1. GENERAL INFORMATION AND PROGRAMMING HINTS

HARDWAR1	DESCRIPTION OF DEVICES, PSU 360/67
OSHASP	DESCRIPTION OF OS/360 AND HASP WORKINGS
ASPRGTC1	S/360 ASSEMBLER PROG. TECHNIQUES AND MODULARITY
DOCUMENT	S/360 DOCUMENTATION HINTS
LINKAGE	S/360 LINKAGE CONVENTIONS
DSECT	S/360 DSECT EXAMPLE
CS102M1	S/360 OPCODE FAMILIES AND NAMING STRUCTURE

2. SPECIFIC COURSE MATERIALS

CS102TPA	PARTIAL OUTLINE OF INTRO. ASSEMBLER COURSE
CS102HN	HANDOUT LIST FOR INTRO ASSEMBLER COURSE
CS102AS1	ASSIGNMENTS FOR INTRO. ASSEMBLER COURSE
CS102FP1,2,J	FINAL PROJECT AND TEST DECKS
CS411TPA	PARTIAL OUTLINE OF SYSTEMS COURSE
CS411HN	HANDOUT LIST FOR ABOVE
CS411GI1	GENERAL INFORMATION FOR ABOVE
DUMPSJCL, DUMPTEST	INITIAL DUMP INTRODUCTION
CS411AS1	ASSIGNMENTS FOR SYSTEMS COURSE
CS411MC1	MACRO ASSIGNMENTS FOR SYSTEMS
CS411MC2	COURSE
CS411FP1, 2, 3, 4, I, J, K	SYSTEMS COURSE FINAL PROJECT AND TEST DECKS

3. SAMPLE PROGRAMS

FLOTLINK	FLOATING PT. FORTRAN/ASSEMBLER LINKAGE
EXCP	EXAMPLE OF A CHANNEL PROGRAM
BSAM	BSAM I/O EXAMPLE
BPAM	BPAM I/O EXAMPLE
QSAM	QSAM I/O EXAMPLE
OVLY1	OVERLAY STRUCTURE EXAMPLE
PTPCHMAC	BPAM/IBM UTILITY EXAMPLE

PENN STATE UNIVERSITY COMPUTATION CENTER
360/67 CONFIGURATION
this writeup: pages 01 - 04, plus Diagram A (separate).

INTRODUCTION

This writeup briefly describes the devices included in the PSU 360/67 system, and shows how they are connected together. Each device is described below, and diagram A shows the connections.

References are made to DEVICE ADDRESSES. Each individually addressable device (such as a single disk drive, card reader, etc) has a 3 digit (hexadecimal) number which uniquely identifies it to the system, and is used in all input/output operations. The DEVICE ADDRESS is of the following form:

abc	where:
a	gives the CHANNEL NUMBER (from 0 up)
b	specifies a CONTROL UNIT attached to that channel
c	notes which device attached to a given control unit.

Since each digit can have the value 0-F, theoretically it would be possible to attach 16 devices to each of 16 control units attached to 16 channels, for a maximum of 4096 separate devices. In practice, this number is much less, since most S/360's allow a MAXIMUM of 7 channels or less.

The devices follow, more or less in order from the CPU outward.

CENTRAL PROCESSING UNIT

2067-1 (a single 360/67 CPU). uses 200 nanosec (.2 microsec) cycle Read Only Storage (ROS) of 88 bits/word to implement S/360 instruction set (Universal plus special model 67 instructions) includes a HIGH RESOLUTION TIMER (13 microsec cycle). includes a BCU (Bus Control Unit), which is connected to all memory modules, and determines which channel or CPU gets to use a given memory module.

PRIMARY STORAGE

2365 III (4 units) each unit contains 256K bytes. Physically each 2365 contains 2 arrays of 128K bytes, with physical word size of doublewords, i.e., each has 2 arrays of 16K doublewords, and is thus 2-way interleaved at this level. Each 2365 is independent of the others.
CYCLE TIME: 750 nanosec / ACCESS TIME: 375 nanosec

2361 II (1 unit) - Large Core Storage (LCS) - 2048K bytes, organized physically of 2-way interleaved doublewords.
CYCLE TIME: 8000 nanosec (8 microsec) / ACCESS TIME: 3.2 mic

Of the two types of storage, the first contains user programs, and heavily used parts of system programs, while the LCS contains less-used system programs, tables, and buffer areas.

CHANNELS

2870 MULTIPLEXOR CHANNEL - includes 2 SELECTOR SUBCHANNELS (used for magnetic tape drives). generally handles LOW-SPEED devices (card readers, printers, etc)
MAXIMUM TOTAL TRANSFER RATE: 426 KB (kilobytes) per second

2860 SELECTOR CHANNELS - 5 total (2 in 2860 II, 3 in 2860 III). used for HIGH-SPEED devices (disk, drum, etc)
MAXIMUM DATA TRANSFER UNIT, EACH SELECTOR: 1250KB

All CHANNELS and the CPU contend for use of memory modules. The BCU arbitrates among them using a simple priority scheme in following order:

```
          SERVED EARLIER   --->  SERVED LATER
CHANNEL # :    1    2    0    3    4    5    CPU
             drums disk  mx  disk disk  ADAGE
```

The above order is used since the drums cannot wait very long and have the highest transfer rate, the multiplexor channel (0) is fairly early because it may have a large number of things to do, and the CPU is always last because it never hurts it to wait.

CONTROL UNITS

Each control unit can attach to a number of devices, and it is used to control greatly different devices in a such a way as to make them appear more alike, as far as the channels are concerned. Each device must be attached to a particular type of control unit, and each control unit normally can control a group of related devices.

2820 STORAGE CONTROL UNIT - controls the 2301 drum units, attached to channel 1 .

2821 CONTROL UNIT - controls UNIT RECORD devices (card readers, printers, punches). attached to multiplexor channel.

2848 DISPLAY CONTROL - controls the 8 2260 scopes which display system status to the operators.

2701 DATA ADAPTOR - controls a small number of high-speed transmission lines, i.e. high speed terminals (4800 bits/sec transmit rate), such as 360/20's at various locations.

2703 TRANSMISSION CONTROL - controls a larger number of lower-speed terminals, including typewriter/teletype terminals and read/print/punch terminals at Commonwealth Campuses (such as IBM 2780, DCS CP-4, etc).

DISPLAY DEVICES

1052 CONSOLE TYPEWRITER - messages are printed here requiring action by computer operators, and they can enter commands to the system at this location.

2260 ALPHAMERIC DISPLAY SCOPES (8 units) - these display current system status (jobs, disk usage, etc), and also are used to display requests for magnetic tapes to be mounted, etc.

SECONDARY STORAGE - DIRECT ACCESS STORAGE DEVICES (DASDs)

2301 MAGNETIC DRUMS (2 drums) - attached to channel 1 via 2820. Each holds 4.09 megabytes (million bytes) of data, rotates once each 17.5 milliseconds, with average rotational delay (latency time) of 8.6 milliseconds. Records data 4 bits in parallel (for high transfer rate). Has 200 conceptual TRACKS, each of 20,483 bytes maximum size. EACH DRUM IS UNREMOVABLE. MAXIMUM TRANSFER RATE: 1.2 megabytes/second (FASTEST DEVICES USED ON THIS SYSTEM).
These hold most heavily-used compilers and system programs.

231x (2314, 2319) MAGNETIC DISK STORAGE FACILITIES - total of 22 disk drives (including 2 spare ones). Each DRIVE holds one 2316 DISK PACK: 29.17 megabytes maximum, on 20 disk surfaces (11 plates - outside ones not used). Uses MOVABLE HEADS to access information. Each CYLINDER (of which there are 200 usable at any one time) contains the 20 TRACKS accessible at one time without moving the READ/WRITE HEADS. Each track can record at most 7294 bytes of information.
NOTE: unlike drums, each DISK PACK can be removed, and another one mounted in is place if desired.
ROTATION TIME: 25 millisec, AVERAGE LATENCY: 12.5 millisec.
SEEK TIMES (time to move HEADS to correct cylinder):
MIN = 25, AVERAGE = 60 or 75, MAX = 130 or 135 millisec.
MAXIMUM DATA TRANSFER RATE: 312,000 bytes/sec.

NOTE: each of the three storage facilities contains its own control unit, and each drive is numbered accordingly, i.e., 230-237, 330-337, 430-433, on channels 2, 3, 4.

TOTAL DASD STORAGE IS AS FOLLOWS:

2314 (8 drv)	233	megabytes
2319 (8 drv)	233	megabytes
2314 (4 drv)	116	megabytes
2301 (2 drums)	8	megabytes
TOTAL	590	megabytes (approx)

SECONDARY STORAGE - SEQUENTIAL DEVICES

240x (2402 III, 2403 III) MAGNETIC TAPE DRIVES - read/write tape at maximum density of 800 BPI (bits/inch), 9 tracks per tape (2 of the drives also read/write 7-track tapes). Each group of 4 drives is connected to one SELECTOR SUBCHANNEL of the MULTIPLEXOR CHANNEL. The control units for these drives are contained in the 2403 units.
MAXIMUM TRANSFER RATE: 90,000 bytes/sec (90KB), using tape speed of 112.5 inches per second, tape gaps of .6 inch between blocks of data.

UNIT RECORD DEVICES

1403 LINE PRINTERS (of various models), printing with maximum rated speed of 1100 lpm (lines/minute) for 1403 N1, 600 lpm for others. Use removable TRAINs, so that different character sets can be obtained (upper case only: QN, upper/lower: TN). Attached to 2821 control units (on multiplexor).

2540 CARD READ/PUNCH - one unit contains a card reader and card punch (treated logically as separate addresses: for example: 00C for reader, 00D for attached punch).
READS cards (optically) at 1000 cpm (cards/minute) maximum.
PUNCHES cards at 300 cpm maximum.
Attached to 2821 control unit.

2671 PAPER TAPE READER - reads punched paper tape at up to 1000 cps (characters per second). attached also to 2821 control unit.

SUMMARY OF DEVICE CHARACTERISTICS

DEVICE TYPE		CAPACITY PER UNIT (megabytes)	TRANSFER RATE (KB/second)	AVERAGE DELAY (seek)	(latency) ms.
2301	DRUM	4.09	1200	0	8.6
2319	DISK	29.17 per pack	312	60	12.5
2314	DISK	29.17 per pack	312	75	12.5
2400	TAPE DRIVE	varies, 20 per 2400-ft tape OK	90	-	-
1403	PRINTER	132 bytes/line	2.4	-	-
2540	READER	80 bytes/card	1.3	-	-
2540	PUNCH	80 bytes/card	0.4	-	-
2671	PAPER TAPE	--	1.0	-	-

REFERENCES: GA22-6810 IBM S/360 SYSTEM SUMMARY
GA27-2719 IBM S/360 MODEL 67 FUNCTIONAL CHARACTERISTICS

OVERVIEW OF OS/360 WITH HASP

This writeup gives a quick overview of the process by which any OS/360 sytstem is initialized, how storage is used (particularly in OS/360/MVT), and describes how OS/360 is modified by the use of HASP (Houston Automatic Spooling Priority system). The storage layout is described for the PSU CC 360/67 system.

I. INITIALIZATION - getting a system up and running

Consider a computer with no operating system currently in it. The first necessity is to get a workable operating system in it, so that jobs can be run. This is NOT a trivial process: note that there is no Program Fetch resident in the machine, no I/O Access Method routines, and not even a correct set of PSW's in low core for directing interrupt actions.

For OS/360, the initialization process is composed of two parts: IPL and NIP. IPL (Initial Program Loader) initializes memory and some other things, and brings the NUCLEUS (the core of the OS) into memory. NIP (Nucleus Initialization Program) performs the remaining actions required to set up a specific NUCLEUS to be ready to execute.

A. IPL - Initial Program Loader

The process of getting an OS/360 system running is called IPLing, and includes the following main steps:

1. The operator makes sure the disk pack called SYSRES (SYStems RESidence) is mounted on a disk drive. The LOAD UNIT switches are set to show the device address of the SYSRES disk pack, and the LOAD button pressed. This causes the CONTROL RECORD to be read from the first record on the disk pack, consisting of a PSW and two CCW's, placed at location 0 in memory. Execution of this record causes the IPL BOOTSTRAP record to be read into memory. The BOOTSTRAP record consists of a set of CCW's which are used to read the IPL program into memory, beginning at location 0. It ends with a LPSW to give control to the IPL program.

2. IPL selects which NUCLEUS will be loaded (there may be a choice which can be given by switches on the operator console).

3. IPL clears all memory above itself to zeroes, also obtaining the size of memory; i.e., it stores until addressing interrupt occurs.

4. IPL clears the floating point registers, thus finding out if the floating point feature is installed.

5. IPL brings the NUCLEUS into memory. First, it relocates the part of itself not yet executed into high memory (near 252K), so that the NUCLEUS can be placed beginning at 0. It then simulates Program Fetch, loading the csects of the NUCLEUS load module into memory. The first csect loaded is the NIP, loaded just below IPL, followed by the I/O Interrupt Handler at 0 (which thus defines all of the special PSW's in low core). IPL then passes control to NIP.

B. NIP - Nucleus Initialization Program

The IPL process described above applies to all versions of OS/360. The NIP is generated in different ways, depending on the specific type of system and choice of options desired. Note: NIP is a csect which is link-edited with the nucleus, so that it can refer to sections of the nucleus via address constants, and provide efficient and specific initialization services. It includes the following steps:

1. The CVT (Communications Vector TAble) is initialized, and its location placed at location 16, so that it can be accessed from any routine, whether part of the nucleus or not.

2. NIP determines whether the computer has Large Core Storage (LCS) attached to it or not. This is particularly necessary for those systems which include HIARCHY SUPPORT, i.e., the ability to usefully distinguish between main core and LCS, perhaps splitting programs into heavily-used and lesser-used sections.

3. NIP checks the workability of operator console(s), and also checks the workability of ready direct-access devices (using TIO instructions). It particularly checks that the SYSRES volume is mounted and contains certain datasets needed by the system.

4. NIP performs various housekeeping actions, such as checking and setting the timer to make sure it is working correctly, initializing some pointers for storage management, initializing the SVC table (which gives a pointer to each routine associated with a defined SVC number). It also sets up to be able to obtain modules from the SYS1.LINKLIB, which contains the heaviest-used load modules for the system, and also establishes communications with the operator.

5. For any system having one, NIP loads reentrant modules into the LINK PACK. These modules can be used during following execution, and are located at the high end of memory. In a system with fast core+LCS, the LINK PACK can be split, residing at both the high end of fast core and the high end of LCS.

6. With the addition of various other miscellaneous operations, NIP prepares a REGION which will contain the MASTER SCHEDULER, which is the program doing overall job scheduling and operator communication. It then can pass control (LINK or XCTL) to the MASTER SCHEDULER, and the system is finally ready to run jobs.

At this point, memory layout (fast core only) is as follows:

```
HIGH ADDRESS        LINK PACK           (reentrant modules)
                    MS (MASTER SCHEDULER)
                    FREE AREA           (dynamic for problem programs)
                    SQS (SYSTEM QUEUE SPACE) (contains space for system
                                         control blocks - TCB's, etc)
LOW ADDRESS         NUCLEUS
```

NOTE: in systems with HIARCHY SUPPORT, FREE AREA, MS, and LINK PACK would also have areas in LCS.

II. RUNNING JOBS IN AN OS/360 SYSTEM

This section describes how jobs are run in a standard OS/360 system, using either OS-MFT or OS-MVT. Note that OS-PCP runs jobs one at a time (sequential scheduling, uniprogramming), with no SPOOLing of jobs to and from disk before and after execution. OS-MFT and OS-MVT are generally similiar in that they both can SPOOL input onto DASD, execute jobs in priority order, and write the output out later. The main difference is in the handling of storage, in which OS-MVT is much more dynamic. Note that all scheduling of jobs and communication with the operator is effectively under the control of the MASTER SCHEDULER.

A. READING INPUT STREAMS

For each existing input stream (card reader, or input on tape), the operator can issue a START RDR command. This causes a copy of the READER/INTERPRETER program (referred to herafter as a RDR) to read cardimages from the requested input device.

During its operation, a RDR reads an input stream, scans JCL cards and converts them to a standard internal text form, and also obtains cataloged procedure definitions from the procedure library (PROCLIB). From the internal text, it builds INPUT QUEUE entries, representing the information on the user JCL cards. It also writes any input data cards onto disk, while placing pointers to the data into the INPUT QUEUE entries so that it can be found later. The job's INPUT QUEUE entry is enqueued in priority order with other jobs awaiting execution.

When all of the cards for a job have been read, it has in effect been split up into the following:

1. INPUT QUEUE ENTRIES, in priority order, in a special system data set used only for work queue entries, referred to as SYSJOBQUE.
2. INPUT STREAM DATA SETS, placed on DASD, using normal OS/360 Direct Access Device Storage Management (DADSM) routines. NOTE: DADSM routines are themselves kept on DASD, nonresident, and allocating disk space often requires a fair number of accesses to disk to look for free space on one, and to allocate the space appropriately. The DADSM routines are quite general and powerful, but also create some overhead.

B. INITIATING JOB STEPS

The operator may START one or more INITIATORs, each of which can initiate jobs from one or more classes(categories) of jobs. Each initiator will then attempt to initiate the highest priority job from the first class of jobs which has a ready job. If there are no jobs awaiting execution in its allowed classes, it WAITs for one to become available. Note that it essentially removes input queue entries from SYSJOBQUE. Like every RDR, each INITIATOR is executed as a separate task. (INITIATOR may be abbreviated INIT).

When an allowable job becomes available, the initiator obtains a REGION for the job (from the FREE area, also called the DYNAMIC area), uses the information from the RDR to allocate DASD storage, tape drives, and other I/O devices. It then ATTACHes the first module of the program to be executed (thus creating the JOB STEP TASK), and WAITs until the job step completes.

When a job step is finished, the TERMINATOR (part of the INITIATOR really, so that the whole unit is called an INITIATOR-TERMINATOR) effectively cleans up, performing disposition of I/O devices (DISP parameter in JCL), and releasing the REGION which had been acquired for the job step.

During this process, job steps are essentially independent, i.e., they could require different sizes of regions, and might execute in different locations. Note that the INITIATOR-TERMINATOR must also control the skipping of steps as controlled by the JCL COND option.

During execution, SYSOUT datasets are written to DASD, to be printed/punched later. When the last job step of a job completes, the INIT creates a work queue entry calling for the job's output to be printed/punched.

C. WRITING SYSTEM OUTPUT

A program called a SYSTEM OUTPUT WRITER (WTR) can be STARTed by the operator to transcribe output from DASD to printers or punches, or even tapes to be printed/punched later. Output can originally be grouped into CLASSes, which can be written according to priority or otherwise treated differently as desired.

COMMENTS ON THE PROCESS ABOVE

The process described above is quite flexible and general. However, it does require a fair amount of time to set up any job, even a small one. As such, it is quite satisfactory for any installation which runs jobs which require a fair amount of time, since then the setup time is negligible. However, due to the use of OS DADSM for PSPOOLed input and output, DASD space can become fragmented, disk head movement can become excessive, and much time can be used up allocating and deallocating disk space. Although OS/360 is quite reasonable in a commercial installation, or in one running a few large jobs, it seems to have too much overhead for university or other installations which often run many small jobs. For this reason, most larger S/360 computers (i.e., models 75,67,65, and larger 50's) typically use some method to reduce the overhead in running small jobs. All of the methods involve 'faking out' OS/360 in some aspect or other. The method emphasized here (which happens to be the most popular one) is HASP (Houston Automatic Spooling Priority) system.

III. RUNNING AN OS/360 HASP SYSTEM

In any OS/360 system, it is fairly typical to have one or more special jobs in the system, which are loaded before normal user jobs. and typically remain resident from one IPL to the next. Such jobs may control remote batch terminals, timesharing typewriter terminals, or provide any other service which the installation desires. Such jobs are normally placed into the high-address sections of the FREE area (or of the two FREE areas, if the system has both main core plus LCS). When HASP is used, it is normally the first job submitted to OS/360, and it essentially takes over the system, even though it appears to OS/360 as just another job.

A. HASP INITIALIZATION

There are two possible cases when starting HASP up after an IPL. A COLD START occurs when the system is completely empty, i.e., there are no jobs already enqueued on disk which can be executed. If there are disk packs on the system containing previously-read jobs, the start is called a WARM START. A WARM START normally occurs if the system was previously taken down on purpose, such as for systems programming, or if enough information had been saved previous to a 'crash'. A COLD START only occurs when the system crashes badly, and destroys records of jobs already SPOOLed onto disk. In this case, the jobs must be read in again.

When HASP first gains control, it issues a special SVC call, which returns to HASP with protect key 0 and supervisor state, also supplying HASP with some useful pointers to control blocks in the nucleus. NOTE: this special SVC can only be called 1 time, since it locks after its first usage after an IPL.

UCB's (Unit Control Blocks) exist for every device connected to the computer system. HASP now scans these, and essentially allocates to itself:

1. All real unit-record devices (readers, punches, printers).

2. All disk packs which have volume label names beginning SPOOL.

It also obtains effective control of the operator's console(s), plus remote terminals, if any.

Finally, HASP modifies the SVC table (which contains pointers to the routines which are called for each specific SVC number), so that the following ones go into HASP, rather than to the original routines (also saving these addresses for later use for itself):

```
SVC  0    (EXCP - all input/output)
SVC 34    (WTL - write to log)
SVC 35    (WTO, WTOR - write to operator, with/withput reply)
```

B. RUNNING NORMAL USER JOBS UNDER OS/360 WITH HASP

1. Input Stage - HASP continually reads cards from whatever card readers are active in the system. It checks for JOB cards, performs various accounting checks on input jobs, and transcribes the jobs to disk. In this stage, each job is split up into two sections: the JCL cards (with certain modifications), and the input data cards. It enqueues the jobs according to a priority scheme, which can be found from many different sources of information. These include category, time, output, storage requirements, originating site of job, and commands from the operator to change priority of either single jobs or entire groups of jobs. The disk allocation scheme used is quite efficient, and is described later.

2. Execution Stage - HASP has the ability to control which jobs may execute in which portions of the OS FREE area, and using the various priority and storage requirements, it selects jobs from its queue to be executed. One OS RDR exists, permanently STARTed to a card reader. This card reader does not actually exist (i.e., it has a device address which does not correspond to a real card reader). Since SVC 0's are intercepted by HASP anyway, HASP effectively selects a job and feeds it to the OS RDR, which thinks the job is coming across a real card reader. The OS RDR includes an EXIT LIST, which allows it to call some routine after it has scanned each JCL Card, but before the JCL card's data is actually recorded. HASP is entered, and takes this opportunity to modify any JCL that it wishes to, for example, removing any REGION= requests on JOB or EXEC cards. HASP has special treatment for any system input or output data sets, as described below:

//XXXXXXXX DD * or DATA : the OS RDR would normally expect data to follow such a card, and would normally thus SPOOL such to disk itself. HASP does not want this to occur, since it has already SPOOLed the data. It happens that there are large number of UCB's for pseudo card readers already in the system. HASP selects one of these UCB's which is not being used, and effectively changes the tables for this type of card so that it appears as:

//XXXXXXXX DD UNIT=xxx

As a result, the OS RDR thinks that the data set will be read from unit xxx, so that it does not try to SPOOL the input. In any case, the input no longer follows that JCL card, because HASP feeds the RDR only the JCL cards of a user job. During this process, HASP connects up the device address xxx to the specific input data set which had been previously SPOOLed.

//XXXXXXXX DD SYSOUT=x : HASP also has a large number of UCB's for nonexistent, pseudo printers/punches. It does the same thing to this kind of card as it does to DD * cards, except that it only allocates the pseudo devices, and will later save the output which is written to them.

As soon as the RDR finishes reading a job, an initiator can immediately initiate it, since HASP chooses jobs appropriately. When the initiator chooses i/o devices, it finds that it can always allocate devices for unit-record i/o, since HASP had already checked to make sure a pseudo reader/printer/punch was available for each SYSIN or SYSOUT data set.

Finally, a job step of the user job executes. When it wishes to read cards or print lines, it acts as though it were using a real device attached to the system, and OS/360 accepts this. Whenever an SVC 0 is issued to request such I/O, HASP intercepts it.

HASP may be entered for any of the following reasons:

1. WTO, WTOR, WTL - HASP adds own processing as desired.
2. I/O to disk, drum, tape, terminals, etc - HASP does not interfere, but passes these on to the real I/O Supervisor.
3. I/O to real unit-record devices - these have probably been issued by HASP in the first place, so it passes control to the real I/O routines to let them perform the I/O.
4. I/O to a pseudo device - these must be caused by user program. For input, HASP fetches the cardimages from disk into memory (if they are not already present), and feeds requested cardimage(s) to the user program by MVCing them there (using user protect key for safety). For output, it blocks up output and eventually writes it to disk. In all cases, HASP simulates the effect of having real card readers/printers/punches, which are odd only in possessing great speed; i.e., the effect on OS/360 is of having issued an I/O request and having had it complete immediately.

During execution, HASP can also provide extra services, such as monitoring time used, output records, etc. It also reorders priorities of executing user tasks so that I/O bound jobs have higher priorities than do CPU-bound ones. This action (which is unknown to OS/360) helps minimize time spent waiting .

3. Output Stages - Print and Punch - after a job has been executed, it enters the Print queue, is printed, enters the Punch queue, and has punched output (if any) actually punched. This activity occurs without the knowledge of OS/360, which believes the job disappeared whenever it finished execution. Only when a job is finished punching is its disk space released. This allows for jobs to be saved across system crashes, and for such useful services as repeating output by operator control.

C. DASD STORAGE MANAGEMENT IN HASP

HASP manages its DASD storage quite efficiently, not only needing NO accesses to DASD to allocate or deallocate space, but also doing a good job of minimizing arm movement on moveable-head devices. HASP requires the use of entire volumes (normally 2311 or 2314 disks). For example, the PSU CC's 360/67 has 3 2314 disk packs for HASP. The management of this storage works as follows:

A MASTER CYCLINDER BIT-MAP is maintained in HASP. This is a string of bytes, in which each bit represents 1 CYLINDER on the SPOOL disks (for example, 600 bits for the cylinders on 3 packs). A one-bit represents a FREE CYLINDER, while a zero-bit shows that the given cylinder is allocated to some job. HASP also remembers for each disk which cylinder was the last referenced, thus always noting the current position of the read/write heads.

Two JOB BIT-MAPS exist for each job, one for SYSIN data and the other for SYSOUT data. Whenever a cylinder is required for a job, HASP searches for a free one in the following fashion:

1. It first searches the master bit-map for a free cylinder at the current position of any read/write head, i.e., where it can read or write without even moving a head.
2. It then searches for a free cylinder at +1 from current head positions, then -1 from each, followed by +2, -2, etc up to +8, -8 cylinders away from current head position.
3. If the above fail, it searches sequentially through all cylinders in the master bit-map.

When a cylinder is found, its bit is turned off in the master bit-map, and turned on in the appropriate job bit-map. The overall effect of this process is to minimize head movement.

When disk storage for a job is to be released, the deallocation scheme is extremely fast and efficient: the job bit-maps are just OR'd into the master bit-map, thus returning all of the cylinders to free storage.

IV. OTHER PSEUDO-DEVICE SYSTEMS FOR USE WITH OS/360

The following are other systems which are based on OS/360, but use some kind of pseudo-devices to make it run faster.

A. ASP - ATTACHED SUPPORT PROCESSOR

In this system, 2 computers are used. All unit-record devices are attached to the multiplexor channel of a medium-sized 360, along with some disk. It performs all SPOOLing, control of remote terminals, etc. It is connected to a larger system via a chnnel. OS/360 is in the large system, and it reads its input and sends its output along the channel-channel hookup between the two CPU's. A typical setup would use a 360/50 hooked to a 360/75.

An advantage over HASP is that ASP offers somewhat better setup facilities for optimizing use of tapes and non-SPOOL disks. A disadvantage is the requirement of two CPU's, either of which may have problems, and thus stop the entire system.

B. LASP (LOCAL ASP) or CLASP (CLOSELY LINKED ASP)

These are versions of ASP in which the code from the smaller computer is moved over into a region on the larger machine. This allows an ASP system to be run on one processor. If the system is also run under straight ASP, it requires switches to switch the unit-record devices over to the bigger machine. It also requires more memory than HASP, but does allow the system to run even with one CPU down.

C. TUCC HYPERDISK

This method uses LCS plus part of a 2314 disk pack to simulate the entire disk pack containing heavy-used systems programs. The most recently used tracks of this disk are kept in LCS, thus making the disk effectively faster, without changing the internals of OS/360.

V. PSU CC 360/67 SYSTEM - OS/MVT WITH HASP

The following tables gi$e the current layout (with no guarantee of future appearance) as of 6/12/72, for the 360/67 at the PSU CC. The system has both fast core (1024K) and Large Core Storage (2048K).

	LOW	HIGH	K	LOW	HIGH
MS	2928	3072	144	2DC000	300000
HASP	1968	2928	960	1EC000	2DC000
FMGR	1628	1968	340	197000	1EC000
RJE	1346	1628	282	150800	197000
WATFOR	1336	1346	10	14E000	150800
RASP	1236	1336	100	135000	14E000
FREE	1024	1236	212	100000	135000
MS	964	1024	60	0F1000	100000
HASP	876	964	88	0DB000	0F1000
RDR	866	876	10	0D8800	0DB000
FMGR	852	866	14	0D5000	0D8800
RJE	832	852	20	0D0000	0D5000
WATFOR	704	832	128	0B0000	0D0000
FREE	122	704	582	01E800	0B0000
NUC	0	122	122	000000	01E800

NOTES

MS (Master Scheduler) includes the Link Pack areas. The fast core section contains mainly modules for the various I/O Access Methods, while the LCS part contains reentrant parts of INITIATORS, RDRS, plus other routines (overlay supervisor, special tables,etc).

HASP Fast core section is most heavily-used sections. LCS part has lesser-used sections, plus such items as in-core SYSJOBQUE (HASP intercepts all RDR and INIT reads/writes to SYSJOBQUE, and keeps such information in about 600K of LCS). Also has HASP buffers for all devices, plus tables of tape names/locations for user tapes.

FMGR File Manager - manages, synchronizes RJE, BAT files.

RJE Remote Job Entry - handles typewriter terminals.

WATFOR WATFOR REgion - RPSS - manages CAtegory W fast processors swapped in and out of memory (WATFOR, ASSIST, PL/C, etc).

RASP Interface between 360/67 and ADAGE AGT/30 graphics computer.

FREE fast core - 560K for user programs (4x140, 2x280, 1x280+2x 140, occasionally 1x560), rest for Sytem Queue Space.
LCS - currently unused, except for systems programs.

S/360 Assembler Language Programming Techniques

John R. Mashey - Winter l972

Topic: Program Modularity and Parametrization Methods:
Using Macros, Internal Subroutine, External Subroutines
This topic: pages 01-08

It is generally important in any computer program to avoid coding any procedure more times than necessary. It is generally best to write something one time, then have it available for later use in many parts of a program. In assembler language, there are three main ways of doing this: macros, internal subroutines, and external subroutines. This writeup describes each of these techniques, gives the advantages and disadvantages of each, and notes under what condition each is best.

I. DESCRIPTION, DEFINITION, AND CALLING

A. MACRO INSTRUCTIONS

A macro instruction is defined, and either placed at the beginning of an assembly language program (a USER macro), or entered into a macro library (a SYSTEM macro). When called, it generates 0 or more assembly language statements at the point of invocation, and the code generated may vary greatly from call to call.

1. DEFINITION

A macro definition begins with MACRO, followed by the PROTOTYPE STATEMENT, which gives the name of the macro. The body of the macro includes 0 or more MODEL STATEMENTS, which are assembler commands and machine instructions to be generated, and macro-operations, which serve to direct the expansion processing of the macro. The macro definition is terminated by the MEND statement. The following steps are typical in defining a macro:

a. DETERMINE BASIC PURPOSE AND GENERATED CODE: It is generally a good idea to write at least some of the statements to be generated together as a code segment first, to get some feel for what is needed.

b. DECIDE ON NECESSARY ARGUMENTS AND THEIR USAGE: it may be a goo idea to write the purpose of each argument in the operand list, punch it and include it in a block of comments at the beginning of the macro. This helps the macro to be done to do what it is supposed to do.

