088 B-79
UPLU	v	9578	2	Unpack Left-Signed Update	3-088 B-79
UPRD	v	9571	2	Unpack Right-Signed Delete	3-088 B-79
UPRU	v	9579	2	Unpack Right-Signed Update	3-088 B-80
UPUD	v	95D1	2	Unpack Unsigned Delete	3-087 B-80
UPUU	v	95D9	2	Unpack Unsigned Update	3-087 B-80
USND	v	95DO	2	Unpack Signed Delete	3-088 B-81
USNU	v	95D8	2	Unpack Signed Update	3-088 B-81
VALC		00-3F	2	Value Call	3-035 B-81
VARI		95	1	Introduce Variant Operator	3-101 B-81
WATI	v	95A4	2	Read Machine Identification	3-066 B-82
WEMC	v	9593	2	Write External Memory Control	3-100 B-82
WHOI	v	954E	2	Read Processor Identification	3-066 B-82
WIPS	v	9599	2	Write Internal Processor State	3-070 B-82
WTOD	v	9549	2	Write Time-of-Day Clock	3-068 B-83
XTND		CE	1	Set to Double-Precision	3-020 B-83
ZERO		BO	1	Insert Literal Zero	3-018 B-83
ZIC	v	9540	2	Zero Interrupt_Count	3-070 B-83

OPERATORS LISTED BY MODE AND OPERATOR

PRIMARY MODE OPERATORS

Mne	Mode	Op	Syls	Function	References
VALC		00-3F	2	Value Call	3-035 B-81
NAMC		40-7F	2	Name Call	3-031 B-43
ADD		80	1	Add	3-007 B-07
SUBT		81	1	Subtract	3-007 B-68
MULT		82	1	Multiply	3-007 B-40
DIVD		83	1	Divide	3-008 B-16
IDIV		84	1	Integer Divide	3-008 B-27
RDIV		85	1	Remainder Divide	3-008 B-50
NTIA		86	1	Integerize Truncated	3-011 B-44
NTGR		87	1	Integerize Rounded	3-011 B-44
LESS		88	1	Less Than	3-009 B-33
GREQ		89	1	Greater Than or Equal To	3-009 B-25
GRTR		8A	1	Greater Than	3-009 B-25
LSEQ		8B	1	Less Than or Equal To	3-009 B-37
EQUL		8C	1	Equal To	3-009 B-22
NEQL		8D	1	Not Equal To	3-009 B-43
CHSN		8E	1	Change Sign	3-025 B-11
MULX		8F	1	Extended Multiply	3-008 B-40
LAND		90	1	Logical And	3-017 B-33
LOR		91	1	Logical Or	3-018 B-37
LNOT		92	1	Logical Not	3-017 B-35
LEQV		93	1	Logical Equivalence	3-018 B-33
SAME		94	1	Logical Equality	3-018 B-56
VARI		95	1	Introduce Variant Operator	3-101 B-81
BSET		96	2	Bit Set	3-022 B-09
DBST		97	1	Dynamic Bit Set	3-022 B-14
FLTR		98	4	Field Transfer	3-024 B-25
DFTR		99	1	Dynamic Field Transfer	3-024 B-15
ISOL		9A	3	Field Isolate	3-023 B-32
DISO		9B	1	Dynamic Field Isolate	3-023 B-16
INSR		9C	3	Field Insert	3-023 B-31
DINS		9D	1	Dynamic Field Insert	3-024 B-15
BRST		9E	2	Bit Reset	3-022 B-09
DBRS		9F	1	Dynamic Bit Reset	3-022 B-13
BRFL		A0	3	Branch False	3-049 B-08
BRTR		A1	3	Branch True	3-049 B-09
BRUN		A2	3	Branch Unconditional	3-049 B-09
EXIT		A3	1	Exit	3-059 B-23
ICUD		A4	1	Input Convert Unsigned Delete	3-089 B-26
NXLN		A5	1	Index and Load Name	3-037 B-45
INDX		A6	1	Index	3-032 B-29
RETN		A7	1	Return	3-061 B-52
DBFL		A8	1	Dynamic Branch False	3-051 B-13
DBTR		A9	1	Dynamic Branch True	3-051 B-14
DBUN		AA	1	Dynamic Branch Unconditional	3-050 B-14

Mne	Mode	Op	Syls	Function	References
ENTR		AB	1	Enter	3-054 B-21
EVAL		AC	1	Evaluate	3-038 B-22
NXLV		AD	1	Index and Load Value	3-036 B-46
MKST		AE	1	Mark Stack	3-052 B-39
STFF		AF	1	Stuff	3-032 B-66
ZERO		B0	1	Insert Literal Zero	3-018 B-83
ONE		B1	1	Insert Literal One	3-018 B-48
LT8		B2	2	Insert 8-Bit Literal	3-018 B-37
LT16		B3	3	Insert 16-Bit Literal	3-018 B-37
PUSH		B4	1	Push Working Stack Onto Activation Rec	3-100 B-50
DLET		B5	1	Delete Top-of-stack	3-064 B-17
EXCH		B6	1	Exchange Top-of-stack	3-064 B-22
DUPL		B7	1	Duplicate Top-of-stack	3-065 B-19
STOD		B8	1	Store Delete	3-042 B-67
STON		B9	1	Store Non-Delete	3-042 B-68
OVRD		BA	1	Overwrite Delete	3-043 B-48
OVRN		BB	1	Overwrite Non-Delete	3-043 B-48
LODT		BC	1	Load Transparent	3-039 B-36
LOAD		BD	1	Load	3-039 B-35
LT48		BE	7-12	Insert 48-Bit Literal	3-019 B-38
MPCW		BF	7-12	Make PCW	3-034 B-40
SCLF		C0	2	Scale Left	3-013 B-56
DSLF		C1	1	Dynamic Scale Left	3-013 B-17
SCRT		C2	2	Scale Right Truncate	3-014 B-57
DSRT		C3	1	Dynamic Scale Right Truncate	3-014 B-19
SCRS		C4	2	Scale Right Save	3-014 B-57
DSRS		C5	1	Dynamic Scale Right Save	3-014 B-18
SCRF		C6	2	Scale Right Final	3-015 B-56
DSRF		C7	1	Dynamic Scale Right Final	3-015 B-18
SCRR		C8	2	Scale Right Rounded	3-015 B-57
DSRR		C9	1	Dynamic Scale Right Rounded	3-015 B-18
ICVD		CA	1	Input Convert Delete	3-089 B-27
ICVU		CB	1	Input Convert Update	3-089 B-27
SNGT		CC	1	Set to Single-Precision Truncated	3-011 B-63
SNGL		CD	1	Set to Single-Precision Rounded	3-011 B-62
XTND		CE	1	Set to Double-Precision	3-020 B-83
IMKS		CF	1	Insert Mark Stack	3-054 B-28
TEED		D0	1	Table Enter Edit Delete	3-093 B-70
PACD		D1	1	Pack Delete	3-086 B-48
EXSD		D2	1	Execute Single Edit Operator Delete	3-094 B-24
TWSD		D3	1	Transfer Words Delete	3-090 B-78
TWOD		D4	1	Transfer Words Overwrite Delete	3-090 B-77
SISO		D5	1	String Isolate	3-083 B-61
SXSN		D6	1	Set External Sign Flip-Flop	3-068 B-70
ROFF		D7	1	Read and Reset Overflow Flip-Flop	3-070 B-53
TEEU		D8	1	Table Enter Edit Update	3-093 B-71
PACU		D9	1	Pack Update	3-086 B-49
EXSU		DA	1	Execute Single Edit Operator Update	3-094 B-25
TWSU		DB	1	Transfer Words Update	3-090 B-78
TWOU		DC	1	Transfer Words Overwrite Update	3-090 B-77