Use POSITIONAL operands for heavily-used arguments, i.e., if an argument MUST be supplied every time, make it positional. In a group of positionals, place the most heavily used ones near the front, since it is much more convenient to omit the later ones than the earlier. Use KEYWORD arguments for values which may not be needed always, or for ones which are conveniently supplied with default values which are most often used. Use SUBLISTS or &SYSLIST for variable numbers of arguments.

c. WRITE ACTUAL BODY OF MACRO, BUILDING MACRO-TYPE COMMANDS AROUND THE MODEL STATEMENTS TO BE GENERATED.

2. INVOCATION

A macro can be called merely by writing its name and supplying it with any needed arguments. Note that a label on a macro call is never generated (and is thus UNDEFINED) unless the macro definition is made to generate it on some model statement.

B. INTERNAL SUBROUTINES

Internal subroutines are sections of code written as parts of a given control section (CSECT), and are only used inside that CSECT. Like external subroutines, internal subroutines can of course call others. They are typically used for small to medium sections of code which are needed at several places in a CSECT, but are not needed by any other, or are not big enough to warrant the overhead in making them external subroutines.

1. DEFINITION

It is often typical to place a group of internal subroutines near the end of the code section of a program (just before the data areas). It is a good idea to set up conventions for the use of internal subroutines, before writing any. The following are often needed: return register (either one standard one, or several different ones), argument registers, and work registers which can be used without saving. In general, internal subroutines should not need to do much saving and restoring of registers. They should be able to return via BR REG.

2. INVOCATION

Calling an internal subroutine is usually done by first filling any argument registers with needed values, then coding: BAL REG, INSUB. This type of linkage can be fast and small.

C. EXTERNAL SUBROUTINES

External subroutines are used for major program segments, and can usually be assembled separately from the rest of the program. In fact they can be written in a different language (i.e., FORTRAN and ASSEMBLER combinations).

1. DEFINITION

An external subroutine may be written in either of two ways in assembly language: as a CSECT, or as an ENTRY within a CSECT. In the first case, the subroutine is entered at the CSECT statements and return at one or more places depending on the desired code. In the second case each entrypoint may be given control, and may share code or be totally separate from the other entries. This form is often used for a group of related routines (like SIN and COS, which are both entries in a CSECT), or for a routine requiring initialization or termination functions different from the normal calling function.

A multiple-entry CSECT is typically set up as follows:

```
CSECTNAM  CSECT
          ENTRY ENTRY1,ENTRY2,...ENTRYN
..... code for entry at CSECTNAM:   multiple-entry  routines  often
..... are entered only at the  entry  points,  not  at  the  CSECT.

ENTRY1   LINKAGE CODE  (SAVE, XSAVE, etc)
..... executable code when called at ENTRY1.....
          RETURN LINKAGE CODE  (RETURN, XRETURN, etc).

..... remaining entrypoint names and code

..... internal subroutines needed by more  than  one  entry  point.

..... data areas used by  various  of  the  entry  point  routines.
```

The following are important points to remember when using multiple entry CSECTS:

THE DIFFERENT ENTRY POINTS NEVER CALL EACH OTHER. In essence, all of the routines represented by the various entry points are at the same level in calling structure of an entire program.

ONLY ONE SAVE AREA IS ACTUALLY NEEDED. Since the routines inside the CSECT never call each other, the user can code the save area at the end of the LAST section of code, so that all of the previous sections can refer to it (note that if placed on the first, it would be difficult for the later ones to access it using a LA instruction: address constants must be used instead). With XSAVE/XRETURN, this means that the SA=* operand is coded only on the LAST XRETURN.

CARE MUST BE TAKEN WITH ADDRESSIBILITY. All of the code sections can of course address the data areas at the end of the CSECT. However, the programmer must be very careful with any internal subroutines he writes, because the BASE REGISTERS USED TO ASSEMBLE INTERNAL SUBROUTINES MUST HAVE THE CORRECT VALUES IN THEM AT EXECUTION TIME. IF THEY DON'T, AS WHEN THEY ARE CALLED FROM DIFFERENT SECTIONS HAVING DIFFERENT USING SETUPS, THEY WILL ASSEMBLE PROPERLY AND THEN BLOW UP AT EXECUTION TIME. IN PARTICULAR, THE PROGRAMMER SHOULD PLACE INSTRUCTIONS TO BE EXECUTED (EX operation) WITH THE SECTION OF CODE USING THEM, AND NOT AT WITH THE DATA AREAS, IF THEY PERFORM ANY SYMBOLIC ADDRESSING.

The problems described above are typically handled either by making all entry point code segments set up the same USING conditions, or by setting a specific register to point to the beginning of the internal subroutines, EXecuted instructions and data. If register 13 points to a save area just above these code sections, it can be used this way, since it will always have that same value. Getting the same USING conditions across an entire multi-entry CSECT can be done:

```
ENTRYX    XSAVE
          L     BASEREG,=A(CSECTNAM)
          USING CSECTNAM,BASEREG
```

Note that the above can be accomplished with the XSAVE AD= operand.

D. COMBINED FORMS

In some cases, it is convenient to combine the ease of use of the macro with the small size of internal or external subroutines. In this case, the macro expansion sets up any needed arguments, saves registers, etc, then generates code to invoke the subroutine. The subroutine then provides the major portion of the processing code, any needed large data areas, etc.

Examples of the combined form are the following macros: XDECI, XDECO, XPRNT, XSNAP, which call XXXXDECI, XXXXDECO, XXXXPRNT, and XXXXSNAP, respectively.

Two different extremes exist in writing combined forms:

1. COMBINED FORM - STANDARD LINKAGE

In some case, the calling sequence to invoke an external subroutine essentially includes the CALL macro or equivalent code, i.e., it uses standard conventions. It typically assumes that registers 0, 1, 14, 15 may be modified without causing trouble. This method is efficient and general, but can cause trouble if used improperly.

2. COMBINED FORM - SPECIAL NONDESTRUCTIVE LINKAGE

In some cases, it may be useful to define a macro instruction which invokes a subroutine, but can be used ANYWHERE without disturbing any registers, changing the condition code, or requiring that certain of the registers not be the ones being used as base registers (in particular, register 15). This is the kind of linkage used from XDECO to XXXXDECO XPRNT to XXXXPRNT, etc. The following shows the general form of such a linkage setup, giving first the kind of code to be generated by the macro part, then the entry and exit code for the associated routine: (NOTE: label is typically an &SYSNDX-generated unique label)

```
         STM  14,0,label          save registers to be changed
         .... evaluate arguments of macro:  any required Load Addresses
         .... must be done using LA 0, argument since doing LA into
         .... any other register could destroy a base register.  If
         .... more than one argument is needed, the remaining ones can
         .... be stored into control block after label.  Examples:
         LA   0,argument
         ST   0,label+12          2nd argument (one arg left in RO)
         .... after all arguments are evaluated and saved, and ONLY
         .... THEN, it is now possible to modify registers:
         L    15,label-4          V-type adcon for routine
         CNOP 2,4                 make sure next inst not on F boundry
         BALR 14,15               call routine, also point 14 at the
                                  argument list following
         DC   V(subroutine entry point) adcon to get there
label    DS   3F                  3 words for saving 14, 15, 0
         DS   F                   space for arguments after first
         .... DS OR DC space here for any remaining arguments
         .... the subroutine will return control to next instruction:
         LM   14,0,4(14)          reload registers.  Note that this is
                                  only safe way, since 15 might have
                                  current base register.
```

The following shows the typical code used to enter and exit the supporting module used with the previous macro expansion. Note that the entry point of the routine might be either a CSECT name, or an ENTRY name, i.e., one CSECT might contain several entrypoints, one for each supporting subroutine needed.

```
entrypoint label definition (CSECT, or label DS OH)
          USING  entrypoint,15      initial base register
          .... save all registers which may be modified by code.  Save
          .... into THIS CSECT (unlike normal OS/360 conventions).
          .... DO NOT SAVE INTO CALLER'S SAVE AREA, since it may not
          .... exist, especially if caller is a lowest-level routine.

          .... initialization code:  if this routine performs I/O, or
          .... calls any others, or requests any supervisor services, it
          .... is a good idea to set up another base register than 15,
          .... set up a save area, and put its address into register 13,
          .... since any of the above actions may result in registers
          .... being saved at wherever 13 points.

          .... processing code to perform required actions

          .... result return code:  result may be left in register 0,
          .... in which case it should not be restored (neither here nor
          .... in generated code before:  i.e., change LM 14,0,4(14) to
          .... LM 14,15,4(14) and STM likewise).

          .... register restoration:  restore all registers modified in
          .... this routine.  Especially restore 14.

          SPM  14                   restore original condition code (note
                                    that calling BALR 14,15 saved it)

          B    number(14)           branch to displacement number beyond
                                    address in 14, enough to pass control
                                    to statement:  LM 14,0,4(14)
```

It may be useful for the programmer to create a DSECT which describes the control block generated by the macro expansion. This would permit the module to refer to arguments and return points using symbols rather than absolute displacements. A typical DSECT might be:

```
dsectnam DSECT
         DS    V(routine)          space for adcon
         DS    3F                  space for regs 14, 15, 0
argument DS    F                   argument value placed here, if any
         .... further arugment DS statements follow.
return   LM    14,0,4(14)     return label (YES, THIS IS LEGAL:  it does
                              NOT generate code, but it makes the point
                              clear as to description of block).
```

If such a DSECT were used, the routine code would include:

```
         USING dsectnam,14     to set up DSECT addressibility
         B     return          return (instead of B number(14)
```

II. ADVANTAGES AND DISADVANTAGES

The following lists the good and bad points of each type:

A. MACRO INSTRUCTIONS

1. ADVANTAGES

Code can be tailored to each individual request, i.e., the code generated by each macro call can vary from a great deal to nothing, such as debug code eliminated by testing a global set variable.

SPEED: macro-generated code can be the fastest in execution, since it can perform its actions without having to set up linkage to another section of code.

VARIABILITY: generated code can vary depending on the nature of arguments passed to a macro (such as testing the TYPE of arguments to generate different instructions).

2. DISADVANTAGES

SLOW ASSEMBLY: macro processing can be very slow.

LARGE CODE: if used improperly, macros can generate large amounts of code very easily. If there are many copies of large blocks of code, much space can be wasted.

OBJECT DECKS: a macro cannot be assembled and an object deck of it gotten like a subroutine can, i.e., if a call is made to a macro, the macro definition must be included in the program or in a library, while a CSECT may be saved as an object deck (which is usually much smaller than the source deck).

B. INTERNAL SUBROUTINES

1. ADVANTAGES

SPEED: although not as fast as in-line code from a macro, the code for an internal subroutine is usually faster than the linkage to an external one. In particular, values can be passed in registers, and usually registers will not have to be saved.

SPACE: internal subroutines require less space than generating the same code several times via macro expansions.

2. DISADVANTAGES

SPACE: if the same function is performed by internal subroutines in several CSECTS, code is thus duplicated and space wasted.

COMPLEXITY: in some cases, in order to make efficient use of a number of internal subroutines, it is necessary to set up fairly extensive rules on usage of registers in a CSECT, so that the linkage among them may be fast and small.

C. EXTERNAL SUBROUTINES

1. ADVANTAGES

SPACE: if written as an external subroutine, code can be usefully called from almost anywhere in a program. Thus, there is only one copy of it, and it generally will occupy the least space.

SEPARATE COMPILE/ASSEMBLY: a routine written as a CSECT can be assembled separately from the rest of the program an object deck can be obtained, and translation time generally saved. The routine may of course be written in a different language than the rest of the program.

2. DISADVANTAGES

LINKAGE TIME: if standard OS/360 linkage is followed, a fair amount of execution time and object code space can be consumed by this linkage. More efficient nonstandard linkage can be used instead, but this brings with it the disadvantage of nonuniformity and lack of generality.

D. COMBINED FORMS

1. ADVANTAGES

In general, the combined forms can possess all the advantages of the separate forms especially since the macro portions can generate different code depending on circumstances; thus the code for the same macro might expand in-line in one case and generate an out-of-line call to a routine in another.

2. DISADVANTAGES

COMPLEXITY: it of course requires somewhat more planning and code to set up a good combined form system, since both a macro and module must be created and meshed together properly.

III. CIRCUMSTANCES FAVORING USE OF THE VARIOUS FORMS

A. MACRO INSTRUCTIONS

In general, a pure macro instruction is used as follows:

VARYING CODE: the required code varies radically from call to call. For example: XSAVE and XRETURN.

SHORT CODE: if a macro can generate less in-line code to perform the required function than is needed to generate a call to the routine, then it should be written as a macro. In some cases, it takes as much work to set up the arguments as it does just to perform the operations. For example: the code to obtain the minimum or maximum of several arguments is probably most efficiently written as a in-line macro.

LINKAGE CODE: code for linking to routines is almost necessarily written as macros, since it makes little sense to call a routine in order to perform linkage, unless the linkage code required is very complex (in which case the program is probably going to be SLOW).

B. INTERNAL SUBROUTINES

Internal subroutines are usually used (as opposed to macros which generate code in-line) under the following circumstances:

CODE WITH LITTLE VARIANCE: if the code is not going to be much different from macro call to macro call, it may be better to let the macro call generate a BAL to one copy of the code as an internal subr.

Internal subroutines are usually used (as opposed to EXTERNAL subroutines) under these circumstances:

SHORT CODE, HEAVILY USED: if code must be used many times by a CSECT, then the faster linkage of internal subroutines usually makes it worth writing it that way.

CODE NEEDED ONLY BY ONE CSECT: if not too long, it is fairly logical to incorporate it as part of that CSECT. It will probably be much more efficient since it will have access to the internal variables of the CSECT, and be able to communicate via register values easily, rather than requiring long operand lists.

C. EXTERNAL SUBROUTINES

LONG CODE: if something is long and complex enough, it may be a good idea to make a separate module of it, test it, get an object deck, then leave it along thereafter.

CODE OF GENERAL USE, NEEDED MANY PLACES: in this case, it is practically necessary to make code an external subroutine, so that it can be accessed where needed.

D. COMBINED FORMS

These are useful anywhere the others are. The nondestructive form is specially useful if it is to be used by beginning programmers.

S/360 Assembler Language
Documentation and Listing Techniques

by John R. Mashey and Andrea Rhodes

Goals of Good Documentation :
1. Aid in designing good programs
2. Aid in debugging programs
3. Make programs clear and understandable once written
4. Make structure of program well-organized

Good documentation is a great aid to producing clear, well-written, and understandable programs, and can save much programming and computing time. Good documentation is especially necessary for programming projects requiring either a long period of time by one programmer, any period of time by more than one programmer, or modifications to any code by anyone other than the original author. Good documentation techniques can be helpful in the following ways:

PROGRAM DESIGN

Many beginning programmers seem to write programs in haphazard and unplanned ways, and often add comments only after the program is running. This method not only leads to poorly-structured programs, but usually results in wasted time, and is not feasible except for relatively trivial problems.

A much better method is to write most of the overall comments with a flow chart first, specifying the structure and conventions of the program, and then writing the program to fit. This usually leads to cleaner-coded, well-structured programs which are produced in less time than those written by most novice programmers.

PROGRAM DEBUGGING

Program debugging is aided by documenting a program before and during its creation, rather than afterward. Many mistakes can be avoided by having programming conventions well-specified before writing the code. The very act of adding a comment to a statement often helps identify errors in the statement, because it forces the programmer to think about the function of the statement. Finally, good documentation is useful if help is required from someone else, since it aids one in understanding the program quickly. (It also makes other people much more willing to look at a program!)

PROGRAM MODIFICATIONS

Clear and complete documentation is absolutely invaluable when a program must be modified, especially if anyone but the original programmer is making the changes. It may be noted that useful programs tend to be modified often.

ASSEMBLY LANGUAGE DOCUMENTATION

The following advantages apply to any computer language. However, they are most important for assembly language, for the following reasons:

1. Assembly language programs typically require many more statements than do high-level language programs for the same task.
2. Assembly language programs are not usually self documenting. Without good documentation, not even the programmer who wrote the code will be able to understand it several months later.
3. Assembly language programs are often very sensitive to minor changes, much more so than higher-level languages.

The remainder of this paper describes a well-documented assembly program, and notes the various techniques which can be used to achieve this result. Briefly, a well-documented program has the following characteristics:

1. The documentation structure mirrors the program structure, and it leads from the general to the specific. Thus, the program begins with a block of comments which describes the overall purpose of the program, and gives some indication of the general structure. Each major section has a block of comments describing it, as does each of the section's subsets.
2. At least 95% of machine-instruction statements have comments.
3. The program is easy to read, and blocked off into logical sections, so that anyone may look at it and understand it easily.
4. Good programs typically have 15-25% of the total statements as comment cards, in addition to the comments on the individual statements.

S/360 ASSEMBLER DOCUMENTATION HINTS--DO'S and DON'TS

DON'T

punch statements in random columns. This makes a program very unreadable. Use a drum card, and if you do not know how, ask your assistant. The following is a defacto standard for S/360 Assembler statements:

Col. 1 : LABELS
Col. 10: OPERATION CODES
Col. 16: OPERAND FIELD
Col. 36: COMMENTS (col. 40 is preferred by some people)
Col. 72: CONTINUATION COLUMN
Col. 73-80: SEQUENCE NUMBERS (very useful--ask your assistant how to sequence a deck if you are unsure)

This layout can be obtained by the use of the following drum card:
Cols. 1,10,16,36,73: punch '1' (gives tab stops at these cols.)
Col. 72: punch '-' (skips col. 72 automatically, unless AUTO DUP/SKIP is off)
All other columns: punch 'A'

If for some reason these columns are not wanted, a standard set should be decided upon, and then held to completely.

DON'T

Place a comment card before every statement. This bad habit makes programs absolutely unreadable. Embedded comments should be used to block programs into logical sections, not to explain the function of individual statements.

DON'T

bury code with too many interspersed comments. If so many comments are necessary, place them in blocks ahead of the program segments and not in the middle.

DO

put a comment on nearly every machine instruction. Comments are also helpful for explanations of variables and flags. Each comment should describe the function of its statement, and generally, it alone. If a comment is needed to describe the function of a block of half-a-dozen cards, it probably should be placed on a comment card preceding the block of code. These comments should be punched when the program is originally punched. A good technique is to add these comments while keypunching the program. Often, this results in catching many mistakes at that point. It is noted that few novice programmers do this, while most experts do. It is also noted that many programmers who do this wish they had started doing so earlier, since they realize how much time they had wasted by not commenting the original deck.

DO

use TITLE, SPACE, and EJECT commands. The command

```
TITLE 'A HEADING MESSAGE'
```

skips the listing to a new page, and prints the heading message at the top of every page until another TITLE command is issued. This not only clearly labels your listing, but it saves time in looking through a listing which is more than a few pages long. The command

```
EJECT
```

skips the listing to a new page, and is useful in blocking off major parts of a program. The command

```
SPACE n
```

inserts n blank lines into the listing at that point. This is useful for blocking off smaller sections of a program, particularly small loops, register equates, etc.

Not only do listing control instructions aid to the readability of a program, but they also save the programmer time in debugging.

DON'T

merely restate an instruction when you place a comment on it. Of the following two examples, which is more explanatory?

```
A     1,VAR       ADD VAR TO REGISTER 1

A     1,VAR       R1=SUMMATION OF ODD PRIME NUMBERS
```

DON'T

put several single comments between statements in an unreadable manner. It is often useful to indent a single comment to column 16. This keeps it from interfering with the reading of labels and opcodes, and thus distinguishes it from the machine instructions.

DO

use comment card blocks which list useful information. For example, a list of register allocation and usage is extremely helpful, not only in debugging, but also in revising a program. Such a list should appear as part of the preface to the appropriate section of code. Another example is a list of calling conventions for subroutines. For extensive programs, lists of the following might be kept at the beginning of each subroutine: MACROS USED, SUBROUTINES CALLED BY THIS SUBROUTINE, SUBROUTINES WHICH CALL THIS SUBROUTINE, VARIABLES USED BY THIS SUBROUTINE, VARIABLES CHANGED BY THIS SUBROUTINE, etc.

DO

block off large sections of comment cards. Large blocks of comments can begin in whatever column is appropriate, but in general, should use most of the card, since they will otherwise add a great deal of length to a program. For the sake of appearance, comments should be blocked off by blank lines (SPACE n) or lines of continuous characters. The most common characters used for this purpose are asterisks (in columns 1-71, or in just the odd columns). An esthetic appearance can be obtained by placing an asterisk in column 71 of each comment card in a major block, with lines of asterisks before and after the entire block of documentation.

DO

flag instructions which will be modified during execution in order to make programming logic obvious. This may be accomplished by using '*-*' or '$', the latter EQU'ed to zero, for any modified field. For example,

```
$        EQU   0                  $ => INST. MODIFIED IN EX

.......... other statements ..........

         STC   2,MVC+1            SET BUFFER LEN. FOR LATER
*                                 USE.

.......... other statements ..........

MVC       MVC    OUTPUT($),0(5)   MOVE VARIABLE  #  BYTES   INTO
*                                 OUTPUT BUFFER.

.......... other statements ..........
```

The above methods have been derived both from the examination of many professionally-written programs and from the authors' own experiences. Thus, they are not arbitrary rules but techniques which have been widely used and proven to be effective aids in programming assembler language.

STANDARD LINKAGE CONVENTIONS

Charles Pfleeger

Under OS/360, certain conventions have been established regarding the use of registers. These conventions will have been followed when you, the problem programmer, receive control from the system; they should be followed for any routines which you call, or for communicating with the system (e.g. system macro calls, SVC's, returning control, etc.). Following these conventions will make your code easier for someone else to follow. Certain debugging aids are also available for those who adhere to standard conventions. In general, unless there is a strong reason to deviate, these conventions should be employed.

REGISTER 14 is called the return register and contains the address to which this routine is to return upon exit.

REGISTER 15 is called the entry point register, and contains the address through which this routine was entered. Note that temporary addressability may be established by

```
        USING   entrypointname,15
```

If this routine calls no other routines, register 15 may be used as a permanent base register. If this routine calls any other routines, however, register 15 will be changed, and should not be used as a permanent base register. In this latter case, the sequence

```
        LR      BASEREG,15
        USING   entrypointname,BASEREG
```

(where BASEREG is any of registers 2-12) may be used to establish permanent addressability.

On return, register 15 may be used to return a code to indicate normal or error return. One frequently-used technique is to set R15 zero on a normal return and set it non-zero if some error condition occurred prior to return.

REGISTER 0 is used to return the single result from some process (as in a Fortran function subprogram). Note: although you will probably not use this convention much, it is heavily used by the operating system. Register 0 cannot be guaranteed to be intact after executing some call to the system, as a system macro, or an SVC.

REGISTER 1 is the pointer to an argument list. It contains the address of the first of one or more full word entries (on consecutive f.w. boundaries). These entries are the addresses of arguments to be used by the calling routine.

If there may be an indefinite number of arguments, (as with a routine which would accept one, two, or any number of arguments--c.f. Fortran MAX0), the first bit of the last address is set to a 1. (This bit will not interfere with ordinary S/360 addresses, since an address can be fully specified in 3 bytes; byte 1 is ignored on an address constant.)

The following example illustrates how to use the address list passed through register 1.

```
            LA   1,ARGLIST      get argument list address
            L    15,=V(CALLRTN) get entry address
            BALR 14,15          call routine
             . . .
ARGLIST     DC   A(ARG1)
            DC   A(ARG2)
             . . .
            DC   X'80',AL3(ARGn)  Note the length factor
                                  does not provide auto-
                                  matic alignment.
             . . .
CALLRTN     CSECT
             . . .
            L    2,0(1)         get addr. of next arg.
            LTR  1,1            last arg. in list?
            BM   RETURN         if yes, return
            LA   1,4(1)         else get addr. of next arg.
```

When a programmer receives control from the system, information from the PARM field of his EXEC card is passed via register 1. Register 1 points to a fullword of storage. Bit 0 of this fullword is set to 1 (to indicate the last--only--argument of the list). This fullword contains the address of a halfword. The halfword is a count of the number of characters in the parm field message, and these characters follow immediately after the halfword count field. The contents of the halfword may be picked up to use as a length count in an execute instruction, and the address of the halfword may be used as a base to move the information characters of the PARM field.

REGISTER 13 is called the save area register. It contains the address of an 18 fullword area (on a f.w. boundary) within the calling routine. The routine called will use this area to save the contents of registers, to be able to return the registers intact to the calling program. This save area has a set format:

```
Word 1    Used by PL/I and FORTRAN
Word 2    address of the save area used by the calling
          program.
Word 3    address of the save area set up by the called
          program.
Word 4    address to which to return (reg. 14).
Word 5    address of entry point (reg. 15).
Word 6    contents of register 0.
. . .
Word 18   contents of register 12.
```

Save areas are chained in a doubly-linked list. At any low-level routine, by tracing back through a chain of save area links, one can eventually return to the system at the original point of call.

When your routine is entered, first you should save registers and then establish and link your own save area.

```
          STM  14,12,12(13)   save regs. 14, 15, and 0-12 in
                              calling program's save area.
          LA   5,MYSAVE       get addr. of my save area
          ST   5,8(13)        link calling pgm. s.a. to mine
          ST   13,4(5)        link my s.a. to calling pgm's
          LR   13,5           transfer pointer to s.a.
```

On return:

```
          L    13,4(13)       retrieve addr. of calling pgm's
                              save area
          LM   14,12,12(13)   restore registers as they were
          BR   14

MYSAVE    DC   18F'0'
```

A calling program is known as a "higher routine", and the routine called is the "lower routine". Register 13 is always to point to an area whose contents may be destroyed.

An exception to the requirement that a routine must always establish a save area is that the lowest-level routine (the one which calls no others) need not set up a save area. The reason for this is the save area is for the use of any called routines, but that the lowest-level routine will have no called routines.

It is important to know the conventions on save areas, but the use of XSAVE AND XRETURN (consult appropriate documentation) can reduce the problems in coding and linking save areas.

THE NAME CONVENTION is a means of having the EBCDIC form of the name of a routine appear at certain key places on dumps. To use this convention, the first four bytes of a routine must be a branch, on 15 as a base register, which passes over a series of bytes.
These bytes contain the EBCDIC form of the name of a routine, and also a length count for this name area. This example shows how to code a name field.

```
     name      CSECT
               B    m+1+4(,15)
               DC   X'm'
               DC   CLm'name'
               next instruction.
```

The value of m must be odd, in order to have the next instruction properly aligned. An alternate approach uses the convention on register 15:

```
     name      CSECT
               USING     name,15
               B         NEXTINST
               DC        X'm'
               DC        CLm'name'
     NEXTINST  next instruction
```

Notes:

O/S follows these conventions strongly. In particular, the system often destroys the contents of registers 0, 1, 14, and 15 when it returns control from a system macro, an SVC, or another system function. One must SAVE THE CONTENTS of these registers BEFORE executing one of these functions; hard-to-locate errors will frequently occur after failure to do so.

It is a good idea to mark a save area upon exit. This is usually done be moving X'FF' into the first byte of the fourth word of the save area (the place register 14 was stored). Although this technique does not seriously affect the contents of the save area for reading in a dump, this technique quickly shows what save areas are active and which are not active when reading a dump.

Register 13 must be kept as the save area pointer; however, by careful programming, it can also double as a base register. Consult the appropriate section from XSAVE and XRETURN documentation for the coding sequence using these macros. You may set up your own save area for this purpose by setting it high in a program, and following it by a USING on register 13, referencing the name of the save area.

For reserving the 18 fullwords of storage for a save area, use DC instead of DS. A constant of F'0', or F'-1' will quickly show in a dump if the save area was ever used.

SAVE and RETURN are two system macros which will eliminate much of the coding for saving and returning conventions. SAVE generates the code necessary to save a specified series of registers. The registers are specified as they would be for a STM instruction. In addition, the operand T will cause registers 14 and 15 to be stored, regardless of what other registers may also be saved from the pair specified. The following example will cause registers 5, 6, ... 10 and 14 and 15 to be saved.

```
         SAVE (5,10),T
```

The RETURN macro will generate code to restore registers, insert a return code in register 15, flag the save area (X'FF' in wd. 4), and branch back via register 14. The registers to be restored are coded as with SAVE. If 15 already has a return code in it and should not be restored, it is coded as RC=(15); else RC=n may be coded, where n is some value to insert into register 15. The operand T causes the flag X'FF' to be inserted in the save area. The following code will restore registers 5, 6, ... 10 to be reloaded, the save area to be flagged, and 15 to be loaded with a value 16.

```
         RETURN   (5,10),T,RC=16
```

NOTE Both of these macros expect that register 13 will already be loaded with the address of the appropriate save area.

The use of the PSU macros XSAVE and XRETURN can provide added flexibility in saving and restoring registers. Both can generate code to print a trace message showing entry and exit from a module; XSAVE can be used to establish and load a base register or to print a snap of the registers saved; XRETURN can create a save area. NOTE that as with RETURN, XRETURN assumes that register 13 still points to the relevant save area.

For most uses, the code XSAVE alone can be used to save registers. For a routine with only one return point, XRETURN SA=* will suffice; if a routine has more than one return point, however, XRETURN alone should be coded at all return points except one, and at that one XRETURN SA=* should be coded. The reason for this is that SA=* will cause a save area to be created; only one should be created per module. For further details on the parameters involved in these two macros, see the appropriate PSU documentation.