Mne	Mode	Op	Syls	Function	References
EXPU		DD	1	Execute Sngl Edit Op, Sngl Ptr Update	3-094 B-24
RTFF		DE	1	Read True-False Flip-Flop	3-066 B-55
MKSN		DF	1	Mark-Stack Bound to Name-Call	3-053 B-39
TLSD		E0	1	Transfer While Less Delete	3-080 B-73
TGED		E1	1	Transfer While Greater or Equal Delete	3-080 B-72
TGTD		E2	1	Transfer While Greater Delete	3-080 B-72
TLED		E3	1	Transfer While Less or Equal Delete	3-080 B-72
TEQD		E4	1	Transfer While Equal Delete	3-080 B-71
TNED		E5	1	Transfer While Not Equal Delete	3-080 B-73
TUND		E6	1	Transfer Chars Unconditional Delete	3-078 B-75
INXA		E7	3	Index Via Address-Couple	3-033 B-32
TLSU		E8	1	Transfer While Less Update	3-080 B-73
TGEU		E9	1	Transfer While Greater or Equal Update	3-080 B-72
TGTU		EA	1	Transfer While Greater Update	3-080 B-72
TLEU		EB	1	Transfer While Less or Equal Update	3-080 B-73
TEQU		EC	1	Transfer While Equal Update	3-080 B-72
TNEU		ED	1	Transfer While Not Equal Update	3-080 B-73
TUNU		EE	1	Transfer Chars Unconditional Update	3-078 B-75
NXVA		EF	3	Index and Load Value Via Addr-Couple	3-037 B-47
CLSD		F0	1	Compare Chars Less Delete	3-081 B-12
CGED		F1	1	Compare Chars Greater or Equal Delete	3-081 B-10
CGTD		F2	1	Compare Chars Greater Delete	3-081 B-11
CLED		F3	1	Compare Chars Less or Equal Delete	3-081 B-11
CEQD		F4	1	Compare Chars Equal Delete	3-081 B-10
CNED		F5	1	Compare Chars Not Equal Delete	3-081 B-12
STAD		F6	3	Store Delete Via Address-Couple	3-042 B-65
STAN		F7	3	Store Non-Delete Via Address-Couple	3-042 B-66
CLSU		F8	1	Compare Chars Less Update	3-081 B-12
CGEU		F9	1	Compare Chars Greater or Equal Update	3-081 B-11
CGTU		FA	1	Compare Chars Greater Update	3-081 B-11
CLEU		FB	1	Compare Chars Less or Equal Update	3-081 B-11
CEQU		FC	1	Compare Chars Equal Update	3-081 B-10
CNEU		FD	1	Compare Chars Not Equal Update	3-081 B-12
NOOP		FE	1	No Operation	3-100 B-43
NVLD		FF	1	Invalid Operator	3-101 B-45

EDIT MODE OPERATORS

Mne	Mode	Op	Syls	Function	References
MINS	e	D0	2	Move with Insert	3-098 B-39
MFLT	e	D1	5	Move with Float	3-098 B-38
SFSC	e	D2	1	Skip Forward Source Chars	3-095 B-59
SRSC	e	D3	1	Skip Reverse Source Chars	3-095 B-65
RSTF	e	D4	1	Reset Float Flip-Flop	3-099 B-54
ENDF	e	D5	3	End Float	3-097 B-20
MVNU	e	D6	1	Move Numeric Unconditional	3-097 B-41
MCHR	e	D7	1	Move Chars	3-097 B-38
INOP	e	D8	1	Insert Overpunch	3-096 B-30
INSG	e	D9	3	Insert Display Sign	3-096 B-31
SFDC	e	DA	1	Skip Forward Destination Chars	3-095 B-58
SRDC	e	DB	1	Skip Reverse Destination Chars	3-095 B-64
INSU	e	DC	2	Insert Unconditional	3-096 B-31
INSC	e	DD	3	Insert Conditional	3-096 B-30
ENDE	e	DE	1	End Edit	3-099 B-19
HALT	e	DF	1	Conditional Processor Halt	3-101 B-26

TABLE EDIT MODE OPERATORS

Mne	Mode	Op	Syls	Function	References
MINS	t	D0	3	Move with Insert	3-098 B-39
MFLT	t	D1	5	Move with Float	3-098 B-38
SFSC	t	D2	2	Skip Forward Source Chars	3-095 B-59
SRSC	t	D3	2	Skip Reverse Source Chars	3-095 B-65
MVNU	t	D6	2	Move Numeric Unconditional	3-097 B-41
MCHR	t	D7	2	Move Chars	3-097 B-38
SFDC	t	DA	2	Skip Forward Destination Chars	3-095 B-58
SRDC	t	DB	2	Skip Reverse Destination Chars	3-095 B-64
INSU	t	DC	3	Insert Unconditional	3-096 B-31
INSC	t	DD	4	Insert Conditional	3-096 B-30

VARIANT MODE OPERATORS

Mne	Mode	Op	Syls	Function	References
ZIC	v	9540	2	Zero Interrupt_Count	3-070 B-83
RUNI	v	9541	2	Indicate Running	3-070 B-55
JOIN	v	9542	2	Set Two Singles to Double	3-020 B-33
SPLT	v	9543	2	Set Double to Two Singles	3-020 B-63
IDLE	v	9544	2	Idle Until Interrupt	3-099 B-27
SINT	v	9545	1	Set Interval Timer	3-068 B-60
EEXI	v	9546	2	Enable External Interrupts	3-068 B-19
DEXI	v	9547	2	Disable External Interrupts	3-068 B-14
WTOD	v	9549	2	Write Time-of-Day Clock	3-068 B-83
CUIO	v	954C	2	Communicate with Universal I/O	3-099 B-12
WHOI	v	954E	2	Read Processor Identification	3-066 B-82
UPLD	v	9570	2	Unpack Left-Signed Delete	3-088 B-79
UPRD	v	9571	2	Unpack Right-Signed Delete	3-088 B-79
PKUD	v	9572	2	Pack Unsigned	3-086 B-50
PKLD	v	9573	2	Pack Left-Signed	3-086 B-49
PKRD	v	9574	2	Pack Right-Signed	3-086 B-49
ICLD	v	9575	2	Input Convert Left-Signed Delete	3-089 B-26
ICRD	v	9576	2	Input Convert Right-Signed Delete	3-089 B-26
BCD	v	9577	3	Binary Convert to Decimal	3-016 B-08
UPLU	v	9578	2	Unpack Left-Signed Update	3-088 B-79
UPRU	v	9579	2	Unpack Right-Signed Update	3-088 B-80
DBCD	v	957F	2	Dynamic Binary Convert to Decimal	3-017 B-13
ASRT	v	9580	3	Assert	3-101 B-08
RSNR	v	9581	2	Read Stack Number	3-066 B-54
RNGT	v	9582	4	Range Test	3-010 B-53
DRNT	v	9583	2	Dynamic Range Test	3-010 B-17
PAUS	v	9584	2	Pause Until Interrupt	3-099 B-49
OCRX	v	9585	2	Occurs Index	3-025 B-47
NTTD	v	9586	2	Integerize Double-Precision Truncated	3-011 B-44
NTGD	v	9587	2	Integerize Double-Precision Rounded	3-012 B-44
AMIN	v	9588	2	Arithmetic Minimum	3-009 B-07
AMAX	v	958A	2	Arithmetic Maximum	3-009 B-07
LOG2	v	958B	2	Leading One Test	3-021 B-36
LNMC	v	958C	4	Long Name Call	3-031 B-35
LVLC	v	958D	4	Long Value Call	3-036 B-38
NORM	v	958E	2	Normalize	3-008 B-43
REMC	v	9592	2	Read External Memory Control	3-099 B-51
WEMC	v	9593	2	Write External Memory Control	3-100 B-82
RIPS	v	9598	2	Read Internal Processor State	3-067 B-52
WIPS	v	9599	2	Write Internal Processor State	3-070 B-82
WATI	v	95A4	2	Read Machine Identification	3-066 B-82
RTOD	v	95A7	2	Read Time of Day Clock	3-067 B-55
MVST	v	95AF	2	Move to Stack	3-062 B-42
LOK	v	95B0	2	Lock Interlock	3-047 B-36
LOKC	v	95B1	2	Conditional Lock Interlock	3-048 B-37
UNLK	v	95B2	2	Unlock Interlock	3-047 B-79
LKID	v	95B3	2	Read Interlock Status	3-048 B-34