The following example causes register 12 to be established as a base register, causes all registers to be saved on entry, cuases no trace messages to be printed on entry or on exit, and causes R15 to be loaded with the return code value 8.

```
        MAIN    CSECT
                XSAVE    BR=12,TR=NO   (Note--default is for all
                                        registers to be saved)
                XRETURN  SA=*,TR=NO,RC=8
```

```
         PRINT NOGEN
         EQUREGS
MAIN     CSECT
         XSAVE .                   ESTABLISH STANDARD LINKAGE
         CALL  NEXT                CALL LOWER ROUTINE
         XRETURN  SA=*              ESTABLISH SAVE AREA
         LTORG
NEXT     CSECT
         XSAVE .                   ESTABLISH STANDARD LINKAGE
         CALL  LAST                CALL LOWEST ROUTINE
         XRETURN  SA=*             ESTABLISH SAVE AREA
         LTORG
LAST     CSECT
         XSAVE .                   ESTABLISH STANDARD LINKAGE
         CALL  TRACE               CALL TRACE RTN TO PRNT S.A.
         XRETURN  SA=*             GENERATE SAVE AREA
         LTORG
*
*  THE ABOVE ROUTINES DO  NOTHING BUT ESTABLISH LINKS TO TRACE
*  THROUGH THE SAVE AREAS
*
*
*  ROUTINE TRACE PROVIDES A PRINTED TRACE OF THE NAMES OF THE
*  CSECTS OF ACTIVE S.A.'S.  IT USES DSECTS SAVEAREA AND NAMECONV
*  TO FORMAT THE SAVEAREA AND FIRST FEW BYTES OF THE PROGRAM.
*
TRACE    CSECT
         XSAVE SA=TRACESA          ESTABLISH LINKS
         USING SAVEAREA,R13
         USING NAMECONV,R15
         XPRNT =CL25'0BACK TRACE OF SAVE AREAS--',25
         L     R13,4(R13)           CONNECT TO FIRST ACTIVE S.A.
LOOP     LTR   R13,R13             CHECK IF END OF CHAIN
         BZ    DONE                IF YES, EXIT
         L     R15,REG15SAV        GET PTR. TO BEGIN. OF CSECT
         CLC   BRANCH,=X'47F0'     CHECK TO SEE IF VALID BRANCH
         BNE   ERROR               IF NOT, ABORT
         IC    R7,LENGTH           PICK UP LENGTH OF NAME
         BCTR  R7,R0               SET UP FOR EXECUTE
         EX    R7,MOVE             MOVE CHARS. OF NAME TO OUTPUT
         XPRNT OUT,40              PRINT NAME OF ROUTINE
         MVC   OUT+1(39),OUT       BLANK OUT OUTPUT AREA
         LM    R14,R11,REG14SAV    RELOAD REGS. (FOR RETURN)
         L     R13,BACKLINK        FOLLOW LAST LINK
         B     LOOP
DONE     XPRNT =CL25'0BACK TRACE COMPLETED',25
          LA    R13,TRACESA
         XRETURN  SA=TRACESA
ERROR    XPRNT =CL25'0ERROR IN TRACE-BACK',25
         ABEND 999                 ABORT
MOVE     MVC   OUT+1(*-*),NAME     INSTR. FOR EXECUTE
OUT      DC    CL40' '
         LTORG
```

```
*
*  THE FOLLOWING DSECT FORMATS THE SAVE AREA
*
SAVEAREA DSECT
UNUSED   DS    F
BACKLINK DS    F                   PTER TO HIGHER S.A.
FORELINK DS    F                   PTER TO LOWER S.A.
REG14SAV DS    F                   SAVE AREA FOR REG 14
REG15SAV DS    F                    START OF S.A. FOR REG 15-12
*
*  THE FOLLOWING DSECT FORMATS THE BEGINNING OF A CSECT.  IF THE
*  NAME CONVENTION IS FOLLOWED, THE FIRST INSTR MUST BE A BR. ON
*  R15 AS A BASE REG. FOLLOWED BY A  LENGTH AND A NAME.
*
NAMECONV DSECT
BRANCH   DS    XL2,XL2             SPACE FOR BSC INSTR(4 bytes)
LENGTH   DS    C
NAME     DS    C                   SPACE FOR NAME (MARK BEGINNING
*                                    ADDR. ONLY)
         END   MAIN
/*
```

Following is the output from this example--

```
*** MAIN ENTERED ***
*** NEXT ENTERED ***
*** LAST ENTERED ***
*** TRACE ENTERED ***
BACK TRACE OF SAVE AREAS
TRACE
LAST
NEXT
MAIN
IEWLCTRL
BACK TRACE COMPLETED
*** TRACE EXITED ***
*** LAST EXITED ***
*** NEXT EXITED ***
*** MAIN EXITED ***
```

Following is the actual assembler listing of the TRACE csect. Notice those instructions which reference labels from the SAVEAREA and NAMECONV dsects. Look at the object code and see what the base register and displacement by which they were assembled is.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT
000300                             192 TRACE    CSECT
                                   193          XSAVE SA=TRACESA
000000                             220          USING SAVEAREA,R13
000000                             221          USING NAMECONV,R15
                                   222          XPRNT =CL15' BACK TRACE
000386 58DD 0004            00004  232          L     R13,4(R13)
00038A 12DD                        233 LOOP     LTR   R13,R13
00038C 4780 C07E            003E0  234          BZ    DONE
000390 58F0 D010            00010  235          L     R15,REG15SAV
000394 D503 F000 C1C6 00000 00528  236          CLC   BRANCH(2),=X'47F0
00039A 4770 C13A            0049C  237          BNE   ERROR
00039E 4370 F004            00004  238          IC    R7,LENGTH
0003A2 0670                        239          BCTR  R7,0
0003A4 4470 C170            004D2  240          EX    R7,MOVE
                                   241          XPRNT OUT,40
0003CE D226 C177 C176 004D9 004D8  251          MVC   OUT+1(39),OUT
0003D4 98EB D00C            0000C  252          LM    R14,R11,REG14SAV
0003D8 58D0 D004            00004  253          L     R13,BACKLINK
0003DC 47F0 C028            0038A  254          B     LOOP
                                   255 DONE     XPRNT =CL20'TRACE COMPL
000406 41D0 C0F2            00454  265          LA    R13,TRACESA
                                   266          XRETURN  SA=TRACESA
                                   285 ERROR    XPRNT =CL20'ERROR IN TR
                                   295          ABEND 999,DUMP
0004D2 D200 C177 F005 004D9 00005  303 MOVE     MVC   OUT+1(*-*),NAME
0004D8 4040404040404040            304 OUT      DC    CL40' '
000500                             305          LTORG


                                   310 *
000000                             311 SAVEAREA DSECT
000000                             312 UNUSED   DS    F
000004                             313 BACKLINK DS    F
000008                             314 FORELINK DS    F
00000C                             315 REG14SAV DS    F
000010                             316 REG15SAV DS    F
                                   317 *


000000                             318 NAMECONV DSECT
000000 47F0 F000            00000  319 BRANCH   B     0(,15)
000004                             320 LENGTH   DS    C
000005                             321 NAME     DS    C
                                   322 *
000000                             323          END   MAIN
```

COMPUTER SCIENCE 102 - RUN ASSIGNMENT

1. Punch up the following program and run:

```
//          YOUR JOB CARD
// EXEC ASACG
//SYSIN DD *
MAIN     CSECT
*  THIS PROGRAM ILLUSTRATES XDUMP AND PROGRAM INTERUPTION
         BALR  12,0                THESE TWO STMTS ARE FOR
         USING *,12                COMMON LINKAGE CONVENTIONS
         LA    3,CARD              PTR TO CARD IMAGE READ IN
         XREAD CARD,80             READ DATA CARD
         XPRNT CARD,80             ECHO PRINT
         XDECI 4,0(3)              CONVERT DECIMAL TO INTERNAL HEX
         XDECI 5,0(1)              CONVERT NEXT # ON CARD
*  THE NEXT STMT PRINTS CONTENTS OF USERS REGISTERS.
*  NOTE REG 4,5
         XDUMP
         B     4000                ABEND-BRANCH OUT OF PROGRAM
CARD     DS    80C
         END
/*
//DATA.INPUT DD *
    100    -1024
/*
```

2. This next program is a batch run of 5 jobs, each terminating abnormally. The program is stored on RJE file. Punch up the follwoing cards EXACTLY to run the program

```
//    YOUR JOB CARD
// EXEC ASACG,PARM=BATCH
//SYSIN DD *
/*INCLUDE RAB01.BATCH
/*
```

3. To merely get a listing of the prog in 2., use the following cards:

```
//       YOUR JOB CARD
/*INCLUDE RAB01.PRINT
/*INCLUDE RAB01.BATCH
/*
```

A GUIDE TO S/360 MNEMONIC OPERATION CODES

I. INTRODUCTION

The beginning programmer facing the variety of operations available on a modern large computer is often overwhelmed by the large number of operations and complexity thereof. In some cases, a few hints can be helpful in learning and remembering the names, purposes, and usage of the various operations. In particular, certain properties of S/360 menemonics can help the learner a great deal. Some of them are:

A. REGULAR SCHEME FOR NAMING OPCODES

In general, a fairly coherent and regular method has been used in naming operations. In some cases, it is possible to determine the bit pattern and operation of a mnemonic just from looking at it. Related operations usually have related mnemonics.

B. COMMONLY USED MNEMONICS

The designers apparently went to some effort to make the most often used mnemonics the shortest and easiest to remember. Most of these have 1 or 2 letter mnemonics.

II. NAMING OF MNEMONIC OPCODES

A. VERB (MODIFIER) (DATA TYPE) (MACHINE FORMAT)

The mnemonics generally follow the format given above, with the VERB always present, while the others may be omitted. The general meanings of the fields are given below.

1. VERB: specifies a general type of action performed, such as addition, subtraction, comparison, data movement.

2. MODIFIER: specifies a modification of the general action given by the verb, such as logical addition (rather than algebraic), moving multiple registers rather than single ones, and performing different actions while loading one register into another.

3. DATA TYPE: specifies the type of data being operated on, and is usually the same letter as that used to define a constant of the given type, such as H (halfword), P (packed decimal), etc.

4. MACHINE FORMAT: gives the type of machine instruction being used. This is most typically done by adding R or I to an RX mnemonic to obtain a similiar RR or SI instruction.

In general, the RX instructions, which are the heaviest used, have the shortest mnemonics, and most of the other instructions can be built from them by adding more letters.

B. EXAMPLES OF COMMONLY USED MNEMONIC ELEMENTS

The following sections explain the common mnemonic elements.

1. VERBS

VERB	MEANING, COMMENTS
A	Add two numbers (which may be binary, decimal, or floating)
B	Branch to another instruction (like GOTO)
C	Compare two fields (numbers or character strings)
CV	ConVert a number from one base to another
D	Divide one number by another
L	Load a quantity into a register from another or from storage
M	Multiply one number by another
MV	MoVe information from one area in storage to another.
N	aNd information together (logical AND)
O	Or information together (logical OR)
S	Subtract one number from another
ST	STore a register (or part of one) into storage
X	eXclusive or information together (logical exclusive OR)

For example, note that a given VERB may begin many instructions, which immediately shows they are related to each other. For example, the following are all comparison operations: C, CD, CE, CH, CL, CP, CR, CDR, CER, CLC, CLR.

2. MODIFIERS

The following lists verbs and their common modifiers.

VERBS	MODIFIERS	MEANING, EXAMPLES
A,C,S	L	Logical addition, comparison, or subtract is used rather than algebraic. EX: AL, CL, CLC, SLR.
B	AL	And Link - form of branch for doing linkage to subroutine so it can return. EX: BAL, BALR
	C	Condition - branch or not depending on a previously set condtion (IF(--) GOTO --). EX: BC, BCR.
	CT	Count - branch form used to decrement a register and branch if value not zero (DO LOOP). EX: BCT, BCTR.
	X	indeX - branch form for incrementing and testing index quantities. (DO LOOP). EX: BXH, BXLE.
L	C	Complement - used to set a register to complement itself or other (Y = -ABS(X)). EX: LNR, LNDR.
	P	Positive - set register to positive value from self or other (Y = ABS(X)). EX: LPR, LPER.
	T	Test - set register to value from self or other,
L,ST	M	Multiple - several registers are loaded or stored in one operation. EX: LM, STM

3. DATA TYPES

As noted previously, a data type character is usually the same as that used in a DC or DS statement to obtain a given type of data. If a type character is omitted, it usually implies that the instruction operates on 32-bit, fullword, binary quantities (such as A, C, S, etc).

DATA TYPE	MEANING, COMMENTS
C	Character - usually a contiguous string of bytes in memory, treated as printable characters or a string of bits. (FORTRAN LOGICAL*1). EX: MVC, CLC, OC, IC, STC. USUALLY IMPLIES SS INSTRUCTION FORMAT (all except IC, STC).
D	Double precision floating point (Doubleword, 64 bit) (FORTRAN REAL*8). EX: AD, SD, LTDR, LD. IMPLIES RR OR RX INSTRUCTION FORMAT.
E	Exponent - single precision floating point (fullword, 32 bit, FORTRAN REAL*4). EX: AE, LER, ME. IMPLIES RR OR RX INSTRUCTION FORMAT.
H	Halfword - 16 bit binary number (FORTRAN INTEGER*2) EX: AH, MH, STH, CH. IMPLIES RX FORMAT.
P	Packed decimal format (2 decimal digits per byte). EX: AP, SP, CP. IMPLIES SS INSTRUCTION FORMAT OF TWO-LENGTH TYPE.

4. MACHINE FORMATS

Several characters are used to denote the specific type of operand format being used (note that the data types can also imply specific formats. If they imply one of several, the last character distinguishes among them).

FORMAT	MEANING, EXAMPLES
I	Immediate - IMPLIES SI FORMAT. EX: MVI, CLI, OI.
R	Register - IMPLIES RR FORMAT. EX: AR, BCR, DDR.

III. EXAMPLE OF FAMILY OF RELATED OPCODES

This section lists all the members of the 'Compare' family of menemonics, showing their relationships adn the elements present in each name. The letters V M D F stand for Verb, Modifier, Data type, and machine Format.

OP-CODE	V	M	D	F	TYPE	COMMENTS
C	C				RX	fullword algebraic compare, the basic one.
CL	C	L			RX	fullword logical comparison (logical modifier)
CD	C		D		RX	compare double precision floating numbers
CE	C		E		RX	compare single precision floating numbers
CH	C		H		RX	compare a register algebraicly with halfword from storage (with sign extension)
CP	C		P		SS	compare two packed decimal numbers
CR	C			R	RR	compare two fullword values algebriacly, gotten from C by adding R.
CLC	C	L	C		SS	compare logically character strings
CLI	C	L		I	SI	compare logical immediate (a byte in memory with the one inside the instruction)
CDR	C	D		R	RR	compare double precision (in registers)
CER	C	E		R	RR	compare single precision (in registers)

The Sytem/370 computers have some additional opocdes:

CLM	C	L		M	RS	compare logical masked (from register to mem)
CLCL	C	L	C	L	RR	compare logical character strings long (up to 16 million bytes in one compare)

Consider the problem of writing a FORTRAN program which would simulate the operation of the instructions above (i.e., maintain variables representing PSW, Memory, GP Registers, etc, and go through the Fetch-Instruction, Decode, Fetch-Operands, Execute cycle). The arrangement of the opcodes would make it easy to share code, i.e., it would not be necessary to code each instruction separately. As an example, consider the following related instructions:

MNEMONIC	HEX CODE	BINARY CODE	SAMPLE INSTRUCTION/ASSEMBLED		
CR	19	0001 1001	CR	0,1	1901
CH	49	1000 1001	CH	0,2(3,4)	49034002
C	59	1001 1001	C	0,4(5,6)	59056004

Examine the bit patterns above. The first teo bits give the Machine Format (00-RR, 10-RX), the third and fourth give a Data Type (01-Fullword,00-Halfword in this case). The fifth-eighth bits give the Verb (1001 - algebraic Compare). In essence, there is only 1 Compare, which is branched to after the operands are obtained.

COMPUTER SCIENCE 102 - TOPICS COVERED, HANDOUTS
WINTER TERM 1972 - MASHEY

The handouts given are described in file CS102HN.

#	DATE	TOPICS, HANDOUTS, READING ASSIGNMENTS
--	--/--/72	----------

1 01/07 introduction to course. prerequistites (101, 401, equiv)
listed text materials for course:
1) STRUBLE: ASSEMBLER LANGUAGE PROGRAMMING: IBM SYSTEM/360
2) IBM: SYSTEM/360 PRINCIPLES OF OPERATION (POP)
3) IBM: S/360 OS ASSEMBLER LANGUAGE
4) PSU: ASSIST INTRODUCTORY ASSEMBLER USER'S MANUAL
(25 cents, at 426 McAllister)
5) IBM: S/360 REFERENCE CARD (GREEN CARD- BRING TO CLASS)

introduction to information representation in computer.
memory, addressing, similiarity to FORTRAN vector wit1 index
beginning at 0 rather than 1. elements of memory in S/360:
byte, halfword, fullword, doubleword.
positional notation. number systems (binary, octal, decimal,
hexadecimal). conversion between them, uses.
representations of binary numbers: Two's complement, One's
complement, Sign-magnitude. advantages and disadvantages:
(TC - 1 zero, but harder for people; OC - 2 zeroes, but easier
to handle; SM - easiest to handle, but slower circuitry)

READING: STRUBLE CHAPTER 1. Look at ASSIST PART III.

2 01/10 more on information representation; introduction to
machine structure.

meanings of bit patterns: 1,2,4-byte binary numbers; charcters
packed decimal (good for people, but wastes space); floating
point (sign, characterisitc, and fraction).

structure of a very simple machine: memory of 16-bit words;
1 register; 1 program coiunter. a few instructions, each with
opcode and address. explanation of basic instruction cycle:
1) Fetch instruction from where program counter points.
2) Increment program counter.
3) Decode instruction into its parts.
4) Execute instruction.
5) Loop back to 1.

S/360 machine structure: memory (note abbrev. K), GP and
floating point registers, PSW. refer to GREEN CARD.

Begin instruction types:
1) RR (names with -R, examples)
2) RX (give first explanation of base-displacement)
3) RS

READINGS: STRUBLE - CHAPTER 2; POP - pp. 7-15.
HANDOUTS: CS102M1 - page 01 (run some ASSIST programs for dumps)

3 01/12 finish operands formats and introduce assembly language.

4) SI instructions (examples: MVI, CLI)
5) SS instructions (examples: MVC, CLC)
machine language - easy for machine to execute, hard to write
assembly language converted by assembler to machine code.

format of assembly language: label opcode operand comments
machine instructions - actual operations to be executed
assembler instructions (pseudo ops) - give information to
the assembler (ex: CSECT, DS, DC)
some basic functions of the assembler:
1) location counter
2) convert mnemonic opcodes
a) machine ops - translate to codes, increm location cntr
b) assembler ops - take actions specified, increm loc cnt
3) operands - convert to internal binary, base-displacement
4) print out a listing
5) make program ready for execution and pass control to it

stepped through complete test program (XREAD, XPRNT, XDECI, XDUMP) and explained listing and contents of dump.

READINGS: STRUBLE: Chapter 3; ASSIST MANUAL: PARTS II and IV; ASSEMBLER LANGUAGE: pp. 1-18.
HANDOUTS: DOCUMENT (documentation techniques for assembler)

4 01/14 go over some dumps and errors; discuss operand fields.

go through various dumps, showing 0C1, 0C4, and 0C6 errors. cover STRUBLE cahpter 3, pp.50-56: symbols, self-defining terms, literals, location counter reference, absolute and relocatable terms, expressions.

READINGS: STRUBLE: Chapter 4 to page 78.
ASSIGNMENT: STRUBLE: Chapter 1: problems 5,6,7,8,9. Chapter 2: problems 2,3. Chapter 3: problems 1,2,3,4,6.
INFORMAL ASSN: modify dump program to use XDECO and DUMP storage; use program with START to check relocatable vs absolutes. modify one of batch programs to get 0C6 rather than 0C4.

5 01/17 introduction to arithmetic and data movement instructions introduce idea of instruction families and regularity of mnemonics. Go thru following instructions: LR, LPR, LCR, LNR, LTR, L, LH, LA. AR, ALR, A, AL, AH, SR, SLR, S, SL, SH. mention M and D, also briefly note existence of Condition Code and show how to test it, without worrying about encoding. 20-minute question answer and review: questions occurred on differences between literals and self-defining terms, and on use of symbolic register equates.

READINGS: STRUBLE: Chapter 5.
HANDOUTS: CS102AS1 (pages 01 - 02) first assignment - input, output of numbers, calculations in binary.
CS102M1 (pages 02 - 05) S/360 mnemonic construction.

6 01/19 quiz and finish up data movement and binary arithemetic.
Twenty-minute quiz (diagnostic mainly): base conversions (2, 8, 10, 16); negative numbers, base-index-displacement addrs, relocatable vs absolute.
Instructions: LM, STM, MVC, MVI. M, MR, D, DR, MH and hints on what to watch for.
Programming techniques: review input/output & conversions (XREAD, XPRNT, XDECI, XDECO); method for building messages and obtaining length for XPRNT via MSGL EQU *-MSG .

ASSIGNMENT: indexing and comparison assignment, CS102AS1 - 03, due 02/02/72.
HANDOUTS: CS102AS1 - 03 (labeled CS 102 AS2 also) - indexing.
READINGS: STRUBLE CHAPTER 5, start on STRUBLE CHAPTER 7.

7 01/21 condition code, branching instructions, loops.
condition code values and encoding. BCR, BC, Extended Mnemonics (recommended for use over BC #). BALR, BAL and subroutines, BCT, BCTR usage, including decrementing regs.
example of basic loop to sum array of numbers.
flowcharting and good design versus kludge programming.

READINGS: STRUBLE Chapters 7,8,5.

8 01/24 finish loop control, begin on USING/DROP, linkage
Explain BXH, BXLE instructions, give typical setups: forward BXLE loop, backwards BXH loop, BXH scan loop.
show need for USING command. give rules for computation of base displacements: minimum base displacement for those which are available, higher numberred register if several have same.
begin conventions: exaplin registers 15, 14 usage on entry.

HANDOUTS: LINKAGE OS/360 linkage conventions
READINGS: STRUBLE: Chapter 5, LINKAGE HANDOUT

9 01/26 savearea linkage ans ome review.
Describe 18-fullword save area. go through the standard code used at beginning and end of a routine, calling methods. Do not work on argument passing, just normal code.
Misc. instructions: IC, STC, start on Shifts.
Various review for problems.
Note general usage of registers: get students into good habits

READINGS: STRUBLE: Chapter 11.

10 01/28 logical/algebraic arithmetic, shifts
20-minute quiz on previous instructions.
differences between condition code setting, aroverflow in algebraic arithmetic and logical arithmetic. examples.
shift instructions and how they are used.
READINGS: STRUBLE: Chapter 11, begin on chapter 10.

11 01/31 bit manipulation and uses. review on branching
bit manipulation instructions: NR, XR, OR, N, X, O, NI, XI, OI, NC, XC, OC, plus TM. what they do, and how to use them.
EQU trick for SI instructions and how to use it.
review: prototypes on loop control, advantages/disadvantages.

READINGS: STRUBLE: Chapter 10, first 3 sections.

12 02/02 assembler housekeeping, misc areas.
go over all of DC, DS operand formats in detail, showing what can exist as duplication factor-type-length-constant, including multiple operands and constants, expressions as duplication factors and length modifiers. also cover TITLE, EJECT, SPACE

READINGS: STRUBLE: CHAPTER 6, pp 110-121, problems 7,9,10
ASM LANG: 3, 7-9, 10-18 (except variable symbols/sequence symbols, 19-21, 29-33. section 5: EQU, DC (all except Bit Length Modifier, Scale Modifier, Exponent Modifier. all types except E, D, L, P, Z,Y, S, Q, complex relocatability). DS, ORG, LTORG, END. SPACE, EJECT, TITLE
POP: pp 24-34 except CVB, CVD. Logical instructions except TR, TRT, ED, EDMK. Branching except EX.

13 02/04 give out final project, discuss assembler/interpreters
concepts of assemblers: 2 pass assemblers, how to set up opcode and symbol tables (indexed jump methods), output desired.
go over structure of SIGMA 4.5 computer and its interpreter, noting indirect addressing in particular.

HANDOUTS: CS102FP1 (01 -08) general assembler/interpreter descr
CS102FP2 (01 -06) specific material for final project

ASSIGN: Final project, due 13 March (described in CS102FPx)

14 02/07 decimal numbers and conversions
zoned/packed decimal to and from binary. PACK, UNPK, CVB, CVD equivalent codes using M, D loops for decimal-binary-decimal.
examples of various formats/conversions.

READINGS: STRUBLE: Chapter 5: 106-110, Chapter 218-228, 228-233.

15 02/09 misc review, misc instructions, program mask.
SPM instruction, use of program mask, review BXLE, BXH, etc.

16 02/11 MIDTERM
covered data representations, most standard instructions, hand assembly, etc.

17 02/14 on midterm and final project
review of midterm results and problem areas. final project: overall structure, useful modules and how to set them up: decimal scan and output conversions, symbol scan, symbol table manager, opcode lookup, hexadecimal output, etc.
review of BXLE loop control.
HANDOUT: CS102PX1 (01 - 03) programming exercises: hand assembly, interrupts.

18 02/16 more on assembly process, location counter control.
use of ORG to set up tables, timetable for gettting final project done, program design process and debugging.

19 02/18 quiz, TR, TRT
30-minute quiz: hand assembly, BXLE loop setup.
TR uses, setup, workings.
TRT uses, setup, examples.

READINGS: STRUBLE CH 15: pp 342-345, 350-352. prob 1,3,4.
ASSIGN: write TRT table for scanning over hex digits.

20 02/21 programming techniques, use of TR, TRT, conversions
use of global table pointer, examples on TR, TRT.
decimal input conversion, using two TRT's, EX, PACK, CVB.
hexadecimal output conversion, using UNPK, TR.

ASSIGN: write code to perform conversions, also to read in names, place in table, then search table for later names.
READINGS: STRUBLE CH 15: ED, EDMK start.

21 02/23 conversions - hexadecimal input, decimal output, ED
go through hexadecimal input, but not in detail (TRT, TRT, EX of MVC right-justified, TR, PACK 9 into 5, ignoring extra byte)
decimal output: CVD, UNPK, OI for plus number, with leading zeroes.
decimal output: begin on ED, EDMK, doing parts with basic workings of ED, and standard pattern for integer numbers.

COMPUTER SCIENCE 102 - ASSIGNMENT 1
DUE _____

This assignment covers simple input/output, binary arithmetic for fullword and halfword numbers, and basic data movement and testing codes for handling such numbers.

AI. BASIC PROGRAM

The basic program should do the following:

A. Read a card (XREAD), and print it out immediately (called an ECHO CHECK - standard practice). The card contains 5 numbers punched on it, which are to be scanned and converted (XDECI) to binary form, and placed in 5 consecutive fullwords in memory. Print the hexadecimal values of these 5 words (20 bytes), using XDUMP.

B. Perform the following computations in a straightforward way, storing each result in name given, using RX instructions where you can):

1. F = A + B + C
2. G = -A - B - C (LCR useful)
3. H = A * B * E
4. I = A / B (be careful, watch for negative #'s)
5. J = MOD(A,B) (i.e., remainder from# 4.)
6. K = ((A + E) * (B - C)) / D

C. Print all of the above values (F - K) in hexadecimal (XDUMP), then also print them in decimal, using XDECO and XPRNT (print their values an headings all on one line.

D. According to the sign of result H, print one of the 3 messages: H IS LESS THAN ZERO, H IS GREATER THAN ZERO, H IS ZERO.

II. EXTENDED VERSION OF PREVIOUS PROGRAM

Modify the previous program (which only had to read 1 card), to read cards and follow the actions above for each card, until there are no more cards (END-OF-FILE). Keep a count of the number of cards read, and print out this total number before ending the program.

III. HALFWORD VERSION OF PROGRAM II.

Modify program II to use halfwords wherever possible (i.e., store A - K as halfwords, use AH instead of A, etc. Watch out for divides, since no DH instruction exists). How much storage is saved?

IV. REGISTER VERSION OF PROGRAM II.

Change program II by saving all values A - F in registers, then use RR instructions rather than RX instructions. Do XDECI commands directly into registers where the values are saved. A useful trick may be to NAME the registers symbolicly:

```
RA      EQU   3                  REGISTER WHERE VALUE A KEPT
        XDECI RA,CARD            CONVERT VALUE A INTO REG RA
```

This technique will make it clear which value you are using (note that any register reference can be symbolic to an EQU symbol).

V. WHAT TO HAND IN

By using the BATCH feature in ASSIST, you can run several programs in one run. Turn in one run, with each of the programs II, III, and IV shown in execution, with results and output as requested. The run will use control cards like:

```
// EXEC ASACG,PARM=BATCH
//SYSIN DD *
$JOB  ASSIST  PROGRAM VERSION II
..... program II
$ENTRY
..... test data
*** repeat above, starting at $JOB, for programs III and IV.
/*
```

The following test data should be used for each program:

A	B	C	D	E
5	2	-4	-2	2
-2	-1	10	1	-1
4096	1	1	-1	-1

Note that the columns they are punched in should not matter.

COMPUTER SCIENCE 102- ASSIGNMENT 2

This assignment uses the concept of indexing into an array of elements.

I. BASIC PROGRAM

A. Read a card (and echo print) containing a maximum of 20 numbers. Convert the numbers to hex(XDECI) and store them in successive fullwords in memory. Use a loop to eliminate redundant coding. Then, for each card, find the maximum value and the minimum value, printing out these numbers with appropriate labels.

B. Form of data

1. Each card contains a maximum of 21 numbers, where the first number =the number of numbers on the card. You will need the first number for a counter in the loop in part A.

2. There are an unspecified # of data cards. i.e., make your program general to accept any # of data cards.

II. DATA FOR YOUR PROGRAM

```
3       56  76  -76
7  11  123  432  -123  748  -9087  -0
6  33 33  45  10  6  90
4  145  1024   6698  -1024  345
$
```

PRIME INDEX - J R MASHEY

THIS FILE PROVIDES THE PRIME INDEX TO FILES WHICH ARE INSTRUCTIONAL FILES OF MATERIAL TO BE INCLUDED ON THE ASSIST DISTRIBUTION TAPE. THE FORMAT OF THIS FILE ALLOWS IT TO BE USED TO PRODUCE PSU JCL INCLUDE CARDS TO BE USED TO COPY THESE FILES FROM BAT FILES TO TAPE.

EACH FILENAME IS PRECEDED BY '>', AND FOLLOWED BY A PSU RJE ID, IF THE FILE IS NOT SAVED UNDER RJE ID JRM02, WHICH IS THE DEFAULT.

THE PROGRAM JRM05.BATCOPY READS THIS FILE AND PRODUCES JOBS TO COPY THE FILES TO TAPE. ON TAPE, THE FORMAT OF THE FILES IS:

,>FILENAME BEGINNING IN COLUMN 1, ON A SEPARATE CARD PRECEDING EACH FILE. THE COMBINATION ',>' IS NOWHERE ELSE USED IN THE TAPE FILE, SO THAT IT IS EASY TO SEARCH THE FILE FOR A SPECIFIC SECTION AND PUNCH OR PRINT IT.

NOTES: THE FOLLOWING COMMENTS MAY BEGIN THE DESCRIPTIONS:
(JCL): THE FILE CONTAINS JOB CONTROL LANGUAGE CARDS FOR USE ON A OS/360/370 SYSTEM, PLUS TYPICAL SAMPLE PROGRAMS.
(TEXT): THE FILE CONTAINS TEXT MATERIAL, WITH EACH PAGE BEGUN BY A CARD HAVING ',' IN COLUMN 1. THESE ALSO HAVE LOWER CASE LETTERS, AND SO TYPICALLY REQUIRE A 'TN' PRINT TRAIN OR EQUIVALENT FOR BEST APPEARANCE.

>AAAINDEX OVERALL INDEX.

>ASBROPS2 (TEXT) ASSIGNMENT USING ASSIST REPLACE MONITOR TO REPLACE THE BASE REGISTER PART OF ASSIST. SEE ASREPLGD, AND $ASBROPS2
>$BRTEST TEST DATA FOR USE WITH ASSIGNMENT ASBROPS2

>ASPRGTC1 (TEXT) ASSEMBLER PROGRAMMING TECHNIQUES: LINKAGE, MACROS, MODULAR PROGRAMMING.

>ATTACH (JCL) - OS/360 SAMPLE PROGRAM- ATTACH, DETACH, ETC

>BDAM1 (JCL) - OS/360 BDAM EXAMPLE, PART 1 OF 2
>BDAM2 (JCL) - OS/360 BDAM EXAMPLE, PART 2 OF 2

>BPAM (JCL) - OS/360 BPAM EXAMPLE

>BSAM (JCL) -OS/360 BSAM EXAMPLE

>CS102AS1 (TEXT) - 1ST ASSEMBLER COURSE ASSIGNMENT
>CS102FP1 (TEXT) - FINAL PROJECT IN FIRST COURSE - PART1.
>CS102FP2 (TEXT) - 2ND PART OF FINAL PROJECT (WHICH ISAN (ASSEMBLER INTERPRETER FOR SMALL MACHINE)
>CS102M1 (TEXT) - MISC. WRITEUPS FOR 1ST ASM COURSE
>CS102TPA (TEXT) - DAY-BY-DAY OUTLINE OF 1ST ASSEMBLER COURSE

>OSHASP	(TEXT) - EXPLAINS JOB SCHEDULING AND FUNCTIONING OF OS/360 WITH HASP.
>OVLY1	(JCL) - ILLUSTRATES USE OF LINK-EDITOR OVERLAY FACILITIES, SHOWING DIFFERENT TREE STRUCTURES.
>PTPCHMAC	(JCL) - ILLUSTRATES USE OF UTILITY IEBPTPCH TO PRINT MACROS FROM LIBRARY.
>QSAM	(JCL) - SHOWS USE OF OS/360 QSAM MACROS.
>RECURASM	(JCL) - SHOWS USE OF GETMAIN/FREEMAIN IN MAKING RECURSIVE ASSEMBLER PROGRAMS.
>SPIESTAE	(JCL) - SHOWS USE OF SPIE/STAE MACROS
>TIME	(JCL) - ILLUSTRATES USE OF TIME, STIMER, TTIMER MACROS FOR TIMING.
>WTOWTL	(JCL) - ILLUSTRATES WTO, WTL MACROS.

ASSIST BASE REGISTER ASSIGNMENT

DUE _____

This assignment is essentially to write the base register handling routine for ASSIST, and run and test it using the ASSIST Replace Monitor. The programmer should first consult the following writeup for general information, ASSIST conventions, and use of the Replace Monitor:

ASSIST REPLACEMENT USER'S GUIDE (ASREPLGD)

I. S/360 BASE REGISTER ASSIGNMENT

This section briefly describes the conversion of program addresses to base-displacement form, as done by S/360 assemblers, particularly ASSIST. The following manual should also be consulted:

IBM S/360 OS Assembler Language GC28-6514, pp. 19-21.

A. Each control section and each dummy section in an assembly is assigned a unique number or section identification (ID), and every label in a given section has that same section ID associated with it.

B. When a register is specified in a USING statement, it is assumed to contain the specified location counter value, and is also flagged with the section ID of the first expression in the USING.

C. When a value used in an instruction must be converted to base-displacement form, the only possible registers which are usable are the ones(if any) which have the same section ID as the value to be converted.

D. If two or more registers are usable as base registers, and have the same section ID, the register used is that one having a value which results in the smallest displacement (0-4095).

E. If two or more registers have the same ID and value, the higher numbered register is used.

F. (ASSIST only) - in ASSIST, all values for USING statements are relocatable, and register 0 is handled exactly the same as any other, which is slightly different from the standard handling. Also, the ASSIST section ID's range from 1 to 255 only.

II. ASSIST INTERFACE REQUIREMENTS FOR BROPS2

General register conventions are given in ASREPLGD. This section describes the module to be written, with the specific requirements for each of the entry points of the BROPS2 module. The register notation used is that from ASREPLGD.

CSECT NAME: BROPS2

ENTRY POINTS: BRINIT, BRUSIN, BRDROP, BRDISP

ENTRY AND EXIT CONVENTIONS

A. BRINIT - is called one time at beginning of asembly, to perform any initialization required by BROPS2. Must be serially reusable, and so cannot just DC any tables to required values.

B. BRUSIN - is called whenever a USING is processed.

ENTRY CONDITIONS

RA = number of register which can be used. (0-15)
RB = address declared for base register. (0-2**24-1)
RC = section ID of the address. (1-255)

C. BRDROP - is called when a DROP is found.

ENTRY CONDITIONS

RA = number of register to be dropped. (0-15)

EXIT CONDITIONS

RB = 0 if register was an active base register.
= nonzero value, if register was not usable at the time.

D. BRDISP - is called to convert an address-ID to base-dsiplacement.

ENTRY CONDITIONS

RA = address to be converted to base-displacement form. (0-2**24-1)
RB = section ID of the address to be converted. (1-255)

EXIT CONDITIONS

RA = base-displacement form of address, if there was one, in low-order halfword of register (bits 16-31). Bits 0-15 should be zeroes.
RB = 0 if the address was properly converted.
= nonzero value if an addressibility error occurred, i.e., if there was no register with the proper section ID, and a value from 0 to 4095 less than the value to be converted.

NOTE the above rules must be followed exactly. If they are not, error messages will be given by the ASSIST Replace Monitor.

III. IMPLEMENTATION METHODS

This section outlines several different methods of implementing the BROPS2 module. The following are ways in which the assignment may be handled, with the instructor specifying in class which one is to be followed:

Write the module in 1 specific way.
Write the module in several ways, and compare their performance.
Write the module any way at all.
Write the module any way, but optimizing for one of several goals.

A. REGISTER TABLE ORDERED BY REGISTER NUMBER

A fairly simple way to handle the problem is to just keep a table ordered by register number, which can easily be indexed into to change the values, and can be searched by a fairly simple loop.

B. REGISTER TABLE - LINKED LIST FORM BY ACTIVE REGISTERS

A linked list can be kept of the active registers and their values and section ID's. This can make for faster searches, but can require more space, and more complex code.

C. REGISTER TABLE WITH LINKED LIST ORDERED BY SECTION ID's

A set of linked lists can be kept, with one for each of the section ID's currently active. Each list links together the register or registers which are active base registers and are flagged with the given section ID. This is potentially the fastest method, but also requires the most complex programming.

D. REGISTER TABLE WITH SEPARATE ID TABLE AND TRT USAGE

In this method, a separate search is made of a 16-byte table which contains the active section ID's, possibly using the TRT instruction, then computing the register number from the position in the index table, and going to antoher table to computer the value. This method can be fast, but may require more space, unless 256 bytes of TRT table are available elsewhere.

POSSIBLE OPTIMIZATION GOALS

SPEED - optimize for the fastest program possible. Note that this involves determining the relative frequency of USING, DROP, and base-displacement computations, which can differ depending on the type of programmer producing the test program (i.e., experienced programmers usually have many more USING's and DROP's because they use DSECTs more than do beginners).

SPACE - optimize to produce the smallest complete program.

PROGRAMMING SIMPLICITY - optimize to produce a running program which is simple and understandable, and can be programmed quickly, i.e. simulating the conditions which require a program to be finished in a short period of time.

```
         TITLE 'ASSIST BASE REGISTER/USING/DROP TEST PROGRAM'
*        THIS PROGRAM PROVIDES VARIOUES ERROR TESTS FOR BROPS2.
*        ALL STATEMENTS LEGAL,EXCEPT THOSE WITH ERROR COMMENTS.
TEST     CSECT
         BALR  12,0
         USING *,12
T1       L     0,AA5              AS100
         L     5,AA4
         USING AA5,9
         ST    3,AA5
         USING AA5,10
         A     2,AA5
         USING AA4,10
         SL    3,AA5
         ST    3,AA4
         DROP  10
         USING DSECT1,7
         STH   4,DS1
         LH    5,DS6
         DROP  8,10               TWO AS003 MESSAGES
         CVB   6,DS8
         USING DSECT1,9
         CVD   6,DS8
         USING DSECT2,10
         M     8,DS4
         LA    5,CARD
         D     8,DS5
         DROP  10
         B     T1
         IC    10,DS2A            AS100  ADDRESSIBILITY
         LA    11,OUTR            AS100 ADDRESSIBILITY
         USING *,13
         ST    5,AA4
AA4      DS    F
         DS    1500F
OUTR     DS    C
AA5      DS    F
         EJECT
DSECT1   DSECT
DS1      DS    H
DS2      DS    HL8
DS4      DS    A
DS5      DS    F
DS6      DS    CL6
DS8      DS    D
DSECT2   DSECT
CARD     DS    CL80
DS2A     DS    P
BASE     CSECT
         USING *,13,14,15
         DROP  8,12               AS003 REGISTER NOT USED ON 8
         AH    6,DS1
         AH    4,AA6
         L     5,AA5              AS100 ADDRESSIBILITY ERROR
         DROP  7,9,10             AS003 ON 10
         DROP  13,14,15
AA6      DS    H
         TITLE 'OVERALL TEST - STUDENT-WRITTEN PROGRAM'
*              THIS SECTION CAN BE USED TO TEST BROPS2 - IT PROVIDDES
*              STUDENT-WRITTEN SAMPLE PROGRAM.
MAIN     CSECT
```

```
          ENTRY     IN,OUT,SUPERVR
          STM       14,12,12(13)
          BALR      12,0
          USING     *,12
          ST        13,MAINSAV
START     LA        9,0
          LA        6,0
          LA        11,0
          L         15,=V(COROUT2)
          BR        15
OUT       ST        14,BIN1
          L         15,BOUT1
          BR        15
IN        ST        14,BOUT1
          L         15,BIN1
INX       BR        15
SUPERVR   L         13,MAINSAV
          LM        14,12,12(13)
          BR        14
          LTORG
          DS        0F
MAINSAV   DS        F
BOUT1     DS        F
BIN1      DC        V(COROUT1)
COROUT2   CSECT
          ENTRY     STORE,STORE1
          PRINT     NOGEN
          BALR      13,0
          USING     *,13
          B         OUT1
          DS        0F
BLANK     DC        4C' '
OUTPUT    DS        64C
OUT1      LA        4,0
          LA        1,OUTPUT
          LA        5,BLANK
          MVC       0(64,1),3(5)
          LA        3,3
          LTR       9,9
          BH        STORE
H1        L         15,=V(IN)
          BALR      14,15
STORE     ST        10,OUTPUT(4)
          LA        4,1(4)
          CLI       10,C'.'
          BE        H9
          L         15,=V(IN)
          BALR      14,15
STORE1    BCT       3,STORE
          LA        3,3
          LA        4,1(4)
          C         4,=F'63'
          BNE       STORE
H9        LA        3,3
          XPRNT     BLANK(3),65
          CLI       OUTPUT+62,C'.'
          BE        SUPER
          LA        9,1
          B         OUT1
SUPER     L         15,=V(SUPERVR)
          BR        15
```

```
         LTORG
COROUT1  CSECT
         BALR      8,0
         USING     *,8
         B         IN1
H2       L         15,=V(OUT)
         BALR      14,15
IN1      L         15,=V(NEXTCHAR)
         BALR      14,15
         CLI       10,C'0'
         BL        H2
         ST        10,NUMBER
         LA        5,NUMBER
         PACK      DOUBLE(8),0(4,5)
         CVB       5,DOUBLE
         L         15,=V(BEGIN)
         BALR      14,15
REPEAT   L         15,=V(OUT)
         BALR      14,15
         S         5,=F'1'
         BNM       REPEAT
         B         IN1
         LTORG
         DS        0F
NUMBER   DS        F
DOUBLE   DS        D
NEXTCHAR CSECT
         ENTRY     BEGIN
         PRINT     NOGEN
         BALR      7,0
         USING     *,7
         B         BEGIN
         DS        0F
BLNK     DC        4C' '
INPUT    DS        16F
ASTERICK DS        F
BEGIN    ST        14,ASTERICK
CHECK    LTR       11,11
         BNE       F3
         LTR       6,6
         BE        READ
F1       C         6,=F'16'
         BNE       F2
         LA        6,0
READ     XREAD     INPUT,64
         LA        2,3
         XPRNT     BLNK(2),65
F2       L         11,INPUT(6)
F3       IC        10,C' '
         SLDA      10,8
         LA        6,1(6)
         CLI       10,C' '
         BNE       ASTERICK
         B         CHECK
         LTORG
         END
```

S/360 Assembler Language Programming Techniques

John R. Mashey - Winter l972

Topic: Program Modularity and Parametrization Methods:
Using Macros, Internal Subroutine, External Subroutines
This topic: pages 01-08

It is generally important in any computer program to avoid coding any procedure more times than necessary. It is generally best to write something one time, then have it available for later use in many parts of a program. In assembler language, there are three main ways of doing this: macros, internal subroutines, and external subroutines. This writeup describes each of these techniques, gives the advantages and disadvantages of each, and notes under what condition each is best.

I. DESCRIPTION, DEFINITION, AND CALLING

A. MACRO INSTRUCTIONS

A macro instruction is defined, and either placed at the beginning of an assembly language program (a USER macro), or entered into a macro library (a SYSTEM macro). When called, it generates 0 or more assembly language statements at the point of invocation, and the code generated may vary greatly from call to call.

1. DEFINITION

A macro definition begins with MACRO, followed by the PROTOTYPE STATEMENT, which gives the name of the macro. The body of the macro includes 0 or more MODEL STATEMENTS, which are assembler commands and machine instructions to be generated, and macro-operations, which serve to direct the expansion processing of the macro. The macro definition is terminated by the MEND statement. The following steps are typical in defining a macro:

a. DETERMINE BASIC PURPOSE AND GENERATED CODE: It is generally a good idea to write at least some of the statements to be generated together as a code segment first, to get some feel for what is needed.

b. DECIDE ON NECESSARY ARGUMENTS AND THEIR USAGE: it may be a goo idea to write the purpose of each argument in the operand list, punch it and include it in a block of comments at the beginning of the macro. This helps the macro to be done to do what it is supposed to do.

Use POSITIONAL operands for heavily-used arguments, i.e., if an argument MUST be supplied every time, make it positional. In a group of positionals, place the most heavily used ones near the front, since it is much more convenient to omit the later ones than the earlier. Use KEYWORD arguments for values which may not be needed always, or for ones which are conveniently supplied with default values which are most often used. Use SUBLISTS or &SYSLIST for variable numbers of arguments.

c. WRITE ACTUAL BODY OF MACRO, BUILDING MACRO-TYPE COMMANDS AROUND THE MODEL STATEMENTS TO BE GENERATED.

2. INVOCATION

A macro can be called merely by writing its name and supplying it with any needed arguments. Note that a label on a macro call is never generated (and is thus UNDEFINED) unless the macro definition is made to generate it on some model statement.

B. INTERNAL SUBROUTINES

Internal subroutines are sections of code written as parts of a given control section (CSECT), and are only used inside that CSECT. Like external subroutines, internal subroutines can of course call others. They are typically used for small to medium sections of code which are needed at several places in a CSECT, but are not needed by any other, or are not big enough to warrant the overhead in making them external subroutines.

1. DEFINITION

It is often typical to place a group of internal subroutines near the end of the code section of a program (just before the data areas). It is a good idea to set up conventions for the use of internal subroutines, before writing any. The following are often needed: return register (either one standard one, or several different ones), argument registers, and work registers which can be used without saving. In general, internal subroutines should not need to do much saving and restoring of registers. They should be able to return via BR REG.

2. INVOCATION

Calling an internal subroutine is usually done by first filling any argument registers with needed values, then coding: BAL REG, INSUB. This type of linkage can be fast and small.

C. EXTERNAL SUBROUTINES

External subroutines are used for major program segments, and can usually be assembled separately from the rest of the program. In fact they can be written in a different language (i.e., FORTRAN and ASSEMBLER combinations).

1. DEFINITION

An external subroutine may be written in either of two ways in assembly language: as a CSECT, or as an ENTRY within a CSECT. In the first case, the subroutine is entered at the CSECT statements and return at one or more places depending on the desired code. In the second case each entrypoint may be given control, and may share code or be totally separate from the other entries. This form is often used for a group of related routines (like SIN and COS, which are both entries in a CSECT), or for a routine requiring initialization or termination functions different from the normal calling function.

A multiple-entry CSECT is typically set up as follows:

```
CSECTNAM  CSECT
          ENTRY ENTRY1,ENTRY2,...ENTRYN
..... code for entry at CSECTNAM:   multiple-entry  routines  often
..... are entered only at the  entry  points,  not  at  the  CSECT.

ENTRY1   LINKAGE CODE  (SAVE, XSAVE, etc)
..... executable code when called at ENTRY1.....
          RETURN LINKAGE CODE  (RETURN, XRETURN, etc).

..... remaining entrypoint names and code

..... internal subroutines needed by more  than  one  entry  point.

..... data areas used by  various  of  the  entry  point  routines.
```

The following are important points to remember when using multiple entry CSECTS:

THE DIFFERENT ENTRY POINTS NEVER CALL EACH OTHER. In essence, all of the routines represented by the various entry points are at the same level in calling structure of an entire program.

ONLY ONE SAVE AREA IS ACTUALLY NEEDED. Since the routines inside the CSECT never call each other, the user can code the save area at the end of the LAST section of code, so that all of the previous sections can refer to it (note that if placed on the first, it would be difficult for the later ones to access it using a LA instruction: address constants must be used instead). With XSAVE/XRETURN, this means that the SA=* operand is coded only on the LAST XRETURN.

CARE MUST BE TAKEN WITH ADDRESSIBILITY. All of the code sections can of course address the data areas at the end of the CSECT. However, the programmer must be very careful with any internal subroutines he writes, because the BASE REGISTERS USED TO ASSEMBLE INTERNAL SUBROUTINES MUST HAVE THE CORRECT VALUES IN THEM AT EXECUTION TIME. IF THEY DON'T, AS WHEN THEY ARE CALLED FROM DIFFERENT SECTIONS HAVING DIFFERENT USING SETUPS, THEY WILL ASSEMBLE PROPERLY AND THEN BLOW UP AT EXECUTION TIME. IN PARTICULAR, THE PROGRAMMER SHOULD PLACE INSTRUCTIONS TO BE EXECUTED (EX operation) WITH THE SECTION OF CODE USING THEM, AND NOT AT WITH THE DATA AREAS, IF THEY PERFORM ANY SYMBOLIC ADDRESSING.

The problems described above are typically handled either by making all entry point code segments set up the same USING conditions, or by setting a specific register to point to the beginning of the internal subroutines, EXecuted instructions and data. If register 13 points to a save area just above these code sections, it can be used this way, since it will always have that same value. Getting the same USING conditions across an entire multi-entry CSECT can be done:

```
ENTRYX    XSAVE
          L     BASEREG,=A(CSECTNAM)
          USING CSECTNAM,BASEREG
```

Note that the above can be accomplished with the XSAVE AD= operand.

D. COMBINED FORMS

In some cases, it is convenient to combine the ease of use of the macro with the small size of internal or external subroutines. In this case, the macro expansion sets up any needed arguments, saves registers, etc, then generates code to invoke the subroutine. The subroutine then provides the major portion of the processing code, any needed large data areas, etc.

Examples of the combined form are the following macros: XDECI, XDECO, XPRNT, XSNAP, which call XXXXDECI, XXXXDECO, XXXXPRNT, and XXXXSNAP, respectively.

Two different extremes exist in writing combined forms:

1. COMBINED FORM - STANDARD LINKAGE

In some case, the calling sequence to invoke an external subroutine essentially includes the CALL macro or equivalent code, i.e., it uses standard conventions. It typically assumes that registers 0, 1, 14, 15 may be modified without causing trouble. This method is efficient and general, but can cause trouble if used improperly.

2. COMBINED FORM - SPECIAL NONDESTRUCTIVE LINKAGE

In some cases, it may be useful to define a macro instruction which invokes a subroutine, but can be used ANYWHERE without disturbing any registers, changing the condition code, or requiring that certain of the registers not be the ones being used as base registers (in particular, register 15). This is the kind of linkage used from XDECO to XXXXDECO XPRNT to XXXXPRNT, etc. The following shows the general form of such a linkage setup, giving first the kind of code to be generated by the macro part, then the entry and exit code for the associated routine: (NOTE: label is typically an &SYSNDX-generated unique label)

```
         STM  14,0,label          save registers to be changed
         .... evaluate arguments of macro:  any required Load Addresses
         .... must be done using LA 0, argument since doing LA into
         .... any other register could destroy a base register.  If
         .... more than one argument is needed, the remaining ones can
         .... be stored into control block after label.  Examples:
         LA   0,argument
         ST   0,label+12          2nd argument (one arg left in RO)
         .... after all arguments are evaluated and saved, and ONLY
         .... THEN, it is now possible to modify registers:
         L    15,label-4          V-type adcon for routine
         CNOP 2,4                 make sure next inst not on F boundry
         BALR 14,15               call routine, also point 14 at the
                                  argument list following
         DC   V(subroutine entry point) adcon to get there
label    DS   3F                  3 words for saving 14, 15, 0
         DS   F                   space for arguments after first
         .... DS OR DC space here for any remaining arguments
         .... the subroutine will return control to next instruction:
         LM   14,0,4(14)          reload registers.  Note that this is
                                  only safe way, since 15 might have
                                  current base register.
```

The following shows the typical code used to enter and exit the supporting module used with the previous macro expansion. Note that the entry point of the routine might be either a CSECT name, or an ENTRY name, i.e., one CSECT might contain several entrypoints, one for each supporting subroutine needed.

```
entrypoint label definition (CSECT, or label DS OH)
          USING  entrypoint,15      initial base register
          .... save all registers which may be modified by code.  Save
          .... into THIS CSECT (unlike normal OS/360 conventions).
          .... DO NOT SAVE INTO CALLER'S SAVE AREA, since it may not
          .... exist, especially if caller is a lowest-level routine.

          .... initialization code:  if this routine performs I/O, or
          .... calls any others, or requests any supervisor services, it
          .... is a good idea to set up another base register than 15,
          .... set up a save area, and put its address into register 13,
          .... since any of the above actions may result in registers
          .... being saved at wherever 13 points.

          .... processing code to perform required actions

          .... result return code:  result may be left in register 0,
          .... in which case it should not be restored (neither here nor
          .... in generated code before:  i.e., change LM 14,0,4(14) to
          .... LM 14,15,4(14) and STM likewise).

          .... register restoration:  restore all registers modified in
          .... this routine.  Especially restore 14.

          SPM  14                   restore original condition code (note
                                    that calling BALR 14,15 saved it)

          B    number(14)           branch to displacement number beyond
                                    address in 14, enough to pass control
                                    to statement:  LM 14,0,4(14)
```

It may be useful for the programmer to create a DSECT which describes the control block generated by the macro expansion. This would permit the module to refer to arguments and return points using symbols rather than absolute displacements. A typical DSECT might be:

```
dsectnam DSECT
         DS    V(routine)          space for adcon
         DS    3F                  space for regs 14, 15, 0
argument DS    F                   argument value placed here, if any
         .... further arugment DS statements follow.
return   LM    14,0,4(14)     return label (YES, THIS IS LEGAL:  it does
                              NOT generate code, but it makes the point
                              clear as to description of block).
```

If such a DSECT were used, the routine code would include:

```
         USING dsectnam,14     to set up DSECT addressibility
         B     return          return (instead of B number(14)
```

II. ADVANTAGES AND DISADVANTAGES

The following lists the good and bad points of each type:

A. MACRO INSTRUCTIONS

1. ADVANTAGES

Code can be tailored to each individual request, i.e., the code generated by each macro call can vary from a great deal to nothing, such as debug code eliminated by testing a global set variable.

SPEED: macro-generated code can be the fastest in execution, since it can perform its actions without having to set up linkage to another section of code.

VARIABILITY: generated code can vary depending on the nature of arguments passed to a macro (such as testing the TYPE of arguments to generate different instructions).

2. DISADVANTAGES

SLOW ASSEMBLY: macro processing can be very slow.

LARGE CODE: if used improperly, macros can generate large amounts of code very easily. If there are many copies of large blocks of code, much space can be wasted.

OBJECT DECKS: a macro cannot be assembled and an object deck of it gotten like a subroutine can, i.e., if a call is made to a macro, the macro definition must be included in the program or in a library, while a CSECT may be saved as an object deck (which is usually much smaller than the source deck).

B. INTERNAL SUBROUTINES

1. ADVANTAGES

SPEED: although not as fast as in-line code from a macro, the code for an internal subroutine is usually faster than the linkage to an external one. In particular, values can be passed in registers, and usually registers will not have to be saved.

SPACE: internal subroutines require less space than generating the same code several times via macro expansions.

2. DISADVANTAGES

SPACE: if the same function is performed by internal subroutines in several CSECTS, code is thus duplicated and space wasted.

COMPLEXITY: in some cases, in order to make efficient use of a number of internal subroutines, it is necessary to set up fairly extensive rules on usage of registers in a CSECT, so that the linkage among them may be fast and small.

C. EXTERNAL SUBROUTINES

1. ADVANTAGES

SPACE: if written as an external subroutine, code can be usefully called from almost anywhere in a program. Thus, there is only one copy of it, and it generally will occupy the least space.

SEPARATE COMPILE/ASSEMBLY: a routine written as a CSECT can be assembled separately from the rest of the program an object deck can be obtained, and translation time generally saved. The routine may of course be written in a different language than the rest of the program.

2. DISADVANTAGES

LINKAGE TIME: if standard OS/360 linkage is followed, a fair amount of execution time and object code space can be consumed by this linkage. More efficient nonstandard linkage can be used instead, but this brings with it the disadvantage of nonuniformity and lack of generality.

D. COMBINED FORMS

1. ADVANTAGES

In general, the combined forms can possess all the advantages of the separate forms especially since the macro portions can generate different code depending on circumstances; thus the code for the same macro might expand in-line in one case and generate an out-of-line call to a routine in another.

2. DISADVANTAGES

COMPLEXITY: it of course requires somewhat more planning and code to set up a good combined form system, since both a macro and module must be created and meshed together properly.

III. CIRCUMSTANCES FAVORING USE OF THE VARIOUS FORMS

A. MACRO INSTRUCTIONS

In general, a pure macro instruction is used as follows:

VARYING CODE: the required code varies radically from call to call. For example: XSAVE and XRETURN.

SHORT CODE: if a macro can generate less in-line code to perform the required function than is needed to generate a call to the routine, then it should be written as a macro. In some cases, it takes as much work to set up the arguments as it does just to perform the operations. For example: the code to obtain the minimum or maximum of several arguments is probably most efficiently written as a in-line macro.

LINKAGE CODE: code for linking to routines is almost necessarily written as macros, since it makes little sense to call a routine in order to perform linkage, unless the linkage code required is very complex (in which case the program is probably going to be SLOW).

B. INTERNAL SUBROUTINES

Internal subroutines are usually used (as opposed to macros which generate code in-line) under the following circumstances:

CODE WITH LITTLE VARIANCE: if the code is not going to be much different from macro call to macro call, it may be better to let the macro call generate a BAL to one copy of the code as an internal subr.

Internal subroutines are usually used (as opposed to EXTERNAL subroutines) under these circumstances:

SHORT CODE, HEAVILY USED: if code must be used many times by a CSECT, then the faster linkage of internal subroutines usually makes it worth writing it that way.

CODE NEEDED ONLY BY ONE CSECT: if not too long, it is fairly logical to incorporate it as part of that CSECT. It will probably be much more efficient since it will have access to the internal variables of the CSECT, and be able to communicate via register values easily, rather than requiring long operand lists.

C. EXTERNAL SUBROUTINES

LONG CODE: if something is long and complex enough, it may be a good idea to make a separate module of it, test it, get an object deck, then leave it along thereafter.

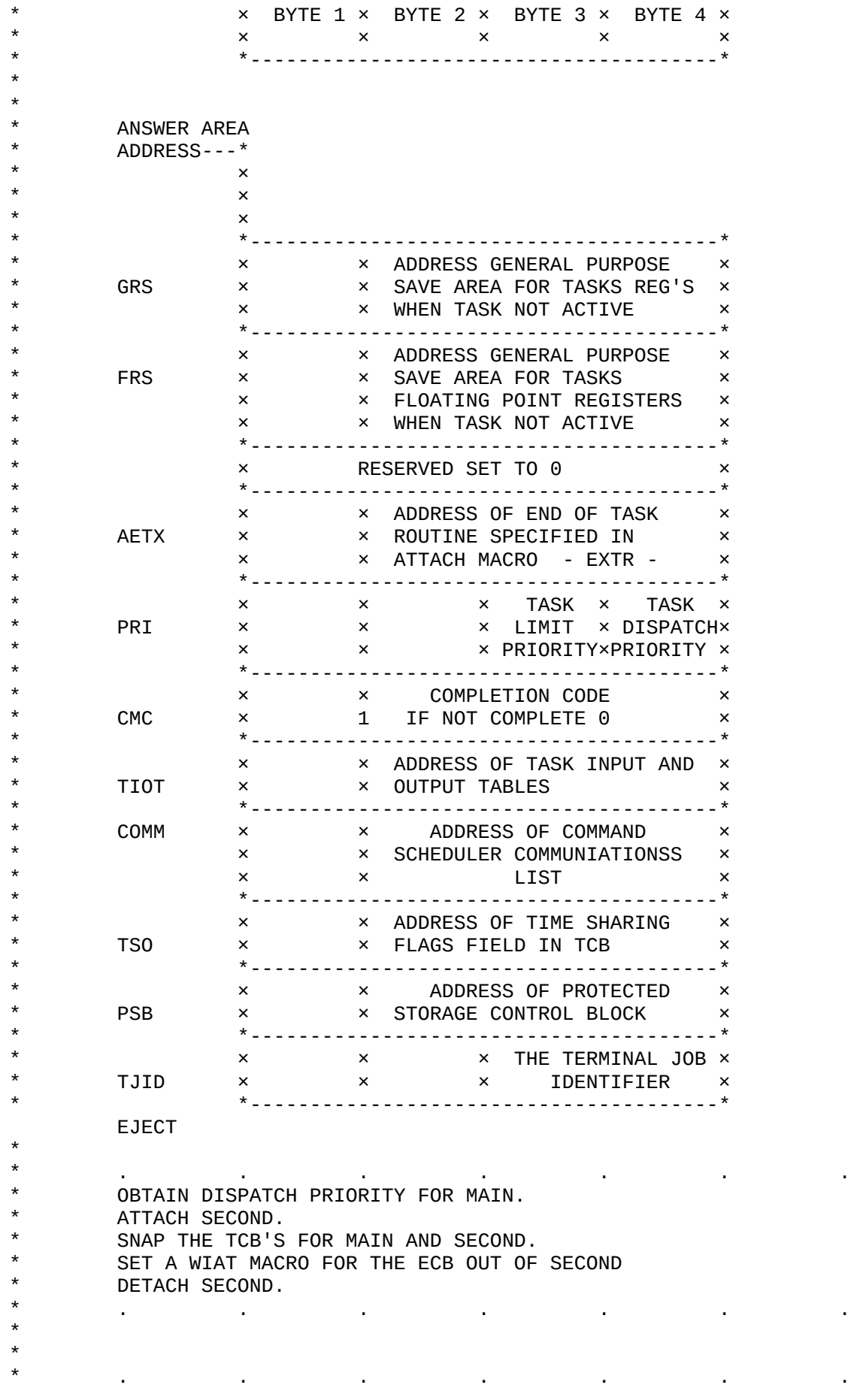
CODE OF GENERAL USE, NEEDED MANY PLACES: in this case, it is practically necessary to make code an external subroutine, so that it can be accessed where needed.

D. COMBINED FORMS

These are useful anywhere the others are. The nondestructive form is specially useful if it is to be used by beginning programmers.