Mne	Mode	Op	Syls	Function	References
STAG	v	95B4	2	Set Tag	3-019 B-66
RTAG	v	95B5	2	Read Tag	3-021 B-55
RSUP	v	95B6	2	Rotate Stack Up	3-065 B-54
RSDN	v	95B7	2	Rotate Stack Down	3-065 B-54
RPRR	v	95B8	2	Read Processor Register	3-067 B-53
SPRR	v	95B9	2	Set Processor Register	3-069 B-64
RDLK	v	95BA	2	Read Lock	3-044 B-51
CBON	v	95BB	2	Count Binary Ones	3-021 B-09
LODT	v	95BC	2	Load Transparent	3-039 B-36
LLLU	v	95BD	2	Linked List Lookup	3-071 B-34
SRCH	v	95BE	2	Masked Search for Equal	3-072 B-64
STOP	v	95BF	2	Unconditional Processor Halt	3-101 B-68
USND	v	95D0	2	Unpack Signed Delete	3-088 B-81
UPUD	v	95D1	2	Unpack Unsigned Delete	3-087 B-80
TWFD	v	95D2	2	Transfer While False Delete	3-083 B-76
TWTD	v	95D3	2	Transfer While True Delete	3-083 B-79
SWFD	v	95D4	2	Scan While False Delete	3-082 B-69
SWTD	v	95D5	2	Scan While True Delete	3-082 B-69
TRNS	v	95D7	2	Translate	3-084 B-74
USNU	v	95D8	2	Unpack Signed Update	3-088 B-81
UPUU	v	95D9	2	Unpack Unsigned Update	3-087 B-80
TWFU	v	95DA	2	Transfer While False Update	3-083 B-76
TWTU	v	95DB	2	Transfer While True Update	3-083 B-79
SWFU	v	95DC	2	Scan While False Update	3-082 B-69
SWTU	v	95DD	2	Scan While True Update	3-082 B-70
SHOW	v	95DE	2	Primitive Display	3-091 B-60
HALT	v	95DF	2	Conditional Processor Halt	3-101 B-26
SLSD	v	95F0	2	Scan While Less Delete	3-079 B-62
SGED	v	95F1	2	Scan While Greater or Equal Delete	3-079 B-59
SGTD	v	95F2	2	Scan While Greater Delete	3-079 B-59
SLED	v	95F3	2	Scan While Less or Equal Delete	3-079 B-61
SEQD	v	95F4	2	Scan While Equal Delete	3-079 B-58
SNED	v	95F5	2	Scan While Not Equal Delete	3-079 B-62
DLAY	v	95F6	3	Delay	3-100 B-16
SLSU	v	95F8	2	Scan While Less Update	3-079 B-62
SGEU	v	95F9	2	Scan While Greater or Equal Update	3-079 B-59
SGTU	v	95FA	2	Scan While Greater Update	3-079 B-59
SLEU	v	95FB	2	Scan While Less or Equal Update	3-079 B-61
SEQU	v	95FC	2	Scan While Equal Update	3-079 B-58
SNEU	v	95FD	2	Scan While Not Equal Update	3-079 B-62
NOOP	v	95FE	2	No Operation	3-100 B-43
NVLD	v	95FF	2	Invalid Operator	3-101 B-45

BSS ENTRANCE EXAM

The following questions relate to the ALGOL compiler.

1. How many bits in a **WORD**?

2. How many bits in a **BYTE**?

3. How many bytes in a **WORD**?

4. What is the Decimal value of Hexadecimal A27?

5. What is the purpose of the Tag?

6. Can data be normally stored into an odd Tag word?

7. Each program has two basic **STACK** structures.

   _____ which contain data;

   _____ which contain code.

8. Solve the following Hexadecimal operations:

```
   A34          1000
 + 3BC        - FFF
 ------       ------
```

9. Can an ALGOL identifier begin with a number?

10. What is the purpose of the **EXPONENT** field in a floating point **WORD** (type **REAL**)?

11. What is the difference between **INTEGER** and **REAL** words?

12. Logical operators **OR**, **AND**, and **NOT** operate only on bit 0.

TRUE **FALSE**

13. This question has two parts:

(a) What will the following literals produce?
Show your answer as though the value had been stored in a single-precision variable (e.g. **REAL X;**).

For Example:
8"ABCDEF" = 4"C1C2C3C4C5C6"

8"CAB" = _____

80"CAB" = _____

480"C3C1C2" = _____

(b) In the third part of (a) above, what does the **480** before the quoted literal mean?

14. Which of the following statements valid?
Note: **DONE** is a **BOOLEAN** variable.

(a)
```
IF DONE
  THEN
ELSE
  VAL:=0;
```

(b)
```
IF DONE THEN;
```

15. Give an example of an explicitly numbered **CASE** statement.

16. Give an example of an implicitly numbered **CASE** statement.

17. What will happen to a program that executes a **CASE** statement that has an arithmetic expression for the argument that is not within the range of any of the specified cases.