```
//*
//*      THIS JOB WILL RUN WITH   TIME= 75 SECONDS
//*                               RECORDS = 1000
//*
// EXEC ASGCL
//SOURCE.INPUT DD *
*
*        .         .         .         .         .         .         .
*        THE PUTPOSE OF THIS JOB IS TO DEMONSTRATE THE MACRO'S LISTEC B
*        THE PUTPOSE OF THIS JOB IS TO DEMONSTRATE THE MACRO'S LISTECD
*        BELOW:
*                  ATTACH
*                  DETACH
*                  WAIT
*                  POST
*                  EXTRACT
*        .         .         .         .         .         .         .
*
*
*        .         .         .         .         .         .         .
*        THE OVERALL FLOW OF THIS PROGRAM IS
*        1         THE CSECTS SECOND AND THIRD ARE COMPILED AND LINKED
*        2         MAIN IS COMPILED AND LINKED EDITED AND EXECUTES.
*        3         DURING THE EXECUTION OF MAIN IT ATTACHES SECOND
*                  TWICE USING THE ATTACH MACRO.
*        BEFORE SECOND IS ATTACHED THE DISPATCH PRIORITY OF MAIN IS
*        OBTAINED USING THE EXTRACT MACRO.  AFTER IT HAS BEEN OBTAINED
*        IT IS DIVIED BY 2 AND WHEN SECOND IS ATTACHED THE PRIORITY OF
*        MAIN IS HALVED USING THE DPMOD PARM.  WHILE BOTH MAIN AND
*        SECOND ARE COMPETING FOR CPU USE, THE TCB'S FOR MAIN AND
*        SECOND ARE SNAPPED USING THE EXTRACT MACRO.
*        WHEN SECOND AND
*        WHEN SECOND IS EXECUTING IS ATTACHED IT IS GIVEN AN ECB AND
*        IN MAIN  A WAIT MACRO IS ISSUED FOR THIS ECB.
*        4         THEN SECOND IS DETACHED IN MAIN USING THE DETACH
*        MACRO.
*        5         THEN THE PRIORITY OF MAIN IS RESTORED.
*        6         THEN SECOND IS ATTACHED AGAIN USING THE ATTACH MACRO
*        ONLY THIS TIME SECOND IS GIVEN AN EXIT ROUTINE.  IN THE EXIT
*        ROUTINE SECOND IS DETACHED AND CONTROL IS RETURNED TO MAIN.
*        AGAIN THERE IS AN ECB GIVEN TO SECOND AND A WAIT MACRO ISSUED
*        IN MAIN.
*        7         THEN THIRD IS ATTACHED BUT IT IS NOT GIVEN AN ECB.
*        AN ADDRESS IS PASSED IN THE ATTACH MACRO FOR THE ECB AND THIRD
*        USES A POST MACRO TO SET THE ECB.
*        .         .         .         .         .         .         .
*
         EJECT
         PRINT NOGEN
*
*        .         .         .         .         .         .         .
*        WHEN SECOND OBTAINS CONTROL IT DETERMINES WHICH CALL IS BEING
*        MADE THEN IT OUTPUTS THE APPROPRIATE HEADING AND RETURNS.
*        .         .         .         .         .         .         .
*
SECOND CSECT
         XSAVE TR=NO
         L     2,0(1)              GET ADD OF PARM LIST
         L     4,4(1)              GET SECOND PARM
         L     4,0(4)        GET VALUE OF SECOND PARM
         LTR   4,4                 CHECK FOR 0
```

```
         BZ    SEC                IF ZERO THEN SECOND CALL
         LA    5,SHEAD1           GET ADD OF OUTPUT FOR FIRST CALL
         B     OUTPUT             GO TO DO OUTPUT
SEC      LA    5,SHEAD2           GET ADD FOR OUTPUT OF SECOND CALL
OUTPUT   PUT   0(2),0(5)
         XRETURN SA=*,TR=NO
SHEAD2   DC    CL132'0THIS IS SECOND CALL TO SECOND'
SHEAD1   DC    CL132'0THIS IS FIRST CALL TO SECOND'
         PRINT GEN
         END
/*
/*LOG
//OBJECT.SYSLMOD DD DSNAME=&&LOADMOD(SECOND)
// EXEC ASGCL
//SOURCE.SYSGO DD DISP=(OLD,PASS)
//SOURCE.INPUT DD *
         PRINT NOGEN
ECBDSECT DSECT
ECBADD   DS    F
*
*        .         .         .         .         .         .         .
*        THE PURPOSE OF THIRD IS GAIN CONTROL AND OUTPUT A MESSAGE
*        THEN TO POST THE ECB THEN RETURN
*        .         .         .         .         .         .         .
*
THIRD    CSECT
         XSAVE TR=NO
         L     2,0(1)             GET ADD OF DCB
         L     3,4(1)             GET ADDRESS OF ECB
         PUT   0(2),THEAD
         PRINT GEN
         POST  0(3),240
         PRINT NOGEN
         XRETURN SA=*,TR=NO
THEAD    DC    CL132'0THIRD NOW EXECUTING '
         END
/*
//OBJECT.SYSLMOD DD DSNAME=&&LOADMOD(THIRD),DISP=(OLD,PASS)
// EXEC ASGCLG
//SOURCE.SYSGO DD DISP=(OLD,PASS)
//SOURCE.INPUT DD *
         PRINT NOGEN
*
*        .         .         .         .         .         .         .
*        THIS IS THE MAIN JOB STEP.
*        MAIN ATTAHCES SECOND TWICE AND THIRD ONCE IT USES ATTACH,
*        DETACH,EXTRACT,CHAP,WAIT, AND POST
*        DETACH,WAIT,AND EXTRACT.
*        IT DOES ALL THREE ONE STEP AT A TIME
*        .         .         .         .         .         .         .
*
MAIN     CSECT
         XSAVE TR=NO
         OPEN (OTPT,OUTPUT)
         PUT   OTPT,MHEAD
         PRINT GEN
         EJECT
*                          TCB DESCRIPTION
*
*                 *--------------------------------------*
*                 ×         ×         ×         ×         ×
```




```
        EJECT
*
*       .         .         .         .         .         .         .
*       OBTAIN DISPATCH PRIORITY FOR MAIN.
*       ATTACH SECOND.
*       SNAP THE TCB'S FOR MAIN AND SECOND.
*       SET A WIAT MACRO FOR THE ECB OUT OF SECOND
*       DETACH SECOND.
*       .         .         .         .         .         .         .
*
*
*       .         .         .         .         .         .         .
```

```
*
*        .         .         .         .         .         .
*        THE ANSWER PARM IS A FULL WORD IN CORE STORAGE FOR THE RESULT
*        OF THE EXTRACT MACRO.
*        'S' PARM INDICATES THIS JOB.
*        THE FIELDS PARM SPECIFIES WHICH ONE WE DESIRE.
*        .         .         .         .         .         .
*
         EXTRACT ANSWER,'S',FIELDS=(PRI)
         SR    10,10               ZERO REGISTER 10
         LA    9,ANSWER            GET ADDRES OF ANSWER
         IC    10,3(9)             GET PRIORITY OF MAIN
         SRL   10,1                DIVIDE DISP PRI BY 2
         LNR   10,10               MAKE DISPATCH PRI NEG
*
*        .         .         .         .         .         .
*        THE EP IS THE ENTRY POINT FOR THE LOAD MODULE TO BE ATTACHED.
*        PARAM SPECIFIES A LIST OF PARM TO BE PASSED TO THE LOAD MODULE
*        AND VL INDICATES AN INDEFINITE NUMBER OF PARMS.
*        ECB SPECIFIES THE ADDRESS OF ANDEVENT CONGROL BLOCK TO BE
*        ECB SPECIFIES THE ADDRESS OF AND VENT CONGROL BLOCK TO BE
*        ECB SPECIFIES THE ADDRESS OF AN EVENT CONGROL BLOCK TO BE
*        POSTED WHEN SECOND COMPLETES EXECUTION.
*        THE LPMOD SPECIFIES AN INTEGER VALUE TO BE SUBTRACTED FROM
*        THE LIMIT PRIORITY OF SECOND.
*        THE DPMOD GIVES ANVALUE TO BE ADDED TO DISPATCH PRIORITY OF
*        MAIN.  IN THIS EXAMPLE A REGISTER VALUE.
*        THE ADDRESS OF THE TCB OF SECOND IS RETURNED IN REG 1.
*        .         .         .         .         .         .
*
         ATTACH EP=SECOND,PARAM=(OTPT,ONE),VL=1,ECB=ECB1,LPMOD=1,      X
               DPMOD=(10)
         ST    1,TCBADD            SAVE TCB ADDRESS.
*
*        .         .         .         .         .         .
*        NEXT SNAP THE TCB FOR MAIN AND SECOND USING EXTRACT MACRO
*        SEVEN1 IS THE AREA FOR THE RESULT.
*        THE 'S' INDICATES TCB FOR MAIN.
*        THE FIELDS SPECIFY THE FIELDS TO BE SNAPPED.
*        .         .         .         .         .         .
*
         EXTRACT SEVEN1,'S',FIELDS=(ALL,TSO,PSB,COMM,TJID)
         PRINT NOGEN
         XSNAP T=NOREGS,STORAGE=(SEVEN1,SEVEN1+44),                    X
               LABEL='TCB FOR MAIN WITH PRIORITY LOWERED'
         PRINT GEN
*
*        .         .         .         .         .         .
*        NEXT SNAP THE TCB FOR SECOND
*        TCBADD IS A FULL WORD CONTAINING THE ADDRESS OF SECOND TCB.
*        .         .         .         .         .         .
*
         EXTRACT SEVEN1,TCBADD,FIELDS=(ALL,TSO,PSB,COMM,TJID)
         PRINT NOGEN
         XSNAP T=NOREGS,STORAGE=(SEVEN1,SEVEN1+44),                    X
               LABEL='THIS IS TCB FOR SECOND ON FIRST CALL'
         PRINT GEN
*
*        .         .         .         .         .         .
*        ISSUE A WAIT MACRO FOR THE EVENT CONTROL BLOCK PASSED TO
*        SECOND IN THE ATTACH MACRO.  CONTROL PROGRAM WILL POST ECB.
```

```
*        .         .         .         .         .         .         .
*
         WAIT  ECB=ECB1
*
*        .         .         .         .         .         .         .
*        THE TCBADD IS THE ADDRESS OF THE TCB FOR  LOAD MODULE TO BE
*        DETACHED
*        .         .         .         .         .         .         .
*
         DETACH TCBADD
         PRINT NOGEN
         PUT   OTPT,MHEAD1
         LPR   10,10
         PRINT GEN
*
*        .         .         .         .         .         .         .
*        RESORE PRIORITY FOR MAIN.
*        (10) INDICATES THE VALUE TO BE ADDED TO DISPATCH FOR MAIN IS
*        IN REG 10.
*        THE 'S' INDICATES THE CURRENT LOAD MODULE.
*        .         .         .         .         .         .         .
*
         CHAP  (10),'S'
         EJECT
*
*        .         .         .         .         .         .         .
*        ATTACH SECOND WITH EXIT ROUTINE TO DETACH SECOND.
*        SNAP TCB FOR MAIN AND SECOND.
*        THEN SET WAIT MACRO IN MAIN BEFORE CONTINUING.
*        .         .         .         .         .         .         .
*
*
*        .         .         .         .         .         .         .
*        EXTR IS THE ONYL NEW PARM IT GIVES THE ADDRESS OF A ROUTINE TO
*        BE GIVEN CONTROL WHEN SECOND FINISHES.
*        .         .         .         .         .         .         .
*
         ATTACH EP=SECOND,PARAM=(OTPT,ZERO),ECB=MECB,ETXR=MEXTR,VL=1
         ST    1,TCBADD            PUT ADD OF TCB FOR DETACH
*
*        .         .         .         .         .         .         .
*        AGAIN SNAP TCB'S FRO MAIN AND SECOND.
*        .         .         .         .         .         .         .
*
         EXTRACT SEVEN,TCBADD,FIELDS=(ALL)
         PRINT NOGEN
         XSNAP STORAGE=(SEVEN,SEVEN+28),T=NOREGS,                      X
               LABEL='THIS IS TCB FOR SECOND ATTACH OF SECOND'
         PRINT GEN
         EXTRACT SEVEN,'S',FIELDS=(ALL)
         PRINT NOGEN
         XSNAP STORAGE=(SEVEN,SEVEN+28),T=NOREGS                       X
               LABEL='TCB FOR MAIN WITH ONLY ALL SPECIFIED FOR FIELDS'
         PUT   OTPT,MHEAD2
         PRINT GEN
*
*        .         .         .         .         .         .         .
*        WIAT FOR SECOND TO COMPLETE BEFORE ATTACHING THIRD.
*        .         .         .         .         .         .         .
*
         WAIT  ECB=MECB
```

```
         EJECT
         XC    MECB(4),MECB        CLEAN OUT MECB FOR WAIT AND POST
*
*        .         .         .         .         .         .         .
*        ATTACH THIRD
*        SNAP PRIORITY OF MAIN
*        WAIT ON THIRD
*        .         .         .         .         .         .         .
*
*        HERE ECB IS PASSED AS PARM
*        DPMOD IS NEGATIVE INTEGER.
*        .         .         .         .         .         .         .
*
         ATTACH EP=THIRD,PARAM=(OTPT,MECB),DPMOD=-30
         ST    1,TCBADD            SAVE TCB ADDRESS FOR DETACH
*
*        .         .         .         .         .         .         .
*        EXTRACT PRIORITY FOR MAIN AND SNAP IT.
*        .         .         .         .         .         .         .
*
         PRINT NOGEN
         EXTRACT ANSWER,'S',FIELDS=(PRI)
         XSNAP T=NOREGS,STORAGE=(ANSWER,ANSWER+4),                      X
               LABEL='PRIORITY SNAPPED FOR MAIN ON ATTACH TO THIRD'
         PRINT GEN
*
*        .         .         .         .         .         .         .
*        WAIT FOR THIRD TO COMPLETE.
*        .         .         .         .         .         .         .
*
         WAIT  ECB=MECB
*
*        .         .         .         .         .         .         .
*        DETACH THIRD.
*        .         .         .         .         .         .         .
*
         DETACH TCBADD
*
*        .         .         .         .         .         .         .
*        CLOSE OUTPUT FILE AND RETURN
*        .         .         .         .         .         .         .
*
         PRINT NOGEN
         CLOSE (OTPT,)
         XRETURN SA=*,TR=NO
         DROP  12
         EJECT
*
*        .         .         .         .         .         .         .
*        EXIT ROUTINE FOR SECOND CALL TO SECOND.
*        .         .         .         .         .         .         .
*
MEXTR    XSAVE TR=NO,SA=NO
         PRINT GEN
*
*        .         .         .         .         .         .         .
*        DETACH SECOND IN EXIT ROUTINE WITH ADD OF TCB OF SECOND
*        IN TCBADD.
*        .         .         .         .         .         .         .
*
         DETACH TCBADD
```

```
         PRINT NOGEN
         XRETURN SA=NO,TR=NO
MECB     DC    F'0'
ZERO     DC    F'0'
ONE      DC    F'1'
ECB1     DC        F'0'
TCBADD   DC    F'0'
ANSWER   DC    F'0'
SEVEN    DC    7F'0'
SEVEN1   DC    7F'0'
MHEAD2   DC    CL132'0SECOND WILL ATTACHED AGAIN WITH POST ON COMPLETE'
MHEAD    DC    CL132'0MAIN IS NOW EXECUTING  NEXT OUTPUT SECOND'
MHEAD1   DC    CL132'0MAIN EXECUTING WITH DISPATCH PRIORITY HALVED'
OTPT     DCB   DSORG=PS,MACRF=PM,LRECL=132,BLKSIZE=132,RECFM=FA,       X
               DDNAME=FT06F001,EROPT=ACC
         END
/*
//OBJECT.SYSLMOD DD DSNAME=&&LOADMOD(MAIN),DISP=(OLD,PASS)
//DATA.STEPLIB DD DSNAME=&&LOADMOD,DISP=(OLD,PASS)
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.XSNAPOUT DD UNIT=AFF=FT06F001
```

```
//*
//*      THIS JOB WILL RUN WITH   TIME = 170 SECONDS
//*                               RECORDS = 1500
//*
// EXEC ASGCG
//SOURCE.INPUT DD *
/*LOG
*
*        .         .         .         .         .         .
*        TEH PURPOSE OF THIS JOB IS TO DEMONSTRATE BASIC DIRECT ACCESS
*        METHOD (BDAM).
*        .         .         .         .         .         .
*
*
*        .         .         .         .         .         .
*        THE BASIC FLOW OF THIS JOB IS AS FOLLOWS.
*        1     JOB STEP 1 CREATES THE DIRECT ACCESS DATA SET
*        2     JOB STEP 2 ADDS RECORD TO THE DATA SET.
*        3     JOB STEP 3 UPDATES THE DATA SET.
*        4     JOB STEP 4 PRINTS OUT THE DATA SET IN LOGICAL ORDER
*        .         .         .         .         .         .
*
         EJECT
*
*        .         .         .         .         .         .
*        THIS JOB STEP CREATES THE DIRECT ACCESS DATA SET.
*        THE INPUT IS CARDS WITH A KEY OF 4 DIGITS BETWEEN 1001 AND
*        1020.  THE KEY IS CONVERTED TO A BINARY NUMBER WHICH IS USED
*        AS THE BLOCK NUMBER FOR THE INPUT RECORD.  FOR EACH INPUT
*        THE NEXT 40 CHARACTERS ON THE CARDS ARE WRITTEN TO DISK.
*        FOR EACH KEY THAT IS NOT PRESENT A DUMMY RECORD IS WRITTEN ON
*        DISK.  THUS THERE ARE 20 BLOCKS OF DATA ON THE DISK.
*        .         .         .         .         .         .
*
         SPACE 5
         PRINT NOGEN
         EQUREGS
MAIN     CSECT
         XSAVE
         PRINT GEN
*
*        .         .         .         .         .         .
*        OPEN THE INPUT DATA SET AND THE OUTPUT DATA SET.  INITIALIZE
*        REGISTER 8, 9, AND 7.  REGISTER 8 HAS MAX KEY VALUE, REGISTER
*        9 HAS THE MIN KEY VALUE, AND REGISTER 7 HAS ADDRESS OF
*        THE COMPARE INSTRUCTION,
*        .         .         .         .         .         .
*
         OPEN  (INPT,INPUT,OTPT,OUTPUT)
         PRINT NOGEN
         LA    R7,COMPARE
         LA    R8,1020            SET R8 TO LAST KEY VAUUE
         LA    R9,1001           SET R9 TO MIN KEY VALUE
*
*        .         .         .         .         .         .
*        READ IN THE CARD AND CONVERT THE KEY TO BINARY FORM IN
*        REGISTER 10.  THEN CHECK TO SEE IF THIS IS THE NEXT KEY IN
*        THE LIST OF KEYS.
*        .         .         .         .         .         .
*
LOOP     GET   INPT,AREA
```

```
         XDECI R10,AREA
COMPARE  CR    R9,R10              CHECK FOR KEY
         BNE   DUMMY               IF NOT GO TO OUTPTT DUMMY RECORD
*
*        .         .         .         .         .         .         .
*        AT THIS POINT WE KNOW THAT THIS IS THE NEXT KEY TO BE WRITTEN
*        SO WRITE THE INPUT TO KISK, AND THE CHECK THE DECB WITH A
*        CHECK MACRO.  THEN INCREASE THE MIN KEY VALUE AND RETURN FOR
*        NEXT INPUT.
*        .         .         .         .         .         .         .
*
         SPACE 5
*        .         .         .         .         .         .         .
*        FOR THE WRITE STATEMENT THE DCEB1 IS THE NAME OF THE DATA
*        EVENT CONTROL BLOCK, THE SECOND PARAMTTE SF INDICATES NORMAL
*        WRITE CONDITION, OTPT IS THE DCB NAME,, AREA IS THE ADDRESS
*        WHERE THE OUTPUT DATA IS STORED.
*        THE PARAMETER TO THE CHECK MACRO IS THE DECB NAME FOR THE
*        WRITE STATEMENT.
*        .         .         .         .         .         .         .
*
         WRITE DECB1,SF,OTPT,AREA
         SPACE 5
         CHECK DECB1
         PRINT NOGEN
         LA    R9,1(R9)            ADD ONE TO KEY COUNT
         B     LOOP                RETURN FOR NEXT INPUT
*
*        .         .         .         .         .         .         .
*        AT THIS POINT THE KEY JUST READ IS GREATER THAN THE MIN VALUE
*        IN REGISTER 9.   NOW CHECK THE EXPECTED KEY VALUE TO SEE IF
*        IT IS GREATER THAN THE MAX KEY VALUE IN REGISTER 8.
*        .         .         .         .         .         .         .
*
DUMMY    CR    R9,R8               CHECK TO SEE IF LAST INPUT
         BH    EOJ                 IF HIGH THEN DONE
         PRINT GEN
*
*        .         .         .         .         .         .         .
*        AT THIS POINT THE LAST KEY READ IN WASA GREATER THAN MIN KEY
*        VALUE BUT LWSS THAN MAX KEY VALUE, THEREFORE, WRITE A DUMMY
*        RECORD TO THE DATA SET.
*        .         .         .         .         .         .         .
*
         SPACE 5
*
*        THE FIRST POSITIONAL PARAMETER  IS THE DECB NAME TO BE
*        CREATED.  THE SD INDICATES THAT THIS IS A DUMMY RECORD.
*        OTPT IS THE DCB NAME.  DUMAREA IS ADDRESS OF 5 BYTES OF CORE
*        FOR USE IN DUMMY OUTPUT.
*        INCREASE THE MIN KEY VALUE AND RETURN TO CHECK THE CURRENT
*        KEY VALUE.
*        AGAIN CHECK IS USED TO CHECK THE DECB.
*        .         .         .         .         .         .         .
*
         WRITE DECB2,SD,OTPT,DUMAREA
         SPACE 5
         CHECK DECB2
         PRINT NOGEN
         LA    R9,1(R9)            INCREASE R9 BY ONE
         BR    R7                  GO TO DO NEXT COMPARE
```

```
*
*        .         .         .         .         .         .         .
*        AT THIS POINT WE HAVE JUST READ IN THE LAST INPUT SO GO TO
*        FINISH OUTPUTTING DUMMY RECORDS UNTIL THE DATA SET IS FULL.
*        .         .         .         .         .         .         .
*
EODADD   LA    R7,DUMMY
         BR    R7                  RETURN TO CONTINUE DUMMY OUTPUT
         PRINT GEN
         SPACE 5
*
*        .         .         .         .         .         .         .
*        NOW WE HAVE FILLED THE DATA SET SO CLOSE THE DATA SETS AND RE
*        NOW WE HAVE FILLED THE DATA SET SO CLOSE THE DATA SETS AND
*        RETURN TO OPERATING SYSTEM.
*        .         .         .         .         .         .         .
*
EOJ      CLOSE (INPT,,OTPT,)
         PRINT NOGEN
         XRETURN SA=*
         PRINT GEN
         SPACE 5
*
*        .         .         .         .         .         .         .
*        WHEN AN UNCORRECTABLE ERROR HAS OCCURED IN THE WRITE STATEMEN
*        THIS ROUTINE IS GIVEN COOTROL BY THE CONTROL PROGRAM.
*        SYNADAF RETURNS IN REGISTER 1 THE ADDRESS OF AN AREA THAT
*        CONTAINES DEBUGGNNG INFORMATION SO PRINT IT AND RETURN
*        THE SYNADRLS MACRO RESTORES THE REGISTERS THAT EXISTED
*        WHEN CHECKER RECEIVED CONTROL IT RESTORES THEM FOR YOU.
*        THE SYNAD ROUTINE CAN NOT SAVE IN CALLING PROGRAM SAVEAREA.
*        .         .         .         .         .         .         .
*
CHECKER SYNADAF ACSMETH=BDAM
         PRINT NOGEN
         XPRNT 0(1)
         PRINT GEN
         SPACE 5
         SYNADRLS
         BR    14
AREA     DC    10F'0'
         DC    10F'0'
DUMAREA  DS    CL5
*
*        .         .         .         .         .         .         .
*        THIS IS DCB FOR BDAM OUTPUT TO CREATE BDAM DATA SET.
*        WHEN CREATING THE DATA SET THE DSORG MUST BE PS.
*        THE MACRF MUST BE (WL)
*        DEVD MUST BE DA.
*        DDNAME IS STANDARD.
*        SYNAD IS ADDRESS OF ROUTINE TO BE GIVEN CONTROL WHEN ERROR
*        BURING EXECUTION OF THE I/O OCCURS.
*        .         .         .         .         .         .         .
*
OTPT     DCB   DSORG=PS,MACRF=(WL),DDNAME=DAOUTPUT,DEVD=DA,            X
               SYNAD=CHECKER
         PRINT NOGEN
INPT     DCB   DSORG=PS,MACRF=GM,LRECL=80,BLKSIZE=80,RECFM=F,          X
               DDNAME=INPUT,EROPT=ACC,EODAD=EODADD
         LTORG
         PRINT GEN
```

```
         END
/*
//DATA.DAOUTPUT DD DSNAME=&&TEMP,UNIT=SYSDA,DISP=(NEW,PASS),           X
// DCB=(DSORG=DA,BLKSIZE=40,KEYLEN=4,RECFM=F),SPACE=(44,(21))
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.XSNAPOUT DD SYSOUT=A
//DATA.INPUT DD *
1001FIRST RECORD
1003THIRD RECORD
1005RECORD FIVE
1007SEVENTH RECORD
1009NINETH RECORD
1011ELEVENTH RECORD
1013THIRTEENTH RECORD
1015FIFTEENTH RECORD
1017SEVENTEENTH RECORD
1019NINETEENTH RECORD
/*
// EXEC ASGCG
//SOURCE.SYSGO DD DISP=(OLD,PASS,DELETE)
//SOURCE.INPUT DD *
*        .         .         .         .         .         .         .
*
*        THIS CSECT ADDS RECORD TO THE DATA SET THAT ALREADY EXISTS.
*        .         .         .         .         .         .         .
*
         PRINT NOGEN
         EQUREGS
SECOND   CSECT
         XSAVE
*
*        .         .         .         .         .         .         .
*        FIRST OPERN THE INPUT AND OUTPUT DATA SETS.  THEN SET REGSTER
*        11 TO 1000 WHICH IS USED TO RELATIVE BLOACK ADDRESS FOR
*        THE RECORDS TO BE ADDED TO THE DATA SET.
*        .         .         .         .         .         .         .
*
         OPEN  (INPT,INPUT,DIRECT,OUTPUT)
         LA    R11,1000
*
*        .         .         .         .         .         .         .
*        READ THE INPUT AND CONVERT THE DEY TO BINARY FROM IN REGISTER
*        2, THEN COMPUTE THE RELATIVE BLOCK ADDRESS AND STORE THIS AT
*        REF.
*        THEN WRITE THE NEW RECORD IN THE PROPER PLACE ON DISK.
*        .         .                   .         .         .         .
*        .         .         .         .         .         .         .
*
NEXTREC  GET   INPT,KEY
         XDECI R2,KEY
         SR    R2,R11
         ST    R2,REF
         PRINT GEN
         SPACE 5
*
*        .         .         .         .         .         .         .
*        THE POSITIONAL PARAMETER ARE
*        1         NAME OF THE DATA EVENT CONTROL BLOCK TO BE CREATED
*        BY THE WRITE MACRO.
*        2         TYPE DA  ADD A NEW BLOCK WHEREEVER THERE IS SPACE;
*        THE SEARCH FOR AVAILBBLE SPACE STARTS ATTHE DEVICE ADDRESS
```

```
*        ADDRESS IN THE BLOCK ADDRESS OPERAND..  TYPE SEARC IS
*        IN THE OPTCD PARAMETER IN DCB MACRO.
*        3         DCB NAME HERE IT DIRECT.
*        4         AREA ADDRESS ADDRESS OF MAIN CORE CONTAINING THE
*        BLOCK TO BE WRITTEN.
*        5         LENGTH - NUMBER OF BYTE TO BE WRITTEN, 'S' INDI-
*        CATES THAT THIS IS OBTAINED FROM BLKSIZE IN DCB.
*        6         KEY ADDRESS IS ADDRESS OF MAIN CORE AREA CONTAIN-
*        ING THE KEY.
*        7         BLOCK ADDRESS  ADDRESS OF THREE BYTES IN AMIN CORE
*        THAT CONTAIN THE RELATIVE BLOCK ADDRESS.
*        .         .         .         .         .         .         .
*        WAIT MACRO HALTS TASK TILL I/O DONE.
*        THEN RETURN FOR NEXT INPUT.
*        .         .         .         .         .         .         .
*
         WRITE DECB,DA,DIRECT,DATA,'S',KEY,REF+1
         PRINT NOGEN
         PRINT GEN
         SPACE 5
         WAIT  ECB=DECB
         PRINT NOGEN
         B     NEXTREC             RETURN FOR NEXT INPUT
*
*        .         .         .         .         .         .         .
*        CLOSE INPUT FILE  CLOSE OUTPUT FILE AND RETURN TO SYSTEM.
*        .         .         .         .         .         .         .
*
EOJ      CLOSE (INPT,,DIRECT,)
         XRETURN SA=*
         PRINT GEN
         SPACE 5
*
*        .         .         .         .         .         .         .
*        THIS IS THE ERROR REUTINE FOR DEBUGGING.
*        .         .         .         .         .         .         .
*
CHECKER SYNADAF ACSMETH=BDAM
         PRINT NOGEN
         XPRNT 0(1)
         PRINT GEN
         SYNADRLS
         BR    14
KEY      DS    F
DATA     DS    CL40
         DC    CL100' '
REF      DS    F
         PRINT NOGEN
INPT     DCB   DSORG=PS,MACRF=GM,LRECL=80,RECFM=F,BLKSIZE=80,          X
               EROPT=ACC,EODAD=EOJ,DDNAME=INPUT
         PRINT GEN
         SPACE 5
*
*        .         .         .         .         .         .         .
*        THE KEYWORD PARAMETERS FOR DCB ARE
*        DSORG IS DA SPECIFYING DIRECT ACCESS.
*        RECFM IS FIXED
*        KEYLEN - KEY LENGTH IS 4 BYTES.
*        BLKSIZE IS 40 BYTES.
*        MACRF - WRITE BLOCKS ARE TO BE ADDED TO THE DATA SET.
*        OPTCD - E EXTENDED SEARCH  R  RELATIVE BLOCK ADDRESSING
```

```
*        LIMCT  NUMBER OF BLOCKS TO BE SERRCHED FOR EKY.
*        DDNAME - STANDARD.
*        SYNAD - ADDRESS OF ROUTINE FOR ERROR DURING WRITE.
*        .         .         .         .         .         .         .
*
DIRECT   DCB   DSORG=DA,RECFM=F,KEYLEN=4,BLKSIZE=40,MACRF=(WA),        X
               OPTCD=ER,LIMCT=3,DDNAME=DIRADD,SYNAD=CHECKER
         LTORG
         END
/*
//DATA.DIRADD DD DSNAME=&&TEMP,UNIT=SYSDA,DISP=(OLD,PASS),             X
// SPACE=(44,(21))
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.XSNAPOUT DD SYSOUT=A
//DATA.INPUT DD *
1002SEDOND RECORD
1004FOURTH RECORD
1006SIXTH RECORD
1008EIGHT RECORD
1010TENTH RECORD
1012TWELTH RECORD
1014FOURTEENTH RECORD
1016SIXTEENTH RECORD
1018EIGHTEENTH RECORD
1020TWENTIETH RECORD
/*
// EXEC ASGCG
//SOURCE.SYSGO DD DISP=(OLD,PASS)
//SOURCE.INPUT DD *
*
*        .         .         .         .         .         .         .
*        THIS CSECT UPDATES THE DATA SET.
*        .         .         .         .         .         .         .
*
*
*        .         .         .         .         .         .         .
*        OPEN THE DATA SET AND SET REGISTER 11 TO 1001 WHICH IS USED
*        TO COMPUTE THE RELATIVE BLOCK ADDRESS.
*        .         .         .         .         .         .         .
*
         PRINT NOGEN
         EQUREGS
THIRD    CSECT
         XSAVE
         OPEN  (INPT,INPUT,DIRECT,OUTPUT)
         LA    R11,1001
*
*        .         .         .         .         .         .         .
*        READ IN TEH INPUT AND CONVDRT THE KEY TO A RELATIVE
*        BLOCK ADDRESS. THEN STORE RELATIVE BLOCK ADDRESS IN REF.
*        READ THE RECORD WITH THIS RELATIVE ALOCK ADDRESS.
*        THE WRITE THE NEW INPUT IN ITS PLACE THEN RETURN FOR NEXT
*        INPUT.
*        .         .         .         .         .         .         .
*
LOOP     GET   INPT,KEY
         XDECI R2,KEY
         SR    R2,R11
         ST    R2,REF
         PRINT GEN
         SPACE 5
```

```
*
*       .         .         .         .         .         .         .
*       THE POSITIONAL PARAMETER FOR READ MACRO ARE:
*       1         DECB NAME TO BE CREATED BY READ STATEMENT.
*       2         TYPE - DI - SEARCH FOR RECORD USES BLOCK
*       IDENTIFICATION.
*       3         DCB ADDRESS DIRECT.
*       4         AREA ADDRESS - 'S' INDICATES THAT DYNAMIC BUFFERING
*       IS TO BE USED.
*       5         LENGHT - 'S' NUMBER OF BYTES COMES FROM DCB.
*       6         KEY ADDRESS - 0 INDICATES KEY NOT TO BE READ.
*       7         BLOCK ADDRESS - ADDRESS OF THREE BYTES CONTAINING
*       RELATIVE BLOCK ADDRESS.
*       .         .         .         .         .         .         .
*
        READ  DECB,DI,DIRECT,'S','S',0,REF+1
        SPACE 5
        CHECK DECB
*
*       .         .         .         .         .         .         .
*       LOAD REGISTER 3 WITH BUFFER ADDRESS. THEN MOVE NEW DATA TO
*       BUFFER AND STORE BUFFER ADDRESS IN DCEB OF WRITE STATEMETN.
*       .         .         .         .         .         .         .
*
        L     R3,DECB+12
        MVC   0(30,3),DATA
        ST    R3,DECBW+12
        SPACE 5
*
*                 .
*       .         .         .         .         .         .         .
*       POSITIONAL PARAMETERS OF WRITE MACRO ARE:
*       1         DECB NAME TO BE CREATED BY WRITRE MACRO.
*       2         TYPE - DI - WRITE THE BLOCK AT THE DEVICE ADDRESS
*       PROVIDED AT THE BLOCK ADDRESS OPERAND.  DATA AND KEYS ARE
*       WRITTEN.
*       3         DCB ADDRESS - DIRECT.
*       4         AREA ADDRESS - 'S' INDICATES DYNAMIC BUFFERING
*       ADDRESS PROVIDED IN DECB +12 PREVIOUSLY BY READ AND STORE.
*       5         LENGTH  - - 'S' INDICATES LENGTH CMMES FROM BLKSIZE
*       IN DCB.
*       6         KEY ADDRESS - 0 INDICATES KEY NOT WRITTEM.
*       7         BLOCK ADDRESS ADDRESS OF THREE BYTES OF MAIN CORE
*       CONTIANING RELATIVE BLOCK ADDRESS.
*       .         .         .         .         .         .         .
*
        WRITE DECBW,DI,DIRECT,'S','S',0,REF+1
        SPACE 5
        CHECK DECBW
        B     LOOP
        PRINT NOGEN
*
*       .         .         .         .         .         .         .
*       CLOSE DATA SETS AND RETURN TO OS.
*       .         .         .         .         .         .         .
*
EOJ     CLOSE (INPT,,DIRECT,)
        XRETURN SA=*
*
*       .         .         .         .         .         .         .
*       ERROR ROUTINE PROVIDED BY SYNAD PARM IN DCB.
```

```
*        .         .         .         .         .         .         .
*
CHECKER SYNADAF ACSMETH=BDAM
         XPRNT 0(1)
         SYNADRLS
         BR    14
KEY      DC    F'0'
DATA     DC    10F'0'
         DC    CL100' '
REF      DC    F'0'
INPT     DCB   DSORG=PS,MACRF=GM,LRECL=80,BLKSIZE=80,RECFM=F,          X
               DDNAME=INPUT,EROPT=ACC,EODAD=EOJ
         PRINT GEN
         SPACE 5
*
*        .         .         .         .         .         .         .
*        THE KEYWORD PARAMETERS FOR DCB ARE
*        DSORG DA DIRECT ACCESS.
*        DDNAME STANDARD DIRECTDD
*        MACRF
*                  R  READ
*                  I SEARCH TO BE MADE BY BLOCK IDENTIFICATION.
*                  S  DYNAMIC BUFFERING.
*                  C  CHECK ABSENCE DENOTES WAIT.
*                  W  WRITE
*                  I SEARCH TO BE MADE BY BLOCK IDENTIFICATION.
*                  C  HCECK ABSECCE DENTOES WAIT.
*        BUFL  BUFFER LENGTH 40
*        OPTCD - R - SEARCH TO BE MADE BY BLOCK IDENTIFICATION.
*        BUFNO - NUMBER OF BUFFERS.
*        SYNAD - ERROR ROUTINE ADDRESS.
*        .         .         .         .         .         .         .
*
DIRECT   DCB   DSORG=DA,DDNAME=DIRECTDD,MACRF=(RISC,WIC),BUFL=40,      X
               OPTCD=R,BUFNO=1,SYNAD=CHECKER
         LTORG
         END
/*
//DATA.XSNAPOUT DD SYSOUT=A
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.DIRECTDD DD DSNAME=&&TEMP,DISP=(OLD,PASS),SPACE=(44,(21)),      X
// UNIT=SYSDA
//DATA.INPUT DD *
1020CGANGE RECORD TWENTY
1002CHANGE RECORD TWO
1010CHANGE RECORD TEN
/*
// EXEC ASGCG
//SOURCE.SYSGO DD DISP=(OLD,PASS)
//SOURCE.INPUT DD *
*
*        .         .         .         .         .         .         .
*        THIS JOB STEP READS IN THE CURRENT RECORDS AND PRINTS THEM
*        OUT.
*        .         .         .         .         .         .         .
*
         PRINT NOGEN
         EQUREGS
FOUR     CSECT
         XSAVE
*
```

```
*        OPEN INPUT AND OUTPUT DATA SETS.
*        SET R11 TO NUMBER OF BLOCKS
*        SET R3 TO 0 THE RELATVVE ADDRESS OF FIRST BLOCK.
*        STORE R3 AT REF.
*        .         .         .         .         .         .         .
*
         OPEN  (INPT,INPUT,OTPT,OUTPUT)
         LA    R11,20
         SR    R3,R3
         ST    R3,REF
         PRINT GEN
         SPACE 5
```

```
*
*        .         .         .         .         .         .
*        THE POSITIONAL PARAMETERS FOR THE READ MACRO ARE
*        1         DECB NAME TO BE CREATED BY READ MACRO.
*        2         TYPE DIRECT ADDRESS ING BY RELATIVE BLOCK.
*        3         DCB ADDRESS NNPT
*        4         AREA ADDRESS AREA WHERE RETRIEVED DATA TO BE PUT.
*        5         LENGTH - 'S' TO BE TAKEN FROM DCB.
*        6         KEY ADDRESS - 0 INDICATES KEY NOT TO BE READ.
*        7         BLOCK ADDRESS ADDRESS OF THREE BYTES OF CORE
*        CONTAINING THE RLLATIVE KEY ADDRESS.
*        .         .         .         .         .         .
*
LOOP     READ  DECB,DI,INPT,INPTT,'S',0,REF+1
         SPACE 5
         CHECK DECB
*
*        .         .         .         .         .         .
*        INCREASE RELATIVE BLOCK ADDRESS, THEN OUTPUT THE RECORD
*        RETURN FOR NEXT INPUT.
*        .         .         .         .         .         .
*
         L     R3,REF
         LA    R3,1(R3)
         ST    R3,REF
         PRINT NOGEN
         PUT   OTPT,INPTT-1
         BCT   R11,LOOP
*
*        .         .         .         .         .         .
*        CLOSE DATA SETS AND RETURN TO OS.
*        .         .         .         .         .         .
*
         CLOSE (INPT,,OTPT,)
         XRETURN SA=*
*
*        .         .         .         .         .         .
*        ERROR REOUTINE FOR USE WHEN ERROR OCCURS.
*        .         .         .         .         .         .
*
CHECKER SYNADAF ACSMETH=BDAM
         XPRNT 0(1)
         SYNADRLS
         BR    14
REF      DC    F'0'
         DC    X'00'
INPTT    DC    10F'0'
OTPT     DCB   DSORG=PS,MACRF=PM,LRECL=40,BLKSIZE=40,RECFM=FA,         X
               DDNAME=FT06F001,EROPT=ACC
         PRINT GEN
         SPACE 5
INPT     DCB   DSORG=DA,MACRF=(RIC),RECFM=F,BLKSIZE=40,                X
               OPTCD=ER,LIMCT=3,DDNAME=DIRECTDD,SYNAD=CHECKER
         LTORG
         END
/*
//DATA.FT06F001 DD SYSOUT=A
//DATA.DIRECTDD DD DSNAME=&&TEMP,DISP=(OLD,DELETE),SPACE=(44,(21)),    X
// UNIT=SYSDA
//DATA.SYSUDUMP DD SYSOUT=A
```

```
//*
//*      THIS PROGRAM IS AN EXAMPLE OF THE BPAM ACCESS METHOD
//*      IT USES THE BLDL MACRO AND THE TWO KINDS OF FIND MACROS
//*      TIMING AND RECORD CONSIDERATIONS: 40 SECONDS, 2500 RECORDS
//*
// EXEC ASGCG,PARM='NOXREF'
//SYSIN DD *
*        THIS PROGRAM IS DESIGNED TO ILLUSTRATE THE USE OF THE
*        PARTITIONED ACCESS METHOD.  IT HAS TWO SECTIONS;
*        SECTION 1,
*              USES THE BLDL MACRO INSTRUCTION TO BUILD A LIST THAT
*              CONTAINS A RELATIVE TRACK ADDRESS FOR EACH USER CREATED
*              ENTRY.  THE LIST MUST BEGIN ON A HALF-WORD BOUNDRY WITH
*              A USER SUPPLIED FULL-WORD OF CONTROL INFORMATION:
*              HALF-WORD 1 -- CONTAINS THE COUNT OF LIST ENTRIES (MAX.
*                             12)
*              HALF-WORD 2 -- CONTAINS THE LENGTH (IN BYTES, MAX. 76)
*                             OF EACH ENTRY
*              THE USER SUPPLIES AT LEAST 14 CONTIGUOUS BYTES FOLLOW-
*              ING THE LIST DISCRIPTOR FULL-WORD FOR EACH LIST ENTRY.
*              BYTES 0-7 CONTAIN THE MEMBER NAME LEFT JUSTIFIED AND
*              RIGHT PADDED WITH BLANKS IF NECESSARY.  BYTES 8-13 ARE
*              LEFT UNALTERED.  THE BLDL MACRO INSTRUCTION COMPLETES
*              EACH LIST ENTRY AND SUPPLIES THE RELATIVE TRACK ADDRESS
*              AND BLOCK NUMBER ON THAT TRACK OF THE MEMBER NAME, THE
*              CONCATENATION NUMBER, WHERE FOUND (PRIVATE,LINK,OR JOB
*              LIBRARIES), WHETHER ENTRY NAME IS A MEMBER NAME OR AN
*              ALIAS, AMOUNT AND TYPE OF USER DATA IN THE PDS DIRECTORY
*              ENTRY.  THIS INFORMATION IS PLACED IN BYTES 8-13 OF EACH
*              LIST ENTRY.  A FIND MACRO INSTRUCTION IS ISSUED WHICH
*              CONVERTS THE RELATIVE ADDRESSES IN THE BLDL LIST INTO
*              ABSOLUTE ADDRESSES AND INSERTS THEM INTO THE DCB (THIS
*              ALLOWS SUBSEQUENT READS/WRITES OR GETS/PUTS TO DEAL
*              WITH THE DESIRED MEMBER).
*
*              *NOTE*  ALL MEMBER ENTRIES MUST BE IN ALPHAMERIC ORDER
*        SECTION 2,
*              ISSUES A FIND MACRO INSTRUCTION ONLY THIS TIME THE FIND
*              IS DIRECTED TO DO THE PDS DIRECTORY SEARCH ITSELF.  THE
*              RELATIVE ADDRESS IS CONVERTED AS IN SECTION 1 AND
*              INSERTED INTO THE DCB.
*
*        THIS PROGRAM USES MACROS:
*              XSAVE
*              XRETURN
*              XPRNT
*              READ
*              BLDL
*              FIND
*              CHECK
*              LISTHD (PROGRAM LOCAL)
*              LIST (PROGRAM LOCAL)
*              DCB
*              DCBD
*        EACH OCCURRANCE IS EXPLAINED IN THE PROGRAM TEXT.
*        REGISTER USAGE IS EXPLAINED IN THE EQUATE SECTION.
*
*
*        PROGRAM LOGIC:
*
*              THIS PROGRAM RECOVERS FROM THE SYS1.MACLIB AND THE
```

```
*              CMACLIB PROGRAM LIBRARIES THE DEFINITIONS FOR FOUR
*              SYSTEM MACROS: CALL, RETURN, SAVE, AND DCB.
*              MACROS CALL, RETURN, AND SAVE ARE RECOVERED
*              USING THE BLDL-FIND MACRO COMBINATION.  THE DCB MACRO
*              IS RECOVERED USING THE 'D' TYPE FIND MACRO INSTRUCTION.
*              THE PROGRAM BUILDS THE BLDL LIST FOR CALL, RETURN,
*              AND SAVE, AND DOES THE BLDL ON THAT LIST.  A LOOP
*              IS ENTERED THAT DOES A FIND ('C' TYPE) ON THE FIRST
*              LIST ENTRY.  AN INNER LOOP STARTS UP THAT READS A
*              BLOCK OF DATA AND DOES DE-BLOCKING AND PRINTS THE
*              DATA.  ON E-O-F, THE LOOP FOOT (AEODAD) IS ACTIVATED
*              WHICH TESTS TO SEE IF THE BLDL LIST HAS BEEN PROCESSED
*              IF YES, THE PROGRAM DOES THE 'D' TYPE FIND FOR THE
*              DCB MACRO LISTING AND READS, DE-BLOCKS, AND PRINTS
*              UNTIL IT IS FINISHED, THE PROGRAM THEN TERMINATES.
*
*       PROGRAM BY: RICHARD FORD
*                   JULY,1972
*
        EJECT
        MACRO
        LISTHD  &NAME,&NUM,&LNGTH
        GBLA  &LEN
        SPACE 2
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                       *
*       THIS MACRO EXPANSION WILL CREATE THE LIST DISCRIPTOR FIELD       *
*       AS REQUIRED BY THE BLDL MACRO INSTRUCTION                       *
*                                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        SPACE 2
.*      ROUND GLOBAL VARIABLE &LNGTH TO EVEN NUMBER IF NECESSARY
&LEN    SETA  (((&LNGTH+1)/2)*2)
.*
.*      ATTACH BLDL LIST TO &NAME
&NAME   DS    0H .                ALIGN TO HALF-WORD BOUNDRY AND
*                                 STICK TO &NAME
        DC    H'&NUM' .           NUMBER OF BLDL LIST ENTRIES
        DC    H'&LNGTH' .         DEFINE LENGTH IN BYTES OF EACH
.*                                ENTRY
        MEND
        SPACE 10
        MACRO
        LIST  &ENTRY
        GBLA  &LEN
        LCLA  &T1
        SPACE 2
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*       THIS MACRO CAUSES EACH LIST ENTRY TO BE PLACED INTO THE BLDL    *
*       LIST AND TO ALLOCATE STORAGE FOR EACH ENTRY INTO WHICH THE      *
*       CONTROL PROGRAM WILL PLACE ANY DIRECTORY INFORMATION IT CAN     *
*       FIND ABOUT EACH ENTRY.  EACH ENTRY IS &LNGTH BYTES LONG.        *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        SPACE 2
.*      SET &T1 TO &LNGTH-8 FOR CORRECT FILL LENGTH AFTER ENTRY NAME
.*      DEFINITION
&T1     SETA  (&LEN-8)            GET LENGTH -8
.*
        DC    CL8'&ENTRY' .       ENTRY NAME IN CHARACTER, LEFT-
*                                 JUSTIFIED AND RIGHT PADDED WTH BLANK
        DC    XL&T1'FF' .         GENERATE CORRECT ENTRY LENGTH
```

```
         MEND
         TITLE  'BPAM I/O EXAMPLE'
*        THE DCB DSECT IHADCB FOLLOWS
         SPACE 2
         DCBD  DSORG=PO,DEVD=DA
         SPACE 2
BPAMIO   CSECT
         SPACE 2
         PRINT NOGEN
BPAM     XSAVE TR='BPAM I/O EXAMPLE'
         PRINT GEN
         SPACE 5
*        MNEMONIC REGISTER EQUATES FOLLOW
         SPACE 2
R1       EQU   1                  WORK REGISTER
DCBADD   EQU   2                  DCB ADDRESS POINTER
FNDADD   EQU   3                  TRACK ADDRESS POINTER
DECB     EQU   4                  ALWAYS POINTS TO THE DECB
AREA     EQU   5                  ALWAYS HAS INTERNAL BUFFER ADDRESS
ONE      EQU   6                  USEFUL CONSTANT 1
HEADING  EQU   7                  EACH PRINT HEADING ADDRESS
NEXTHD   EQU   8                  OFFSET TO NEXT HEADING
BLDLCNT  EQU   9                  COUNTING REGISTER FOR PROGRAM CNTRL
RCRDPT   EQU   10                 RECORD POINTER REGISTER
END      EQU   11                 BUFFER END POINTER
BASE     EQU   12                 BASE REGISTER
SAVE     EQU   13                 SAVE AREA POINTER
RETADD   EQU   14                 RETURN ADDRESS POINTER
ENTRY    EQU   15                 ENTRY POINT ADDRESS POINTER
         SPACE 5
*        INITIALIZE REGISTERS WHERE POSSIBLE
         SPACE 2
         LA    DCBADD,BPAMDCB      GET DCB ADDRESS
         LA    FNDADD,BLDLLIST+12  GET BLDL RELATIVE ADDRESS
         LA    DECB,MAC            GET DECB ADDRESS
         LA    AREA,BUFFER         GET INTERNAL BUFFER AREA ADDRESS
         LA    ONE,1               USEFULL CONSTANT
         LA    HEADING,CALLHD      FIRST PAGE HEADING ADDRESS
         LA    NEXTHD,81           OFFSET TO NEXT HEADING
         LA    BLDLCNT,0           INITIALIZE COUNT
         B     OPDCB               SKIP BLDL LIST SPACE
         SPACE 5
*        NEXT, USE OUR MACROS TO GENERATE BLDL LIST SPACE
         SPACE 2
*        GENERATE LIST HEAD(DISCRIPTOR WORD)
         LISTHD  BLDLLIST,3,76
*        GENERATE EACH LIST ENTRY
         LIST  CALL
         SPACE 5
         LIST  RETURN
         SPACE 5
         LIST  SAVE
         SPACE 5
*        OPEN SYS1.MACLIB AND CMACLIB DATA SETS
         SPACE 2
OPDCB    OPEN  ((DCBADD),(INPUT))
         TM    BPAMDCB+48,X'10'    DID DCB GET OPEN OK
         BNO   EXIT                IF NOT OPEN QUIT
         SPACE 5
*        CONSTRUCT BLDL LIST WITH THE BLDL MACRO INSTRUCTION
         SPACE 2
```

```
         BLDL  (DCBADD),BLDLLIST
         SPACE 5
*        ALL INITIALIZATION AND PROGRAM SETUP IS DONE
*        BEGIN BY PRINTING THE FIRST HEADING TO LABEL OUR OUTPUT
         SPACE 2
         PRINT NOGEN
CFIND    XPRNT 0(HEADING),81       PRINT THE HEADING
         XPRNT THRESKP,3           SKIP SOME LINES
         SPACE 5
         PRINT GEN
*        DO FIND ON EACH LIST ENTRY TO RECOVER THE ACTUAL TRACK
*        ADDRESS OF THE MEMBER INTO THE DCB SO THAT IT CAN BE
*        ACCESSED BY QSAM(GET/PUT) OR BSAM(READ/WRITE)
         SPACE 2
         FIND  (DCBADD),(FNDADD),C
         SPACE 2
*        THE 'C' PARAMETER INDICATES TO THE FIND MACRO THAT IT IS
*        DEALING WITH THE LIST PRODUCT OF THE BLDL MACRO INSTRUCTION
*        EXECUTION
         SPACE 5
*        NOW READ THE FIRST BLOCK OF THE RETRIEVED DATA INTO SOME
*        INTERNAL BUFFER AREA SO THAT IT CAN BE HANDLED IN SOME
*        USEFULL FASHION
         SPACE 2
BREAD    READ  MAC,SF,(DCBADD),(AREA),'S'
         SPACE 5
*        NOW CHECK THE COMPLETION OF THE I/O INITIATED BY THE
*        READ AS NO DE-BLOCKING OF THE RECOVERED DATA CAN BE DONE
*        UNTIL THE TRANSFER IS COMPLETE
         SPACE 2
         CHECK (DECB)
         SPACE 2
*        THE READ RETURNS THE BLOCK SIZE INTO THE DCB FOR USE
*        IN RECORD DEBLOCKING AND OTHER DATA MANAGEMENT
         SPACE 5
         USING IHADCB,DCBADD       NOTE USING ON DCB DSECT
         SPACE 5
*        NEXT FOUR  INSTRUCTIONS ALLOW THE PROGRAM TO CORRECTLY
*        ACCESS NEXT BUFFER FULL OF INFORMATION ALSO TO DETECT
*        SHORT BLOCKS AT END OF THE CURRENT MEMBER
         SPACE 2
         LH    END,DCBBLKSI        GET BLOCK SIZE FROM DCB
         L     ENTRY,16(DECB)      GET IOB ADDRESS
         SH    END,14(ENTRY)       GET RELATIVE END OF NEW BUFFER
         AR    END,AREA            GET ABSOLUTE ENDING ADDRESS
         SPACE 2
         LR    RCRDPT,AREA         COPY AREA POINTER INTO RECORD
*                                   POINTER REGISTER
DPRINT   MVC   PRNTAREA+1(80),0(RCRDPT)  DE-BLOCK RECORDS INTO
*                                   81 BYTE PRINTING BUFFER
         PRINT NOGEN
         XPRNT PRNTAREA,81         PRINT THE LOGICAL RECORD
         PRINT GEN
         SPACE 5
         LA    RCRDPT,80(RCRDPT)   UPDATE POINTER BY 80 BYTE LRECL
         CR    RCRDPT,END          FIND OUT IF BUFFER IS EMPTY
         BNL   BREAD               IF AT END READ NEW BLOCK
         B     DPRINT              OTHERWISE-CONTINUE PRINTING
*                                   OUT OF BUFFER AREA
         SPACE 10
AEODAD   AR    BLDLCNT,ONE         INCREMENT MEMBER COUNTER
```

```
         C     BLDLCNT,=F'3'       IS BLDL SECTION COMPLETE
         BL    UPDATE              YES-GO TO UPDATE FIND MACRO
*                                  POINTER
         TM    TERMFLAG,X'FF'      OTHERWISE-IS TERMINATION FLAG
*                                  SET
         BNO   DRECTSCH            NOT SET-GO FOR DIRECTORY
*                                  SEARCH EXAMPLE
         SPACE 2
*        OTHERWISE IT IS TIME TO END EXECUTION  SO CLOSE THE
*        OPEN DATA SET AND RETURN TO THE SYSTEM
         SPACE 2
EXIT     CLOSE (DCBADD)
         SPACE 2
*        TERMINATE NORMALLY
         SPACE 2
         PRINT NOGEN
         XRETURN  SA=*
         PRINT GEN
         SPACE 10
UPDATE   LA    FNDADD,76(FNDADD)   UPDATE POINTER TO NEXT LIST
*                                  ENTRY
         AR    HEADING,NEXTHD      UPDATE HEADING POINTER REG
*                                  TO NEXT HEADING
         B     CFIND               DO NEXT FIND, ETC.
         SPACE 5
*        FOLLOWING SECTION DOES A DIFFERENT TYPE OF FIND MACRO
*        INSTRUCTION WHICH FORCES A SEARCH OF THE DATA SETS
*        DIRECTORY IMMEDIATELY.  IT TENDS TO BE SOMEWHAT LESS
*        LESS EFFICIENT(SLOWER) THAN A BLDL-FIND COMBINATION,
*        BUT MAY BE EASIER TO ORGANIZE AND CODE.
         SPACE 5
DRECTSCH OI    TERMFLAG,X'FF'      SET TERM FLAG TO STOP
         SPACE 5
         FIND  (DCBADD),NAME,D
         SPACE 2
*        NOTE THAT THE 'C' PARAMETER IN THE FIND MACRO
*        HAS CHANGED TO A 'D'.  THIS INDICATES TO THE SYSTEM
*        THAT THE MEMBER NAME EXISTS ON A DOUBLE-WORD
*        BOUNDRY AND IS A DOUBLE-WORD, LEFT-JUSTIFIED AND
*        RIGHT PADDED WITH BLANKS IF NECESSARY.
         SPACE 2
         PRINT NOGEN
         XPRNT DISRCH,81           PRINT NEW HEADING
         XPRNT THRESKP,3           SKIP SOME LINES
         PRINT GEN
         B     BREAD               GO BACK TO PROCESS AS BEFORE
         SPACE 2
         LTORG                     START LITERAL POOL HERE
         SPACE 2
         DS    0D                  GET DOUBLE-WORD ALIGNMENT
NAME     DC    CL8'GET'            DEFINE NAME FOR FIND
        SPACE 5
BPAMDCB  DCB   DSORG=PO,           INDICATE PARTITIONED DATA SET       X
               DDNAME=MACLIB,      LOGICALLY CONNECT TO DD CARD        X
               EODAD=AEODAD,       ON END-OF-DATA GO HERE              X
               MACRF=R             INDICATE MACRO TYPE
*        ALL OTHER REQUIRED INFORMATION WILL BE TAKEN FROM
*        EITHER THE DD CARD OR THE DATA SET LABEL.
         SPACE 5
CALLHD   DC    CL81'3********** CALL MACRO LISTING'
         DC    CL81'1********** RETURN MACRO LISTING'
```

```
         DC    CL81'1********** SAVE MACRO LISTING'
         SPACE 2
DISRCH   DC    CL81'1********** GET MACRO LISTING USING DIRECT FIND'
PRNTAREA DC    CL1' '              PRINTING BUFFER
         DS    80C                 RECORD SPACE
TERMFLAG DC    X'00'               INITIALIZE TERM FLAG TO OFF
THRESKP  DC    CL3'3  '            LINE SKIPPER
         CNOP  0,8                 BUFFER ALIGNMENT AND SPACE
BUFFER   DS    500D
         END   BPAMIO
/*
//DATA.MACLIB  DD  DSN=SYS1.MACLIB,DISP=SHR,UNIT=2314,VOL=SER=PSU01
// DD  DSN=CMACLIB,DISP=SHR,UNIT=2314,VOL=SER=PSU02
/*LOG
/*DUMP
//DATA.SYSUDUMP DD SYSOUT=A
/*
```

```
//*
//*
//*
//*       THIS PROGRAM DEMONSTRATES THE BSAM I/O TECHNIQUES.
//*             1.  USES MACROS READ, WRITE, CHECK, POINT, DCB, OPEN,
//*               CLOSE, AND A READ-POINT READ FORM.
//*             2.  WE WILL READ A CARD FROM THE READER, THEN ECHO
//*               PRINT IT TO THE PRINTER.  THE CARD IMAGE IS THEN
//*               WRITTEN TO DISK USING THE READ-POINT FORM .
//*                 AT EOD, A POINT MACRO IS ISSUED TO REPOSITION THE
//*               DISK SO THAT THE FIRST RECORD CAN BE RECOVERED.
//*                 AT EOD FROM THE DISK, THE DCBS ARE CLOSED AND
//*               CONTROL IS RETURNED TO THE CALLING PROGRAM.
//*             3.  TIMEING CONSIDERATIONS:  50 SECONDS,1000 RECORDS
//*
//*
//*
// EXEC ASGCG,PARM.DATA=MAP
//SOURCE.INPUT DD *
         TITLE 'BSAM I/O EXAMPLE'
BSAMIO   CSECT
*                                      SET STANDARD OS LINKAGE
         XSAVE TR=NO
         SPACE 5
*        OPEN ALL DCBS, CARDCB TO READ DATA, PRNTDCB TO ALLOW WRITE
*        TO THE PRINTER, AND THE INTERMEDIATE DISK STORAGE.  OUTIN
*        OPTION IN OPEN MACRO ALLOWS A WRITE OPERATION FOLLOWED BY
*        A READ OPERATION WITHOUT AN ADDITIONAL OPEN-CLOSE SET.
         SPACE 5
OPENDCBS OPEN  (CARDCB,(INPUT),PRNTDCB,(OUTPUT),DISKDCB,(OUTIN))
         SPACE 5
*        TEST THE OPEN, IF OPENS DID NOT GO,  TERMINATE WITH ABEND
*        GIVING USER 500 COMPLETION CODE
         SPACE 5
         TM    CARDCB+48,X'10'          DID THE CARD READER OPEN GO
         BNO   ABTERM                   ABNORMALLY TERMINATE
         TM    PRNTDCB+48,X'10'         DID PRINTER OPEN GO
         BNO   ABTERM                   ABNORMALLY TERMINATE
         TM    DISKDCB+48,X'10'         DID DISK OPEN GO
         BNO   ABTERM                   ABNORMALLY TERMINATE
         SPACE 5
*        READ A CARD ON THE READER
         SPACE 5
         LA    5,0                      SET UP CARD COUNTER
READ     READ  CARDECB,SF,CARDCB,BLOCK      READ A CARD IMAGE INTO BLOC
         SPACE 5
*        ISSUE CHECK TO TEST COMPLETION OF I/O OPERATION
         SPACE 5
         CHECK CARDECB                  CHECK FOR I/O COMPLETION
         SPACE 5
         LA    5,1(5)                   COUNT CARDS COMMING IN
*        DO A WRITE TO PRINTER FOR ECHO PRINT
         SPACE 5
         WRITE DISKDECB,SF,DISKDCB,BLOCK    WRITE TO DISK FROM BLOCK
         SPACE 5
*        ISSUE CHECK AS ABOVE
         SPACE 5
         CHECK DISKDECB                 IS I/O OPERATOON COMPLETE
         SPACE 5
*        SAVE CARD IMAGE ON DISK
         SPACE 5
```

```
         WRITE PRNTDECB,SF,PRNTDCB,BLOCK-1  WRITE TO THE PRINTER
         SPACE 5
*        CHECK AS BEFORE
         SPACE 5
         CHECK PRNTDECB               IS I/O OPERATION COMPLETE
         SPACE 5
         B     READ                   LOOP TO GET ALL DATA
         SPACE 5
*        PRINT A LITERAL INDICATING EOD ON CARD READER,  BEGIN
*        READING FROM THE DISK
         SPACE 5
ENDATA   WRITE PRNTDEC1,SF,PRNTDCB,LITERAL
         SPACE 5
*        ISSUE CHECK AS ABOVE
         SPACE 5
         CHECK PRNTDEC1
         SPACE 5
*        SINCE DISK DCB WAS NOT CLOSED AT EOD A POINT MACRO MUST
*        BE ISSUED TO REPOSITION THE THE DISK AT THE FIRST RECORD
         SPACE 5
         POINT DISKDCB,POINT          REPOSITION DISK TO GET 1ST REC.
         SPACE 5
*
*        NOW READ FROM DISK AND WRITE TO PRINTER  USING SAME DATA
*
         SPACE 5
DISKRD   READ  DISKDEC1,SF,DISKDCB,BLOCK
         SPACE 5
*        CHECK FOR END OF I/O OPERATION
         SPACE 5
         CHECK DISKDEC1               CHECK I/O COMPLETION
         SPACE 5
*        WRITE TO PRINTER AS CARDS COME OFF DISK
         SPACE 5
         WRITE PRNTDEC2,SF,PRNTDCB,BLOCK-1
         SPACE 5
*        CHECK I/O DONE
         SPACE 5
         CHECK PRNTDEC2               I/O COMPLETION CHECK
         SPACE 5
         BCT   5,DISKRD               WRITE ONLY THE CARDS THERE
         SPACE 5
END      CLOSE (CARDCB,,PRNTDCB,,DISKDCB)  CLOSE ALL ACTIVE DCBS
*                                     SET UP RETURN TO CALLER (OS)
         XRETURN SA=*,TR=NO
         SPACE 5
ABTERM   ABEND 500,DUMP               GET ABEND DUMP IN CASE OF TROUBL
         DC    0F'0'
POINT    DC    X'00000100'            CONTROL WORD FOR POINT MACRO
         DC    C' '                   ASA CARRIAGE CONTROL CHAR.
BLOCK    DC    80C' '                 BUFFER AREA
LITERAL  DC    C'-REPEAT READ DATA FROM DISK',80C' '
         SPACE 5
CARDCB   DCB   RECFM=F,               DATA FORMAT IS FIXED           X
               BLKSIZE=80,            PHYSICAL BLOCK SIZE IS 80 BYTES X
               LRECL=80,              LOGICAL RECORD SIZE IS 80 BYTES X
               DDNAME=INPUT,          LOGICALLY CONNECT TO INPUT DATA S
               EODAD=ENDATA,          ON EOD GO TO ENDATA            X
               SYNAD=ABTERM,          I/O ERROR GO TO ABTERM         X
               MACRF=(R),             BSAM I/O USES READ/WRITE MACROS X
               DSORG=PS               DATA SET ORGANIZATION
```