18. Give an example of a BAD GO TO.

19. Must the lower bound of an ARRAY always be ZERO?

20. What is the function of the following:

TRUTHSET

TRANSLATETABLE

21. What is the purpose of the **VALUE** arithmetic **REAL** intrinsic function? For example, **VALUE (TERMINATED)**.

22. What is a **FIB** and what is it used for?

23. Which I/O type is usually more efficient?

SEQUENTIAL **RANDOM**

24. What will A[0] contain after execution of each of the following **REPLACE** statements:

```
REAL I;
I:=4"C1";

a)   REPLACE POINTER(A) BY I FOR 1;


b)   REPLACE POINTER(A) BY I.[7:48] FOR 1
```

25. What is the difference between **FORMAL** and **ACTUAL** parameters?

26. Can a **PROCEDURE** be passed as a parameter to another **PROCEDURE**?

27. What happens when an expression is passed as a "call by name" parameter?

28. What is the purpose of **LEXICOGRAPHIC (LEX)** levels?

29. What **LEX** level will a procedure run at?

30. Will a procedure always run at a **LEX** level higher than the caller?

31. What is the purpose of the **MCP** procedure **BLOCKEXIT**?

32. What is the purpose of the **SHARING** dollar card option?

33. Give an example of an indirect **LIBRARY** procedure call.

34. What statements cause an **INDEPENDENT** process to be invoked?

35. What statements cause a **DEPENDENT** process to be invoked?

36. What is a **PIB** and what is it used for?

37. What is the difference between an **ASYNCHRONOUS** and a **SYNCHRONOUS** process?

38. What is the purpose of the following verbs:

PROCURE

LIBERATE

39. Give an example of the use of the **EXCEPTIONEVENT** task attribute.

40. What is the purpose of an **INTERRUPT** procedure?

41. What is the purpose of an **EPILOG** procedure?

42. What is a **DOPE** vector?

43. Give an example of address equation?

44. What is the purpose of the **OWN** phrase in declarations?

45. Give an example of the use of **ARRAY REFERENCE** variables?

46. What is a **TAG** sort?

47. What is the difference between the following dollar card actions:

SET

RESET

POP

48. What is the purpose of user-defined dollar options?

50. What is the function of the following:

MASKSEARCH

ONES

FIRSTONE

LISTLOOKUP

ARRAYSEARCH

The following questions relate to the COBOL74 compiler.

51. What is the difference between an **INDEX** and a **SUBSCRIPT**?

52. What is the advantage of the **USAGE BINARY**?

53. Is the code generated for handling tables is the same as it is in **ALGOL**?

Yes No

54. How are **FILE** attributes declared?

55. Is it possible to **PERFORM** a paragraph in another **SECTION**?

56. What is the default entrypoint name for a **COBOL74** library?

57. What can change the default entrypoint name for a **COBOL74** library?

58. What causes the creation of a new code segment?

59. What **USAGE** types are their and what internal formats do they have?

60. Where is the operational **SIGN** carried for different numeric data types?

61. What must the compiler generate for arithmetic computations using **DISPLAY** type numeric data items?

BSS EXIT EXAM

REVERSE POLISH NOTATION

Write an equivalent Reverse Polish string for each of the following arithmetic expressions:

1. ((A + (B * C)) - ((B * C) + A)) * D

_____

2. A + B - (C * (D + E))

_____

3. (((A + B) * (C - D)) / Z) + Y

_____

4. (X + Y) * C / ((A - (B + C)) * J)

_____

5. (A + (B * C)) / D

_____

6. (A / B) + (C * D) - (E * ((A + B) / C))

_____

Write an equivalent arithmetic expression for each of the following Polish strings:

1. A B + C D - * X + Y -

_____

2. A B * C / E + F -

_____

3. A B + C - D * F /

_____

4. A B C D E + / - *

_____

BASIC STACK ARCHITECTURE

1. T / F :
More than one process can use the same **SEGMENT DICTIONARY.**

2. T / F:
One process may access more than one **SEGMENT DICTIONARY.** positive calls pass proc & param

3. T / F:
A code segment may be referenced by several **PCWs**, each of which give an entry point into the code.

4. T / F:
The **LEXICOGRAPHIC** level at which a procedure will execute depends on the **LEX** level of the calling procedure. NO

5. T / F:
A procedure running at **LEX** level 3 may declare a procedure which will run at **LEX** level 3.

6. T / F:
The **STACK VECTOR** is an array of data descriptors, each of which describes an area of memory allocated as a **STACK** or **SEGMENT DICTIONARY.** All **STACKs** and **SEGMENT DICTIONARIES**, including the **MCP** are thus described.

7. T / F:
The memory address of the **STACK VECTOR** descriptor depends on the setting of the **D[0]** register.

8. T / F:
The process stack is a record of the current state of execution of a program.

9. T / F: — an address couple contains a source addr & dest addr.
destination address.

10. T / F:
When the processor attempts to execute a program segment that is not resident in memory, an interrupt occurs.

11. T / F: NOT IRW
Almost all data addressing is done relative to a **DISPLAY** register.

12. T / F:
The **LEX** level a procedure will run at is determined at compile time.

15. Draw the **PROCESS** stack for the following program as it would appear at the place pointed to by the arrow:

```
BEGIN
REAL A,B,C;

PROCEDURE PRO (A,B,C);
  VALUE B,C;
  REAL A,B,C;
    BEGIN
    INTEGER I,J,K;
    I:=J:=6;
    K:=((I*C) + (B-J));
    A:=A+K;

--------->

    END OF PROCEDURE PRO;

% Main line of program.
A:=7;
B:=5;
PRO (A,B,C);
END OF PROGRAM.
```

DISPLAY REGISTERS

1. T / F:
   The F register contains an index which, when added to the contents of BOSR, locates a MSCW.

2. T / F:
   The D[2] register points to the same location as does BOSR.

3. T / F:
   The D[1] register always points to a program SEGMENT DICTIONARY.

4. T / F:
   The F register of a central processor points to the last MSCW in the STACK which is active on that processor.

5. T / F:
   Tags are used to address DISPLAY registers.

6. T / F:
   D[2] points to a program's SEGMENT DICTIONARY.

7. T / F:
   D[0] points to the MCP's stack.

8. T / F:
   DISPLAY registers indicate the LEX level a procedure is running at.

9. A procedure is declared at **LEX** level 3 and called at **LEX** level 5. What **DISPLAY** register points to the **MSCW** for this procedure? A

_____

10. What processor register contains the memory address of the most recently built **MSCW**? F

11. T / F:
The top of stack registers (A and B) hold the current object code being executed.

12. Segment descriptors contained in the users program **SEGMENT DICTIONARY** are addressed relative to what processor register?

D(1)

_____

13. T / F:
An instruction is referenced by [SDI:PWI:PSI].

WORD FORMATS

1. T / F:
   An RCW contains a return point in a code sequence.

2. T / F:
   A tag of 8 indicates a PCW.

3. T / F:
   A MOM descriptor is always indexed.

4. T / F:
   The choice of dimensions for an array can affect the number of data descriptors in dope vectors.

5. T / F:
   The TOSCW indirectly contains the S and F register settings for an inactive stack.

6. T / F:
   A MSCW is generated when a procedure is entered.

7. T / F:
   The overlay bit indicates if the data area has been overlayed.

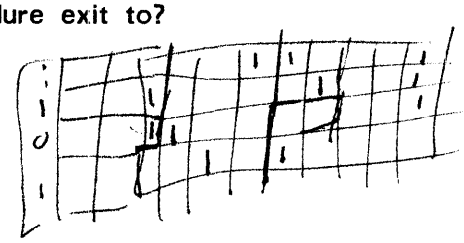
8. T / F:
   Data arrays are not allocated in memory until an item in the array is accessed.

9. T / F:
   It is necessary for the programmer to initialize stack data items to zero, otherwise the item will contain any miscellaneous value.

10. In a MSCW, the displacement field is relative to which processor register?

11. Given the following RCW where will the procedure exit to?

3 500621 894007



SDLL #0 MCP

SDI _____

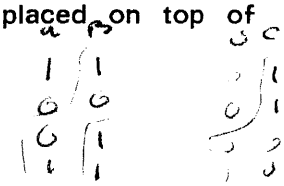
PWI _____

PSI _____

LL 5

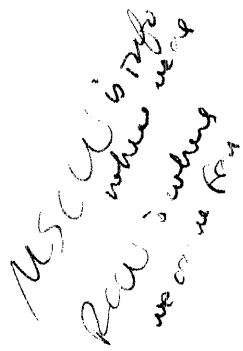
CS 1 (Normal or Control)

12. A process has just caused the following 2 words to be placed on top of stack 15E.



3 008B6D A9B42C RCW

3 15E04A 10C022 MSCW



A. What is the stack number of the procedure we are entering?

15E

B. Is the PCW for the procedure we are entering in this stack?

_____

C. What Lex Level are we going to? 3

_____

D. What is the LEX level of the procedure we came from?

6

_____

E. What is the SDI:PWI:PSI when we left the procedure?

5

_____

13. In a data descriptor, a SIZEF of 7 means what?

_____

14. The first word of a process stack contains either:

TOSCW or processor ID

15. T / F:
An SIRW makes it possible to address memory locations which are not within any stack.

16. T / F: — ed blow upon enter. also a single MSCW at stack bottom
MSCWs and RCWs always occur in pairs.

17. T / F:
When exiting a procedure, if the SDLL bit (bit 13) of the RCW is one, D[1] is selected as the SDI base otherwise D[0] is selected.