```
*        PHYSICAL SEQUENTIAL.
         SPACE 5
PRNTDCB  DCB   RECFM=FA,               RECORD FORMAT FOR OUTPUT        X
               BLKSIZE=80,             PHYSICAL BLOCK SIZE             X
               LRECL=80,               LOGICAL RECORD SIZE             X
               DDNAME=FT06F001,        CONNECT TO OUTPUT DEVICE        X
               SYNAD=ABTERM,           I/O ERROR GO TO ABTERM          X
               DSORG=PS,               DATA SET ORGAN. IS PHYS. SEQU.  X
               MACRF=(W)               END PRINTER DCB
         SPACE 5
DISKDCB  DCB   RECFM=F,                RECORD FORMAT FOR DISK I/O      X
               BLKSIZE=80,             PHYSICAL BLOCK 80 BYTES         X
               LRECL=80,               LOGICAL RECORD LENGTH           X
               DDNAME=FT09F001,        LOGICALLY CONNECT TO DEVICE     X
               EODAD=END,              ON EOF GO TO END                X
               SYNAD=ABTERM,           PERMANENT I/O ERROR GO TO ABTERMX
               DSORG=PS,               DATA ORGAN. IS PHYS. SEQU.      X
               MACRF=(RP,W)            READ WRITE COMBINATION
         SPACE 5
*        END OF DCBS FOR I/O OPERATION
         END   BSAMIO
/*
//DATA.FT09F001 DD UNIT=SYSDA,SPACE=(CYL,1),DSN=&&TEMP,                X
//             DISP=(NEW,DELETE)
/*LOG
//DATA.XSNAPOUT DD SYSOUT=A
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.INPUT DD *
I AM THE FIRST CARD TO BE READ
I AM THE SECOND CARD TO BE READ
I AM THE THIRD CARD TO BE READ
CARDS ARE ECHO PRINTED AND WRITEN TO DISK
THEN ARE READ FROM THE DISK AND PRINTED AGAIN
THIS IS THE LAST CARD
/*
```

COMPUTER SCIENCE 102 - ASSIGNMENT 1
DUE _____

This assignment covers simple input/output, binary arithmetic for fullword and halfword numbers, and basic data movement and testing codes for handling such numbers.

AI. BASIC PROGRAM

The basic program should do the following:

A. Read a card (XREAD), and print it out immediately (called an ECHO CHECK - standard practice). The card contains 5 numbers punched on it, which are to be scanned and converted (XDECI) to binary form, and placed in 5 consecutive fullwords in memory. Print the hexadecimal values of these 5 words (20 bytes), using XDUMP.

B. Perform the following computations in a straightforward way, storing each result in name given, using RX instructions where you can):

1. F = A + B + C
2. G = -A - B - C (LCR useful)
3. H = A * B * E
4. I = A / B (be careful, watch for negative #'s)
5. J = MOD(A,B) (i.e., remainder from# 4.)
6. K = ((A + E) * (B - C)) / D

C. Print all of the above values (F - K) in hexadecimal (XDUMP), then also print them in decimal, using XDECO and XPRNT (print their values an headings all on one line.

D. According to the sign of result H, print one of the 3 messages: H IS LESS THAN ZERO, H IS GREATER THAN ZERO, H IS ZERO.

II. EXTENDED VERSION OF PREVIOUS PROGRAM

Modify the previous program (which only had to read 1 card), to read cards and follow the actions above for each card, until there are no more cards (END-OF-FILE). Keep a count of the number of cards read, and print out this total number before ending the program.

III. HALFWORD VERSION OF PROGRAM II.

Modify program II to use halfwords wherever possible (i.e., store A - K as halfwords, use AH instead of A, etc. Watch out for divides, since no DH instruction exists). How much storage is saved?

IV. REGISTER VERSION OF PROGRAM II.

Change program II by saving all values A - F in registers, then use RR instructions rather than RX instructions. Do XDECI commands directly into registers where the values are saved. A useful trick may be to NAME the registers symbolicly:

```
RA      EQU   3                  REGISTER WHERE VALUE A KEPT
        XDECI RA,CARD            CONVERT VALUE A INTO REG RA
```

This technique will make it clear which value you are using (note that any register reference can be symbolic to an EQU symbol).

V. WHAT TO HAND IN

By using the BATCH feature in ASSIST, you can run several programs in one run. Turn in one run, with each of the programs II, III, and IV shown in execution, with results and output as requested. The run will use control cards like:

```
// EXEC ASACG,PARM=BATCH
//SYSIN DD *
$JOB  ASSIST  PROGRAM VERSION II
..... program II
$ENTRY
..... test data
*** repeat above, starting at $JOB, for programs III and IV.
/*
```

The following test data should be used for each program:

A	B	C	D	E
5	2	-4	-2	2
-2	-1	10	1	-1
4096	1	1	-1	-1

Note that the columns they are punched in should not matter.

COMPUTER SCIENCE 102- ASSIGNMENT 2

This assignment uses the concept of indexing into an array of elements.

I. BASIC PROGRAM

A. Read a card (and echo print) containing a maximum of 20 numbers. Convert the numbers to hex(XDECI) and store them in successive fullwords in memory. Use a loop to eliminate redundant coding. Then, for each card, find the maximum value and the minimum value, printing out these numbers with appropriate labels.

B. Form of data

1. Each card contains a maximum of 21 numbers, where the first number =the number of numbers on the card. You will need the first number for a counter in the loop in part A.

2. There are an unspecified # of data cards. i.e., make your program general to accept any # of data cards.

II. DATA FOR YOUR PROGRAM

```
3       56  76  -76
7  11  123  432  -123  748  -9087  -0
6  33 33  45  10  6  90
4  145  1024   6698  -1024  345
$
```

COMPUTER SCIENCE 102 FINAL PROJECT
DUE _____
Pages 1-8

I. INTRODUCTION

One of the concerns of a programmer is to design computer systems, languages, and translators for the languages. The monitor is that part of the system which controls the running of the translators and interpreters.It examines the JCL to determine the beginning and end of a job.

The function of the translator, or assembler, is to accept as input a source language(e.g. IBM 360 Assembler Language) and generate equivalent code in some target language (e.g. machine language). This code is called object code. In addition to translating, the assembler should have the capability of detecting syntactic errors in the source listing. The interpreter however executes the intructions of the target language.

II. LANGUAGE FORMAT

A. Control Cards

1. Control cards will be used in TRIVIAL. These cards direct the monitor about how to process a job. On the IBM s/360 these cards are called Job Control Language cards(JCL). Also included with the JCL cards will be control cards used by the assembler.

2. These cards indicate the start of a program, end of assembly and start and end of data.

B. Assembly Language

There are two types of instructions:

1. Machine instructions-can be executed by the computer hardware The instruction consists of a mnemonic opcode followed by one or more operands. A label field is optional. All are in free format, i.e. one or more blanks in between.

```
ex- LABEL   OPCODE    OPERAND(S)    COMMENT
                        separated by
                         commas
```

2. Assembler Instructions-or pseudo-instructions, or pseudo-ops. These are instructions to the assembler. Examples are reserving storage(DS), reserving storage with initial value(DC), setting the location counter (ORG and START instruction).

##

C. Machine Code (or Object Code)

1. Machine code is the output of the assembler. The format of the code varies from machine to machine, but in general consists of the following:

OPCODE- coded form of the operation to be performed.
REGISTER-part of an operand
ADDRESS-part of the operand. It can consist of

a. base and displacement 10(2)
b. an actual address 100
c. an indirect address
d. an indexed register form 0(11,12) reg 11 index
e. shift counter SLA 2,4
f. immediate operand CLI 2,x'F0'

III. IMPLEMENTATION

A. Monitor

1. The monitor examines the control cards to determine the start and end of a job.

2. The monitor usually examines switches to determine what to do. e.g. If the assembler sets the switch for valid assembly done. the monitor then calls for execution of the program.
3. The monitor skips to the next job if the current job can not be executed as shown by flags.(may skip data cards)

B. Assembler

Assemblers vary as to the number of passes it performs. The number is a function of the size of the machine, the speed(wanted) and the complexity if the source language. We will be interested in a 2-pass assembly, which is most common.

1. The first pass

a. In the first pass, the opcodes are checked for validity (e.g. an opcode may not exist or may not be implemented on a certain machine).

b. Symbols (statement labels,variable names) are entered in the symbol table.

c. Storage is reserved for constants. Note that some storage requested must be on a specified boundary as in the 360(fullword,..)
##

d. The location counter is incremented according to the length of the instruction or storage(allocated). Note instructions may vary in length for some computers (like the 360).

e. The source statements are saved so that a listing may be printed at a later pass. Information added to each source statement would be the opcode type, statement number, the location counter value, or anything else.

f. Assembler instructions are recognized and appropriate action is taken.

g. A copy of these source statements may reside in core if there is enough space, or may be put on disk or magnetic tape.

2. The second pass-first it must retreive the source statements fo further processing.

a. The operands are then processed according to their type & and type of op-code they are with.

b. Object code is then created for the instructions and data definitions; It then resides in core or is put on disk or tape.

c. A listing is printed. It contains the location counter, generated code, and source statment for each source statement. Error messages are also printed on the listing.

d. The errors to check for (assembly time errors)

1. multiply defined symbols
2. undefined symbols
3. invalid opcode
4. invalid values e.g. if a machine has 16 registers and program uses register 17
5. invalid labels-in Fortran, a label of 9 letters

3. During assembly a table (which will remain fixed) will contain all of the legal opcodes. Information included could be the name of the opcode (mnemonic) and type (machine or pseudo-op).

4. A symbol table is also formed. It contains the symbol, the location counter value, and maybe other flags. Each symbol is entere when it appears and when a reference is made in an operand the symbol is looked up and the value obtained.

##

C. Interpreter/Execution monitor

1. If the target language is executable of the 360, then merely branch to the program.

2. Otherwise, initialize the program counter (location counter) to BEGIN and execute the object code using pseudo registers and pseudo storage. You will also need a Program Status Word(PSW) containing necessary flags.Execution is as follows:

1. Fetch the next instruction & increment location counter according to the length of the instruction
2. Decode the instruction & evaluates address(es).
3. Execute the instruction
4. Go to 1.

3. During execution there is to be error checking for:

a. reading beyond end of file

b. executing too many instructions (a limit will be set on the job card)

c. invalid op-code (by branching into data area)

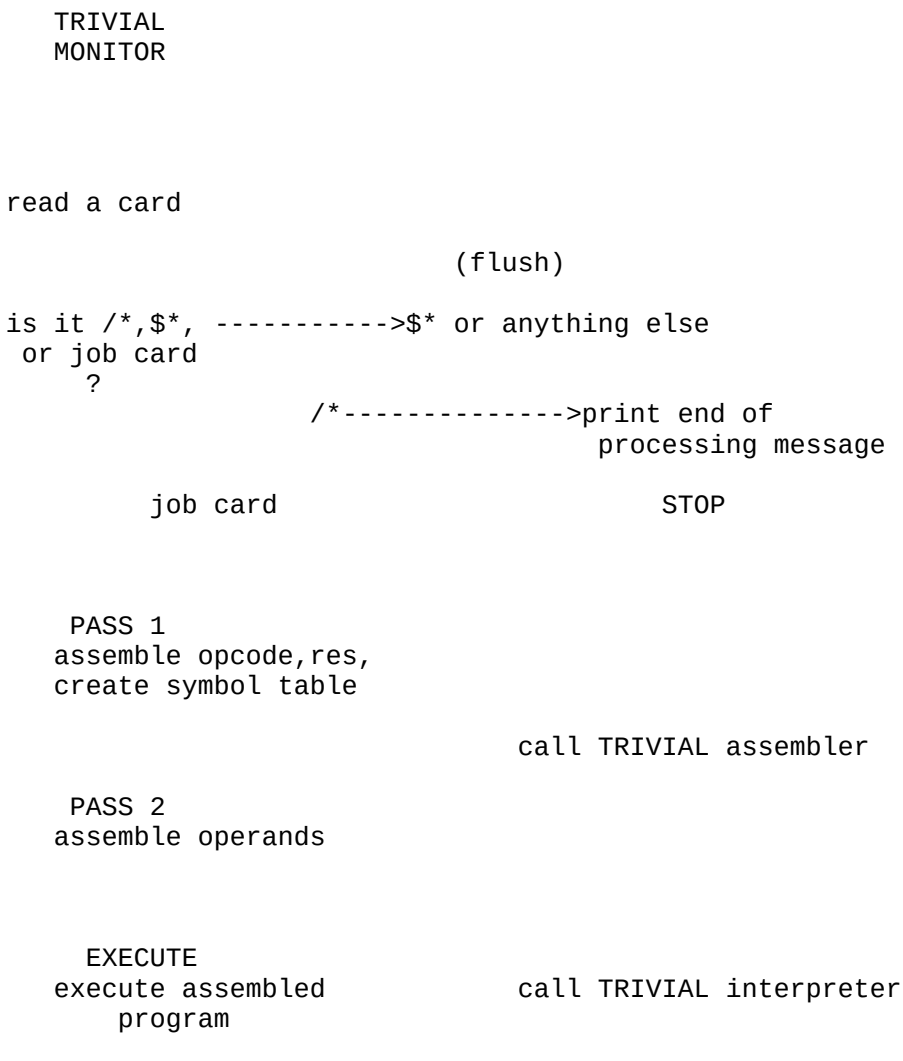
d. address out of range of program

4. A dump should be provided at the end of each job. Information included will be the value of the program counter, the contents of th registers, and the storage.

##

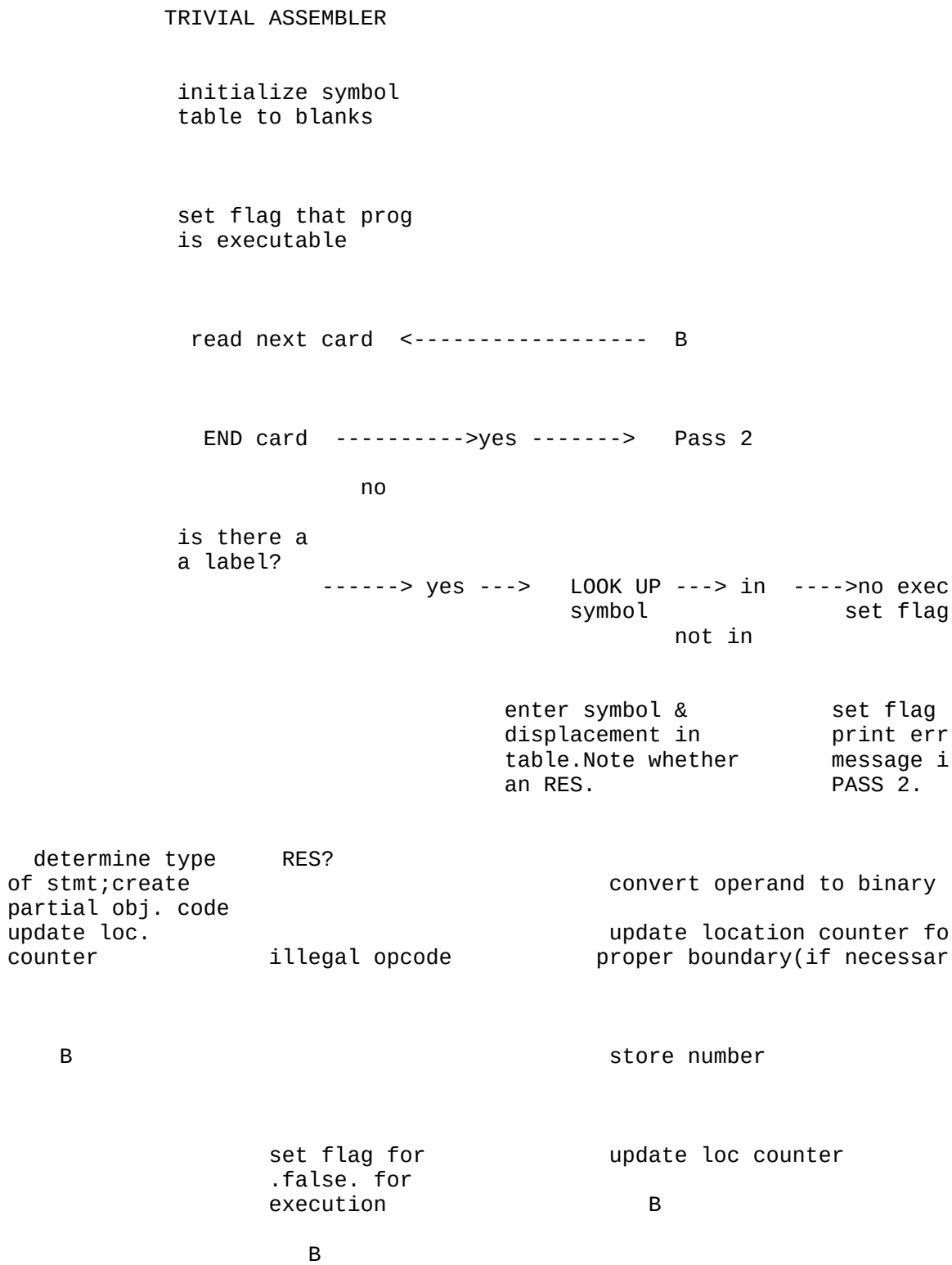
IV. FLOWCHARTS

A. General Flow of Monitor



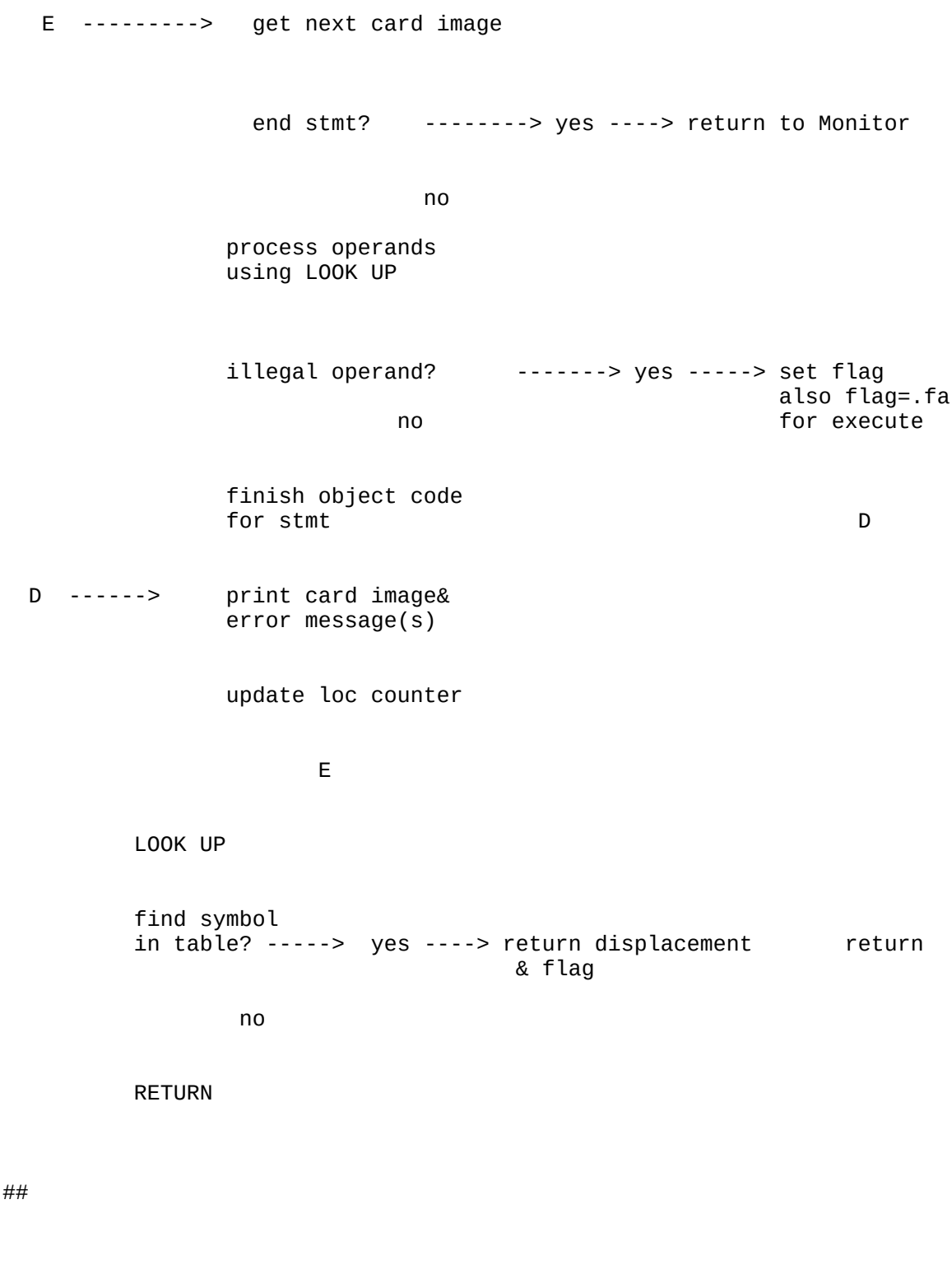
##

B. TRIVIAL assembler-Initialization and Pass 1



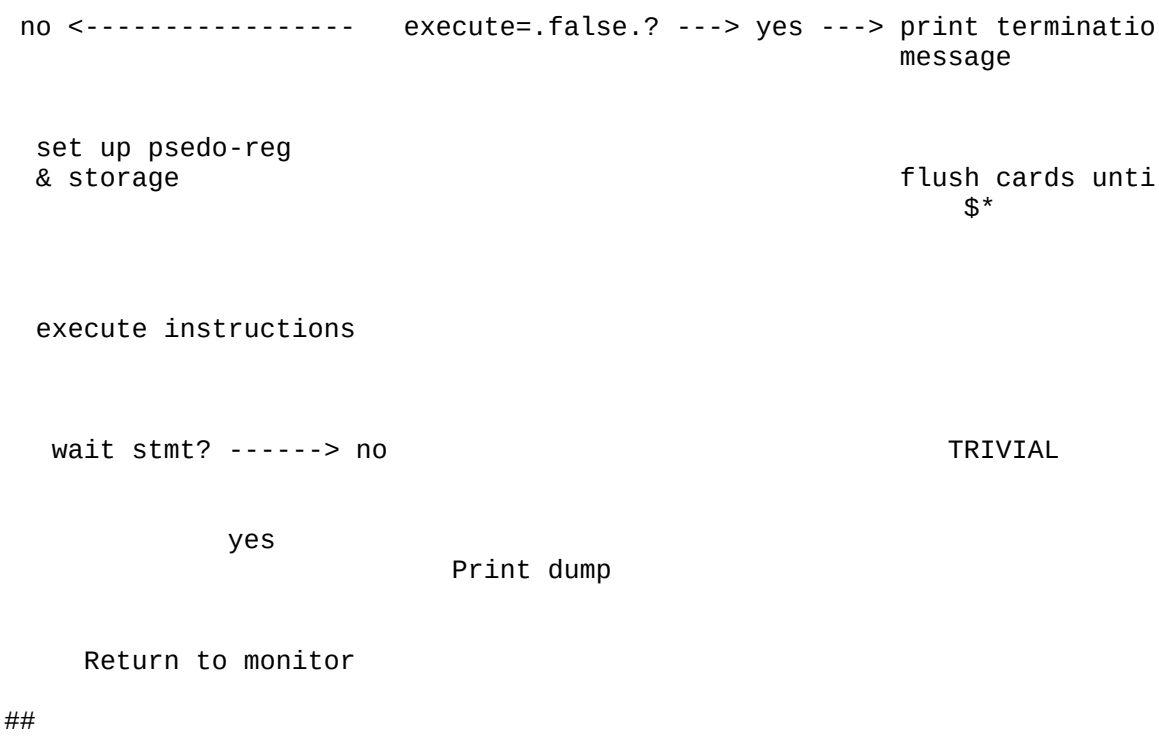
TRIVIAL Assembler Pass 2

PASS 2



##

EXEC


no <----------
execute=.false.?
---> yes --->
print terminatio
message
set up psedo-reg
& storage
flush cards unti
$*
execute instructions
wait stmt?
------> no
TRIVIAL
yes
Print dump
Return to monitor

##

V. SPECIFIC LANGUAGE DETAILS FOR THE TRIVIAL SYSTEM

This section describes the exact details for the various parts of the TRIVIAL system. The monitor (main program) processes various control cards, which direct further processing. The TRIVIAL assembly language is converted by the assembler into machine code for a computer system called the SIGMA 4.5 . The interpreter portion then simulates the operation of the SIGMA 4.5, using S/360 instructions.

A. CONTROL CARDS

1. JOB CARD - SHOWS BEGINNING OF TRIVIAL PROGRAM

This card has the following format (starting in column 1):

```
$$JOB  jobname  number
```

$$JOB	identifies this as a JOB card
jobname	is a sequence of up to 20 nonblank characters which identify this JOB.
number	gives a maximum limit on the number of instructions which can be executed by the user program on the simulated SIGMA 4.5 .

There may be any number of blanks before and after jobname, i.e., these cards are FREE FORMAT.

2. $* CARD - INDICATES END OF A JOB

This card is of following format, beginning in column 1:

```
$*   rest of card is ignored.
```

3. OVERALL DECK SETUP

The input to TRIVIAL is made up of 1 or more JOBS, each as follows:

```
$$JOB     jobname#1   # instructions limit for job 1
..... TRIVIAL assembly program
     END card  (showing end of assembly program)
..... 0 or more data cards to be read by program
$*
```

Note that any user program cannot be allowed to read beyond a $* or $$JOB card into the next user's program. Test decks will be supplied to the students.

B. TRIVIAL ASSEMBLY LANGUAGE

The following describes the format of a TRIVIAL assembly program, giving in detail the forms of mnemonic opcodes, operands, and labels. NOTE: the reader should probably first consult Part C, since it gives the machine code formats used by the SIGMA 4.5 .

1. MACHINE INSTRUCTIONS

There are essentially two formats for machine instructions: RX format, which is similiar to S/360 RX format, and RI format, which operates on a register and an immediate operand field.

a. RI INSTRUCTIONS

These instructions follow the format below:

```
label     opcode    register,immediate

label     is an optional statement label
opcode    is one of the immediate operand instructions (like AI)
register  is a decimal number from 0 - 15 (no leading zeroes.
immediate is a signed or unsigned decimal number from  -1048576
          to +1048575  to be used as an immediate value.
```

b. RX INSTRUCTIONS

This format is basically like S/360 RX format, with some restrictions plus an addition for indirect addressing.

```
label     opcode    register,address

label     is optional statment label
opcode    is one of the RX opcodes, like AW.
register  is decimal number from 0 - 15 (like register in RI above).
address   represents an address in memory, and may have any of the forms
          given below.  Note that  symbol represents any legal label,
          n represents any unsigned number  representable in 17 bits,
          and index represents any decimal number from 0 to 7, giving
          an index register.  The formats are then:

          symbol         implying direct addressing, index = 0
          symbol+n       "    "    "    "    "    "    "
          symbol-n       "    "    "    "    "    "    "
          n              "    "    "    "    "    "    "
          any of the above followed by (index), i.e,  symbol+n(index)
                         implying index register of given value.
          any of the above preceded by *, indicating INDIRECT ADDRESS.

          Any place  symbol appears above, the character  $  can appear,
          which refers to the location counter value  (like S/360 *).
```

EXAMPLES: AI 0,1000 ; AW AREA+12(7) ; BCS 8,LOOP1 ;
BDR 2,LOOP ; LW 1,*VECTOR+5(2) , BCS 15,$+2

2. ASSEMBLER INSTRUCTIONS (PSEUDO OPS)

a. ORG - sets location counter to value specified in operand, which may be any of the forms symbol, symbol+n, symbol-n, n, $, $+n, $-n allowed by first section of RX operand format.

EXAMPLES: ORG LABEL+2

NOTE: may not have a LABEL field specified, and any symbols used must be previously defined.

b. END - signifies end of TRIVIAL program, may have no label.

c. RES (REServe) - similiar to S/360 DC and DS, reserves storage and may fill in constants. It may have any r of the following forms:

```
label     RES  number
```

The statement above reserves number words of storage, number being a positive integer from 0 to 1023 .

```
label     RES  W'number'
```

This statement causes the decimal number (signed/unsigned) to be converted to binary and assembled at the given location. (corresponds to S/360 DC F'number').

```
label     RES  numberW'number'
```

The first number gives a duplication factor from 1 - 1023, and causes that many copies of the constant to be assembled (like S/360 DC numberF'number').

3. MISCELLANEOUS

The label field may contain any label usable on S/360, except that $. @, # are excluded , i.e., a label begins with a letter, and then containues with 0-7 letters/digits.

NOTE THAT ALL STATEMENTS ARE FREE FORMAT: they may have 1 or more blanks between fields, but no blanks inside each field.

C. DESCRIPTION OF THE SIGMA 4.5 COMPUTER SYSTEM

1. MEMORY, REGISTERS, AND PROGRAM STATUS BITS

The MEMORY of the SIGMA 4.5 (model W72), is composed of 1024 WORDS, (addressed at locations 0, 1, 2, ... 1023). Each is 32 BITS long.

The SIGMA 4.5 contains 16 REGISTERS, each 32 bits long, numbered from 0 to 15. Of These, all may be used for holding operands and doing arithmetic, while registers 1-7 only may be used as INDEX REGISTERS in address calculations.

A PROGRAM STATUS WORD is used to keep the current status of the SIGMA 4.5 computer. Of the 32 bits in this word, the following uses are made of:

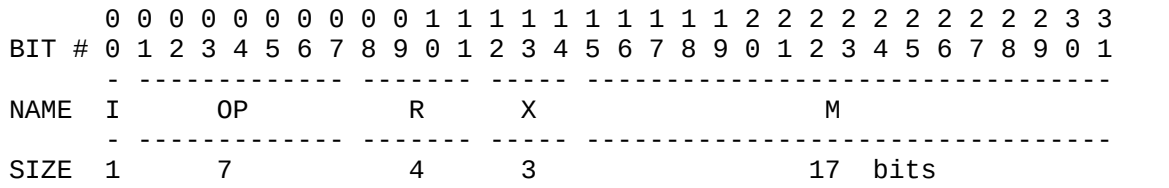
BITS	NAME	USAGE
0-1	CONDITION CODE	exactly like S/360 condition code.
2-14	UNUSED	
15-31	PROGRAM COUNTER	17-bit address of the next instruction in memory to be executed.

NOTE: all numbers are encoded as 32-bit signed numbers, using two's complement notation.

2. INSTRUCTION FORMATS, ENCODING, ACTIONS

The SIGMA 4.5 has two basic instruction formats:
RX (Register to Storage, Indexed, almost exactly like S/360 RX).
RI (Register Immediate - uses Immediate operand in instruction).

The layout of the RX-format instruction is as follows:



The various fields above are used as follows:

I INDIRECT ADDRESS: 0 => direct addressing, 1 => indirect (see below)

OP OPCODE: number from 0 to 127 noting which operation to be done.

R REGISTER: number from 0 - 15 specifying a register (usually), or sometimes a Branch Mask or other value.

X INDEX REGISTER: number from 0-7 specifying an index register, with 0 specifying NO indexing.

M MEMORY ADDRESS: number from 0 - 128K, specifying the actual location in memory of an operand.

EFFECTIVE ADDRESS CALCULATION: RX INSTRUCTIONS

In general, the SIGMA 4.5 calculates an EFFECTIVE ADDRESS (EA) using the I, X, and M fields in the procedure below, then operates in some way on the word at the EA and the word in register R.

The EA is determined by the following steps:

First, the M field is taken either as the address of a word in memory, or the address of an address of a word in memory, depending on the I bit, as follows:

If I=0, then EA1 = M (Direct addressing)

If I=1, then EA1 = low-order 17 bits of word at location M. (This is called INDIRECT ADDRESSING).

Now, EA1 is either taken as the EA, or is modified by adding to it the contents of an index register X, as follows:

If X=0, EA = EA1 (no indexing).

If X ^= 0, EA = EA1 + contents of register X. NOTE: EA1 is a positive 17 bit number, while value in register X may be either positive or negative. Thus, if M=100, X=1, and register 1 contains -10, the location referenced should be 90 = 100 + -10 .

NOTES AND GENERAL INFORMATION: The SIGMA 4.5 stops at doing only 1 indirect address step, and thus uses ONE-LEVEL INDIRECT ADDRESSING. Some other machines would extend this by testing the I bit in a word, and if on, perform another indirect addressing step, until a word was found wit1out I=1, in which case that word would have the address of the word finally used as an operand. This type is called MULTILEVEL or CASCADED INDIRECT ADDRESSING.

Also, since SIGMA 4.5 adds the index register X AFTER any indirect addressing is done, it is called POST-INDEXING. Some machines perform indexing before indirect addressing, and are thus called PRE-INDEXING systems.

The RI (Register Immediate format) instructions are basically similiar to the RX, except:

The I bit is ignored.

The X and M fields together form a two's-complement, signed, 20-bit number, which is extended to 32 bits, then used as the operand in combination with the register R. This allows numbers from -1048576 to +1048575 to be specified in the instruction.

3. OPCODE TABLE

The following table lists the mnemonic opcode for each machine instruction, the machine code for each operation (hexadecimal, 7-bits), the instruction format (RX, RI), whether or not the instruction sets the condition code, and finally, exactly what the instruction does. If an instruction corresponds to an S/360 instruction, it sets the CC in the same way, unless otherwise specified.

NAME	CODE	TYPE	CC SET	ACTION
AI	20	RI	YES	ADD IMMEDIATE : add Immediate field to reg R.
LI	22	LI	NO	LOAD IMMEDIATE : extend immedaite field to 32 bits, load that value into register R.
MI	23	RI	YES	MULTIPLY IMMEDIATE : extend immedaite field, multiply with register R, truncate to botain low-order 32 bits in R, setting CC for result.
WAIT	2E	RX	NO	WAIT: execution terminates, EA placed into specified register as a completion code.
AW	30	RX	YES	ADD WORD: like S/360 A.
CW	31	RX	YES	COMPARE WORD : like S/360 C.
LW	32	RX	NO	LOAD WORD : like S/360 L.
MTW	33	RX	YES	MODIFY AND TEST WORD: the R field does not refer to a register; instead, it is taken as a signed, 4-bit number (-8 to +7), extended to 32 bits, and added to word at EA, setting CC
STW	35	RX	NO	STORE WORD : like S/360 ST.
MW	37	RX	YES	MULTIPLY WORD: like MI, except operand taken from EA location instead of immediate field.
SW	38	RX	YES	SUBTRACT WORD : like S/360 S.
BDR	64	RX	YES	BRANCH ON DECREMENTING REGISTER: Modify reg R by subtracting 1. Set CC according to result and branch to EA location if result > 0.
AWM	66	RX	YES	ADD WORD TO MEMORY : add contents of register R to word at location EA.
BCS	69	RX	NO	BRANCH CONDITIONS SET : uses R as a Mask field to branch or not : same as S/360 BC
RD	6C	RX	YES	READ DIRECT : read a card (set CC = 0), if no more remain, CC = 1. A single signed number anywhere on the card is converted to binary and placed in the register specified by R. EA is totally ignored, and may have any value.
WD	6D	RX	NO	WRITE DIRECT : the number in register R is converted to decimal and printed.