18. When are the fields of a MSCW inserted and what are their purposes?

stack #
disp up
environment bit (entered yet)
lex level where it was declared
history — count backwards to prior MSCW in this stack

19. Given the following RCW, where will the procedure exit to?

3 000401 80A003

MCP/USER	USER
SDI	3
PWI	18
PSI	2
LL	2
CS	Normal (Normal or Control)

PROCESSOR OPERATORS

1. T / F:
   The TOSCW is built by the MVST (move to stack) machine instruction.

2. T / F:
   A code syllable is 4 bits long.

3. T / F:
   An ADD instruction adds the top of stack to a word in memory.

4. T / F:
   Segment 0 of a code file is is used to contain information used by the MCP necessary for task initiation.

5. The MVST machine instruction causes the central processor to access a data descriptor at D[0] + 2. To what does this data descriptor point?

   _____ stack vector

6. Generate machine code or assembly language for:

```
X := A [ B [ I , J ] + K ];
```

```
name A
name B
val I
NXLN
val J
NXLV
val call K
add
NXLV
name X
STOD
```

7. What is the missing statement in:

```
BEGIN
REAL X,Y,Z;
----------> X:=Y
```

CODE IS: 1003 1004 B204 81 82 5002 B8 (NON-EMODE)

1003 1004 B204 81 82 F62002 (EMODE)

8. When executing a **MVST** where do we get the following:

A. Stack to go to. TOSCW

B. BOSR of new stack. stack_vector

C. LOSR. stackvector addr +5

D. S register. TOSCW stack height

E. DISPLAY reg settings. clean MSCW

TFFF

9. A program aborted at:

2B:26D:3

Just before the program aborted :

D[1] was set to 113D7

D[2] was set to 11DBF

The contents of several memory locations are given below:

invalid index (negative

ADDRESS								
01843	0	E53100	6527B0	0	1EF50B	01FA31	0	DABDAB DABDAB
113E3	5	080000	2407EB	3	800000	F1552	3	000000 0005C2
113F5	3	800001	122FA1	5	08001B	A40870	3	800002 2215C6
113FB	5	880008	70C441	3	000001	0006A1	3	800000 C16A48
113FE	3	800000	F1420E	3	800002	B15FE2	3	800001 315F9B
11401	3	800003	013FFA	3	800028	A13C8A	3	800001 115CF1
11404	3	000000	0005C3	5	080001	7403B2	3	000000 0006A4
11DC4	5	800001	011DA4	0	000000	000213	0	628000 040000
13DB9	3	41EDBD	95BE60	3	04B99A	2E01A0	3	218B20 029A17
13EF6	3	3BA610	03B810	3	075005	AD700A	3	B8A262 40B3B3
14210	3	CB3453	8B0985	3	BFFFFF	FFFFFF	3	5695A3 9B6C30
16A4A	3	B20380	95BC95	3	B9AE60	2EB208	3	ABA3A3 223F20

Determine why the program aborted.

try this

10. Given the following partial program and code:

Write the ALGOL statements which compiled into the following code:

```
$ SET LIST STACK CODE
BEGIN
REAL A,B,C,D;

PROCEDURE PRO(X,Y,D);
  VALUE X,D;
  REAL X,Y,D;
    BEGIN
    INTEGER I,II;
    LABEL L;


FFB2057006B9 877005B83004 B2058CA10006 7003AC300630 023005300480
100480828OB8 A3B0B0B4A220 00FFFFFFFFFF

     --------->

     --------->

     --------->

          L:
          END OF PROCEDURE PRO;


FFB2035002B8 B2045003B8B1 5004B8B20450 05B8AE500610 02B203805003
AF1004ABA3B0 B0B0B0BFFFFF 00020060E004 B4A22000FFFF

     --------->

     --------->

          END OF PROGRAM.
```