```
$$JOB  LEGALOPS1  0
*********************************************************************
********** ATTENTION:  USE LARGE PARAMETER IF YOU ARE GETTING AS999,
********** SIMILIAR MESSAGE ABOUT EXCEEDING SPACE - SEE ASSIST MANUAL.
********* NOTE EXTENSION/EXTENSION/EXTENSION:  FINAL PROJECTS DUE AT  *
********* 11AM  WEDNESDAY, NOT MONDAY.                               *
*********************************************************************
*  THIS PROGRAM CONTAINS NO ERRORS. IT IS A TEWST FOR THE
*  RECOGNITION OF ALL LEGAL OPCODES. THE PROGRAM WILL NOT
*  BE EXECUTED
     AI       1,-2
      AW   1,3(2)
    LI   1,2
```

```
     CW   1,2
LABEL     MI   1,2
    LW   1,3(2)
     MTW    1,3(2)
      STW     1,2
       ORG    LABEL
       MW   1,2
      SW    1,2
      BDR    1,3(2)
       AWM     1,3(2)
       BCS      1,3(2)
      RD     1,3(2)
       WD      1,3(2)
       WAIT    2,2
WORD    RES    2
      RES      W'2'
     RES     2W'2'
     END
$*
$$JOB   LEGALOPS2  0
*  THIS PROGRAM TEST THE RECOGNITION OF LEGAL OPERANDS
         MI        1,2
         MI      1,-2
         MI      1,+2
        AW    1,2
        AW    1,*2
        AW    1,2(3)
        AW    1,*2(3)
        AW    1,$
        AW    1,*$
        AW    1,$(3)
        AW    1,*$(3)
        AW    1,$+2
        AW    1,*$+2
        AW    1,*$+2(3)
        AW    1,$+2(3)
        AW    1,SYMBOL
        AW    1,*SYMBOL
        AW    1,SYMBOL+7
        AW    1,*SYMBOL+7
        AW    1,SYMBOL+1(2)
        AW    1,*SYMBOL+1(2)
        AW    1,*SYMBOL-1(2)
SYMBOL      RES     5
        END
$*
$$JOB     PASS1ERR   0
*  THIS PROGRAM CONTAINS ERRORS IN LABELS AND ASSEBLER OPS
       LI    5,0
      LI     3,0
     SWR    3,0(2)    ILLEGAL OPCODE
      AI      2,4
LOOP     MI      2,CONST
7LAB    SW    2,CONST+1     ILLEGAL LABEL
    STW      2,SAVE     UNDEFIND SYMBOL
ORGLINE     ORG   LOOP    NO LABEL ALLOWED
     RD    3,AREA1      UNDEFINED LABEL
     STW    3,SUM
OUT    WD    3,AREA
     AWM    3,SUM
BRANCH    BDR    2,LOOP
```

```
      READ      2,AREA    ILLEGAL OPCODE
BADLABEL    LW    5,SUM     LABEL TOO LONG
OUT     WD    5,SUM     MULTIPLY DEFINED LABELS
*  NOTE- BLANK CARD FOLLOWS

     MTW   0,OUT
      ORG     $-100    ORG TO NEGATIVE ADDRESS
SUM    RES   1
     RES    W'-9999'
*   THE FOLLOWING RES STMTS ARE IN ERROR
AREA   RES   1100    TOO LARGE
     RES    0W'1'
     RES    10W
   RES    W'10 '
     RES   W'
      RES    W'12345A'
     RES      -10W
THISLABELISTOOLONG    ANDSOITTHISOPCODE
 MTW                                  0,0
ENDLABEL     END       ILLEGAL LABEL FIELD
$*
$$JOB    PASS2ERR    0
*  THIS PROGRAM CONTAINS ERRORS IN THE OPERANDS
*   RI FORMATS
*       OMMITTED OPERAND FOLLOWS
     AI
     LI    16,0     REGISTER TOO LARGE
    LI    15      OPERAND OMITTED
     LI    15,    OPERAND OMMITTED
      LI    ,0     OMITTED REGISTER
      LI     15,524288    TOO BIG
       LI      15,-524289
       LI      15,524287     LEGAL
     LI     15,-524288      LEGAL
      LI      15,0,     INVALID DISPLACEMENT
*
*    RX FORMAT
     AW    16,0     REGISTER TOO LARGE
    AW    15
      AW      15,       OMITTED OPERAND
     AW     15,*      OMITTED OPERAND
        AW      15,*(1)     OMITTED SYMBOL
X       AW      15,5+X      WRONG ORDER
      AW      15,5+1
     AW     15,X+      ILLEGAL OPERAND
          AW      15,X+(3)
       AW      15,*X-(3)
       AW      15,X-(3,
     AW     ,X-X        OMITTED REGISTER
      AW    15,0(8)
     AW     15,X+5(8)      INDEX ERG TOO LARGE
      AW     15,$(8)     INDEX REG TOO LARGE
      AW      15,$-2,
      AW      15,0(7)+
       AW      15,$-20000      BAD ADDRESS
           AW     -1,0     ILLEGAL REG SPECIFICATION
     AW    0,-1
       AW    0,1(-1)
        AW        0,131071     LEGAL
       AW      0,131072    TOO BIG
           END
```

```
$*
$$JOB     READBEYONDEND  100
LOOP      RD   0,0       READ A CARD
          WD   0,0       WRITE IT BACK OUT
          BCS  15,LOOP   LOOP FOREVER, OR UNITL CARDS RUN OUT
          END
 1        THIS IS THE ONLY DATA CARD
$*
$$JOB     PROGRAMLOOPS   10
          BCS  15,$      SAME AS B *
          END
$*
$$JOB     BRANCHTOBADOPCODE  5
          BCS  15,$+1    BRANCH TO NEXT STMT
          RES  W'0'      NOT A LEGAL OPCODE
          END
$*
$$JOB     ADDRESSOUTSIDE 5
          LI   1,100000  BIG NUMBER
          SW   0,$(1)    TOO BIG - ADDRESSING ERROR
          END
$*
$$JOB     ADDRESSINGTEST   20
          LI   1,1       FOR INDEXING
          LI   2,-1      FOR INDEXING
          LW   8,X(1)    LOAD AN 8 INTO R8
          LW   7,X(2)    LOAD A 7 INTO R7
          LW   6,*X      LOAD A 6 INTO R6
          LW   5,*X(1)   LOAD A 5 INTO R5
          LI   3,512     ANOTHER INDEX VALUE
          AWM  1,*Y(2)   BOMB OUT OF RANGE
          WAIT 0,0       NEVER REACH HERE
*
          RES  W'7'
X         WAIT 0,X6      FOR INDIRECT ADDRESSING
          RES  W'8'
X6        RES  W'6'
          RES  W'5'
Y         RES  W'10000000'  BIG
          END
$*
$$JOB  COMPUTEXCUBED   100
          RD   0,0       GET NUBMER IN
          STW  0,X       SAVE THE VALUE
LOOP      MW   0,X       MULITPLY 1 TIME
          MTW  15,CNT    DECREMENT CNT TO CNT-1
          BCS  7,LOOP    BRANCH IF NOT ZERO
          WD   0,0       WRITE RESULT OUT
X         RES  1
CNT       RES  W'2'
          END
$*
```

COMPUTER SCIENCE 102 - RUN ASSIGNMENT

1. Punch up the following program and run:

```
//          YOUR JOB CARD
// EXEC ASACG
//SYSIN DD *
MAIN     CSECT
*  THIS PROGRAM ILLUSTRATES XDUMP AND PROGRAM INTERUPTION
         BALR  12,0                THESE TWO STMTS ARE FOR
         USING *,12                COMMON LINKAGE CONVENTIONS
         LA    3,CARD              PTR TO CARD IMAGE READ IN
         XREAD CARD,80             READ DATA CARD
         XPRNT CARD,80             ECHO PRINT
         XDECI 4,0(3)              CONVERT DECIMAL TO INTERNAL HEX
         XDECI 5,0(1)              CONVERT NEXT # ON CARD
*  THE NEXT STMT PRINTS CONTENTS OF USERS REGISTERS.
*  NOTE REG 4,5
         XDUMP
         B     4000                ABEND-BRANCH OUT OF PROGRAM
CARD     DS    80C
         END
/*
//DATA.INPUT DD *
    100    -1024
/*
```

2. This next program is a batch run of 5 jobs, each terminating abnormally. The program is stored on RJE file. Punch up the follwoing cards EXACTLY to run the program

```
//    YOUR JOB CARD
// EXEC ASACG,PARM=BATCH
//SYSIN DD *
/*INCLUDE RAB01.BATCH
/*
```

3. To merely get a listing of the prog in 2., use the following cards:

```
//       YOUR JOB CARD
/*INCLUDE RAB01.PRINT
/*INCLUDE RAB01.BATCH
/*
```

A GUIDE TO S/360 MNEMONIC OPERATION CODES

I. INTRODUCTION

The beginning programmer facing the variety of operations available on a modern large computer is often overwhelmed by the large number of operations and complexity thereof. In some cases, a few hints can be helpful in learning and remembering the names, purposes, and usage of the various operations. In particular, certain properties of S/360 menemonics can help the learner a great deal. Some of them are:

A. REGULAR SCHEME FOR NAMING OPCODES

In general, a fairly coherent and regular method has been used in naming operations. In some cases, it is possible to determine the bit pattern and operation of a mnemonic just from looking at it. Related operations usually have related mnemonics.

B. COMMONLY USED MNEMONICS

The designers apparently went to some effort to make the most often used mnemonics the shortest and easiest to remember. Most of these have 1 or 2 letter mnemonics.

II. NAMING OF MNEMONIC OPCODES

A. VERB (MODIFIER) (DATA TYPE) (MACHINE FORMAT)

The mnemonics generally follow the format given above, with the VERB always present, while the others may be omitted. The general meanings of the fields are given below.

1. VERB: specifies a general type of action performed, such as addition, subtraction, comparison, data movement.

2. MODIFIER: specifies a modification of the general action given by the verb, such as logical addition (rather than algebraic), moving multiple registers rather than single ones, and performing different actions while loading one register into another.

3. DATA TYPE: specifies the type of data being operated on, and is usually the same letter as that used to define a constant of the given type, such as H (halfword), P (packed decimal), etc.

4. MACHINE FORMAT: gives the type of machine instruction being used. This is most typically done by adding R or I to an RX mnemonic to obtain a similiar RR or SI instruction.

In general, the RX instructions, which are the heaviest used, have the shortest mnemonics, and most of the other instructions can be built from them by adding more letters.

B. EXAMPLES OF COMMONLY USED MNEMONIC ELEMENTS

The following sections explain the common mnemonic elements.

1. VERBS

VERB	MEANING, COMMENTS
A	Add two numbers (which may be binary, decimal, or floating)
B	Branch to another instruction (like GOTO)
C	Compare two fields (numbers or character strings)
CV	ConVert a number from one base to another
D	Divide one number by another
L	Load a quantity into a register from another or from storage
M	Multiply one number by another
MV	MoVe information from one area in storage to another.
N	aNd information together (logical AND)
O	Or information together (logical OR)
S	Subtract one number from another
ST	STore a register (or part of one) into storage
X	eXclusive or information together (logical exclusive OR)

For example, note that a given VERB may begin many instructions, which immediately shows they are related to each other. For example, the following are all comparison operations: C, CD, CE, CH, CL, CP, CR, CDR, CER, CLC, CLR.

2. MODIFIERS

The following lists verbs and their common modifiers.

VERBS	MODIFIERS	MEANING, EXAMPLES
A,C,S	L	Logical addition, comparison, or subtract is used rather than algebraic. EX: AL, CL, CLC, SLR.
B	AL	And Link - form of branch for doing linkage to subroutine so it can return. EX: BAL, BALR
	C	Condition - branch or not depending on a previously set condtion (IF(--) GOTO --). EX: BC, BCR.
	CT	Count - branch form used to decrement a register and branch if value not zero (DO LOOP). EX: BCT, BCTR.
	X	indeX - branch form for incrementing and testing index quantities. (DO LOOP). EX: BXH, BXLE.
L	C	Complement - used to set a register to complement itself or other (Y = -ABS(X)). EX: LNR, LNDR.
	P	Positive - set register to positive value from self or other (Y = ABS(X)). EX: LPR, LPER.
	T	Test - set register to value from self or other,
L,ST	M	Multiple - several registers are loaded or stored in one operation. EX: LM, STM

3. DATA TYPES

As noted previously, a data type character is usually the same as that used in a DC or DS statement to obtain a given type of data. If a type character is omitted, it usually implies that the instruction operates on 32-bit, fullword, binary quantities (such as A, C, S, etc).

DATA TYPE	MEANING, COMMENTS
C	Character - usually a contiguous string of bytes in memory, treated as printable characters or a string of bits. (FORTRAN LOGICAL*1). EX: MVC, CLC, OC, IC, STC. USUALLY IMPLIES SS INSTRUCTION FORMAT (all except IC, STC).
D	Double precision floating point (Doubleword, 64 bit) (FORTRAN REAL*8). EX: AD, SD, LTDR, LD. IMPLIES RR OR RX INSTRUCTION FORMAT.
E	Exponent - single precision floating point (fullword, 32 bit, FORTRAN REAL*4). EX: AE, LER, ME. IMPLIES RR OR RX INSTRUCTION FORMAT.
H	Halfword - 16 bit binary number (FORTRAN INTEGER*2) EX: AH, MH, STH, CH. IMPLIES RX FORMAT.
P	Packed decimal format (2 decimal digits per byte). EX: AP, SP, CP. IMPLIES SS INSTRUCTION FORMAT OF TWO-LENGTH TYPE.

4. MACHINE FORMATS

Several characters are used to denote the specific type of operand format being used (note that the data types can also imply specific formats. If they imply one of several, the last character distinguishes among them).

FORMAT	MEANING, EXAMPLES
I	Immediate - IMPLIES SI FORMAT. EX: MVI, CLI, OI.
R	Register - IMPLIES RR FORMAT. EX: AR, BCR, DDR.

III. EXAMPLE OF FAMILY OF RELATED OPCODES

This section lists all the members of the 'Compare' family of menemonics, showing their relationships adn the elements present in each name. The letters V M D F stand for Verb, Modifier, Data type, and machine Format.

OP-CODE	V	M	D	F	TYPE	COMMENTS
C	C				RX	fullword algebraic compare, the basic one.
CL	C	L			RX	fullword logical comparison (logical modifier)
CD	C		D		RX	compare double precision floating numbers
CE	C		E		RX	compare single precision floating numbers
CH	C		H		RX	compare a register algebraicly with halfword from storage (with sign extension)
CP	C		P		SS	compare two packed decimal numbers
CR	C			R	RR	compare two fullword values algebriacly, gotten from C by adding R.
CLC	C	L	C		SS	compare logically character strings
CLI	C	L		I	SI	compare logical immediate (a byte in memory with the one inside the instruction)
CDR	C	D		R	RR	compare double precision (in registers)
CER	C	E		R	RR	compare single precision (in registers)

The Sytem/370 computers have some additional opocdes:

CLM	C	L		M	RS	compare logical masked (from register to mem)
CLCL	C	L	C	L	RR	compare logical character strings long (up to 16 million bytes in one compare)

Consider the problem of writing a FORTRAN program which would simulate the operation of the instructions above (i.e., maintain variables representing PSW, Memory, GP Registers, etc, and go through the Fetch-Instruction, Decode, Fetch-Operands, Execute cycle). The arrangement of the opcodes would make it easy to share code, i.e., it would not be necessary to code each instruction separately. As an example, consider the following related instructions:

MNEMONIC	HEX CODE	BINARY CODE	SAMPLE INSTRUCTION/ASSEMBLED		
CR	19	0001 1001	CR	0,1	1901
CH	49	1000 1001	CH	0,2(3,4)	49034002
C	59	1001 1001	C	0,4(5,6)	59056004

Examine the bit patterns above. The first teo bits give the Machine Format (00-RR, 10-RX), the third and fourth give a Data Type (01-Fullword,00-Halfword in this case). The fifth-eighth bits give the Verb (1001 - algebraic Compare). In essence, there is only 1 Compare, which is branched to after the operands are obtained.

COMPUTER SCIENCE 102 - TOPICS COVERED, HANDOUTS
WINTER TERM 1972 - MASHEY

The handouts given are described in file CS102HN.

#	DATE	TOPICS, HANDOUTS, READING ASSIGNMENTS
--	--/--/72	----------

1 01/07 introduction to course. prerequistites (101, 401, equiv)
listed text materials for course:
1) STRUBLE: ASSEMBLER LANGUAGE PROGRAMMING: IBM SYSTEM/360
2) IBM: SYSTEM/360 PRINCIPLES OF OPERATION (POP)
3) IBM: S/360 OS ASSEMBLER LANGUAGE
4) PSU: ASSIST INTRODUCTORY ASSEMBLER USER'S MANUAL
(25 cents, at 426 McAllister)
5) IBM: S/360 REFERENCE CARD (GREEN CARD- BRING TO CLASS)

introduction to information representation in computer.
memory, addressing, similiarity to FORTRAN vector wit1 index beginning at 0 rather than 1. elements of memory in S/360: byte, halfword, fullword, doubleword.
positional notation. number systems (binary, octal, decimal, hexadecimal). conversion between them, uses.
representations of binary numbers: Two's complement, One's complement, Sign-magnitude. advantages and disadvantages: (TC - 1 zero, but harder for people; OC - 2 zeroes, but easier to handle; SM - easiest to handle, but slower circuitry)

READING: STRUBLE CHAPTER 1. Look at ASSIST PART III.

2 01/10 more on information representation; introduction to machine structure.

meanings of bit patterns: 1,2,4-byte binary numbers; charcters packed decimal (good for people, but wastes space); floating point (sign, characterisitc, and fraction).

structure of a very simple machine: memory of 16-bit words; 1 register; 1 program coiunter. a few instructions, each with opcode and address. explanation of basic instruction cycle:
1) Fetch instruction from where program counter points.
2) Increment program counter.
3) Decode instruction into its parts.
4) Execute instruction.
5) Loop back to 1.

S/360 machine structure: memory (note abbrev. K), GP and floating point registers, PSW. refer to GREEN CARD.

Begin instruction types:
1) RR (names with -R, examples)
2) RX (give first explanation of base-displacement)
3) RS

READINGS: STRUBLE - CHAPTER 2; POP - pp. 7-15.
HANDOUTS: CS102M1 - page 01 (run some ASSIST programs for dumps)

3 01/12 finish operands formats and introduce assembly language.

4) SI instructions (examples: MVI, CLI)
5) SS instructions (examples: MVC, CLC)
machine language - easy for machine to execute, hard to write
assembly language converted by assembler to machine code.

format of assembly language: label opcode operand comments
machine instructions - actual operations to be executed
assembler instructions (pseudo ops) - give information to
the assembler (ex: CSECT, DS, DC)
some basic functions of the assembler:
1) location counter
2) convert mnemonic opcodes
a) machine ops - translate to codes, increm location cntr
b) assembler ops - take actions specified, increm loc cnt
3) operands - convert to internal binary, base-displacement
4) print out a listing
5) make program ready for execution and pass control to it

stepped through complete test program (XREAD, XPRNT, XDECI, XDUMP) and explained listing and contents of dump.

READINGS: STRUBLE: Chapter 3; ASSIST MANUAL: PARTS II and IV; ASSEMBLER LANGUAGE: pp. 1-18.
HANDOUTS: DOCUMENT (documentation techniques for assembler)

4 01/14 go over some dumps and errors; discuss operand fields.

go through various dumps, showing 0C1, 0C4, and 0C6 errors. cover STRUBLE cahpter 3, pp.50-56: symbols, self-defining terms, literals, location counter reference, absolute and relocatable terms, expressions.

READINGS: STRUBLE: Chapter 4 to page 78.
ASSIGNMENT: STRUBLE: Chapter 1: problems 5,6,7,8,9. Chapter 2: problems 2,3. Chapter 3: problems 1,2,3,4,6.
INFORMAL ASSN: modify dump program to use XDECO and DUMP storage; use program with START to check relocatable vs absolutes. modify one of batch programs to get 0C6 rather than 0C4.

5 01/17 introduction to arithmetic and data movement instructions introduce idea of instruction families and regularity of mnemonics. Go thru following instructions: LR, LPR, LCR, LNR, LTR, L, LH, LA. AR, ALR, A, AL, AH, SR, SLR, S, SL, SH. mention M and D, also briefly note existence of Condition Code and show how to test it, without worrying about encoding. 20-minute question answer and review: questions occurred on differences between literals and self-defining terms, and on use of symbolic register equates.

READINGS: STRUBLE: Chapter 5.
HANDOUTS: CS102AS1 (pages 01 - 02) first assignment - input, output of numbers, calculations in binary.
CS102M1 (pages 02 - 05) S/360 mnemonic construction.

6 01/19 quiz and finish up data movement and binary arithemetic.
Twenty-minute quiz (diagnostic mainly): base conversions (2, 8, 10, 16); negative numbers, base-index-displacement addrs, relocatable vs absolute.
Instructions: LM, STM, MVC, MVI. M, MR, D, DR, MH and hints on what to watch for.
Programming techniques: review input/output & conversions (XREAD, XPRNT, XDECI, XDECO); method for building messages and obtaining length for XPRNT via MSGL EQU *-MSG .

ASSIGNMENT: indexing and comparison assignment, CS102AS1 - 03, due 02/02/72.
HANDOUTS: CS102AS1 - 03 (labeled CS 102 AS2 also) - indexing.
READINGS: STRUBLE CHAPTER 5, start on STRUBLE CHAPTER 7.

7 01/21 condition code, branching instructions, loops.
condition code values and encoding. BCR, BC, Extended Mnemonics (recommended for use over BC #). BALR, BAL and subroutines, BCT, BCTR usage, including decrementing regs.
example of basic loop to sum array of numbers.
flowcharting and good design versus kludge programming.

READINGS: STRUBLE Chapters 7,8,5.

8 01/24 finish loop control, begin on USING/DROP, linkage
Explain BXH, BXLE instructions, give typical setups: forward BXLE loop, backwards BXH loop, BXH scan loop.
show need for USING command. give rules for computation of base displacements: minimum base displacement for those which are available, higher numberred register if several have same.
begin conventions: exaplin registers 15, 14 usage on entry.

HANDOUTS: LINKAGE OS/360 linkage conventions
READINGS: STRUBLE: Chapter 5, LINKAGE HANDOUT

9 01/26 savearea linkage ans ome review.
Describe 18-fullword save area. go through the standard code used at beginning and end of a routine, calling methods. Do not work on argument passing, just normal code.
Misc. instructions: IC, STC, start on Shifts.
Various review for problems.
Note general usage of registers: get students into good habits

READINGS: STRUBLE: Chapter 11.

10 01/28 logical/algebraic arithmetic, shifts
20-minute quiz on previous instructions.
differences between condition code setting, aroverflow in algebraic arithmetic and logical arithmetic. examples.
shift instructions and how they are used.
READINGS: STRUBLE: Chapter 11, begin on chapter 10.

11 01/31 bit manipulation and uses. review on branching
bit manipulation instructions: NR, XR, OR, N, X, O, NI, XI, OI, NC, XC, OC, plus TM. what they do, and how to use them.
EQU trick for SI instructions and how to use it.
review: prototypes on loop control, advantages/disadvantages.

READINGS: STRUBLE: Chapter 10, first 3 sections.

12 02/02 assembler housekeeping, misc areas.
go over all of DC, DS operand formats in detail, showing what can exist as duplication factor-type-length-constant, including multiple operands and constants, expressions as duplication factors and length modifiers. also cover TITLE, EJECT, SPACE

READINGS: STRUBLE: CHAPTER 6, pp 110-121, problems 7,9,10
ASM LANG: 3, 7-9, 10-18 (except variable symbols/sequence symbols, 19-21, 29-33. section 5: EQU, DC (all except Bit Length Modifier, Scale Modifier, Exponent Modifier. all types except E, D, L, P, Z,Y, S, Q, complex relocatability). DS, ORG, LTORG, END. SPACE, EJECT, TITLE
POP: pp 24-34 except CVB, CVD. Logical instructions except TR, TRT, ED, EDMK. Branching except EX.

13 02/04 give out final project, discuss assembler/interpreters
concepts of assemblers: 2 pass assemblers, how to set up opcode and symbol tables (indexed jump methods), output desired.
go over structure of SIGMA 4.5 computer and its interpreter, noting indirect addressing in particular.

HANDOUTS: CS102FP1 (01 -08) general assembler/interpreter descr
CS102FP2 (01 -06) specific material for final project

ASSIGN: Final project, due 13 March (described in CS102FPx)

14 02/07 decimal numbers and conversions
zoned/packed decimal to and from binary. PACK, UNPK, CVB, CVD equivalent codes using M, D loops for decimal-binary-decimal.
examples of various formats/conversions.

READINGS: STRUBLE: Chapter 5: 106-110, Chapter 218-228, 228-233.

15 02/09 misc review, misc instructions, program mask.
SPM instruction, use of program mask, review BXLE, BXH, etc.

16 02/11 MIDTERM
covered data representations, most standard instructions, hand assembly, etc.

17 02/14 on midterm and final project
review of midterm results and problem areas. final project: overall structure, useful modules and how to set them up: decimal scan and output conversions, symbol scan, symbol table manager, opcode lookup, hexadecimal output, etc.
review of BXLE loop control.
HANDOUT: CS102PX1 (01 - 03) programming exercises: hand assembly, interrupts.

18 02/16 more on assembly process, location counter control.
use of ORG to set up tables, timetable for gettting final project done, program design process and debugging.

19 02/18 quiz, TR, TRT
30-minute quiz: hand assembly, BXLE loop setup.
TR uses, setup, workings.
TRT uses, setup, examples.

READINGS: STRUBLE CH 15: pp 342-345, 350-352. prob 1,3,4.
ASSIGN: write TRT table for scanning over hex digits.

20 02/21 programming techniques, use of TR, TRT, conversions
use of global table pointer, examples on TR, TRT.
decimal input conversion, using two TRT's, EX, PACK, CVB.
hexadecimal output conversion, using UNPK, TR.

ASSIGN: write code to perform conversions, also to read in names, place in table, then search table for later names.
READINGS: STRUBLE CH 15: ED, EDMK start.

21 02/23 conversions - hexadecimal input, decimal output, ED
go through hexadecimal input, but not in detail (TRT, TRT, EX of MVC right-justified, TR, PACK 9 into 5, ignoring extra byte)
decimal output: CVD, UNPK, OI for plus number, with leading zeroes.
decimal output: begin on ED, EDMK, doing parts with basic workings of ED, and standard pattern for integer numbers.

CMPSC 411 - ASSIGNMENT 1
LINKAGE HANDLING, FORTRAN/ASSEMBLER AND OBJECT DECKS

DUE _____

This writeup: pages 01 - 02.

I. MAXIM FUNCTION SUBPROGRAM

Write a function in assembler language, consisting of 1 CSECT named MAXIM, which accomplishes the following:

A. Follows standard OS/360 calling conventions, receiving the address of an argument list in register 1. There may be a variable number of arguments in the list. Each address in the argument list points to a fullword somewhere in storage.

B. The program should algebraicly compare the fullwords addressed by the argument list, and place the value of the maximum one in register 0 as a result, then return control to the caller.

C. In writing this program, DO NOT USE XSAVE OR XRETURN macros.

II. WRITE FORTRAN TEST PROGRAM TO TEST MAXIM

This program should test MAXIM by using it as a function, i.e., it will have statements of the form I = MAXIM(1,2,-5,-10,4,5) in it. It should have at least two tests of this form. Print out the results to show they are correct. Use the above set of data plus another of you own choosing.

III. WRITE ASSEMBLER MAIN PROGRAM TO TEST MAXIM

Write an assembler main program which has the same logic and test values as does the previous FORTRAN program. It should use standard OS/360 linkage, its own save area (of course), and utilize the IBM macros SAVE and RETURN. It should do the following:

A. As given by the LINKAGE writeup, obtain the PARM field (see if any exists by testing for zero length). Either print the PARM field, or the message NO PARM EXISTS. Assume first character of PARM is a legitamate carriage control.

B. Make same two calls on MAXIM. The first must be hand coded (no macros), while the second uses the IBM macro CALL.

IV. JOB CONTROL LANGUAGE

Do not punch an object deck until you are sure the MAXIM program is correct. While testing, you can use the following JCL:

```
// EXEC ASGC
//SOURCE.INPUT DD *
..... MAXIM program.....
// EXEC FGCG,PARM.DATA=MAP
//SOURCE.INPUT DD *
..... main program
..... should do a  CALL LETDMP  before using MAXIM
//DATA.SYSUDUMP DD SYSOUT=A   (may need XSNAPOUT card following also)
```

An alternate form of the preceding is to use:

```
// EXEC FGC
//SOURCE.INPUT DD *
..... FORTRAN main program
// EXEC ASGCG,PARM.DATA='MAP'
//SOURCE.INPUT DD *
..... MAXIM
//DATA.SYSUDUMP DD SYSOUT=A
```

When testing MAXIM with the assembler program, you may either run it and MAXIM as one assembly (1 ASGCG), or as two (1 ASGC and 1 ASGCG).

V. WHAT TO HAND IN

A. Run a job which produces the object deck for MAXIM, and also does the test using the assembler main program (note that the END card in the second assembly should specify the name of the main program on it so that execution will start there.) Use the following deck setup:

```
// EXEC ASGC,PARM=DECK
//SOURCE.INPUT DD *
..... MAXIM
// EXEC ASGCG,PARM.DATA='MAP/0I AM A PARM FIELD'
//SOURCE.INPUT DD *
..... main program
*** sysudump and xsnapout cards, if needed
```

B. Using the object deck produced by the previous program, run this test of MAXIM with FORTRAN:

```
// EXEC FGCG,PARM.DATA='MAP'
//SOURCE.INPUT DD *
..... FORTRAN main program
//DATA.DECK DD *
..... object deck from MAXIM
***  sysudump and xsnapout cards as needed.
```

VI. THOUGHT QUESTIONS

A. Suppose that you also wrote MAXIM as a FORTRAN function. What would you do to obtain an object deck of it and use it as a subprogram of your assembler test program?

B. Does MAXIM need to have its own save area? If so, why? If not, why not?

C. How does the object deck of MAXIM compare with its source deck? Does the answer tell you why people use object decks?

D. What reasons can you think of for using a mixture of FORTRAN programs an assembler programs? What does FORTRAN do well that assembler does not, and vice-versa?

01/09/73: date of last revision

COMPUTER SCIENCE 411 - GENERAL INFORMATION

This writeup provides general information regarding CMPSC 411 - SYSTEMS ORGANIZATION AND PROGRAMMING, as currently taught at PSU. It notes the prerequisites, text materials, handouts, assignments, and generally describes what is taught in this course, and what is expected of the students taking it.

INDEX

I. PREREQUISITES

A. CMPSC 102 (or 410) or equivalent: effectively, a fairly complete introduction to much of System/360 computer structure and assembler language programming. The following should have been covered:

S/360 structure: registers, PSW, memory organization, common interrrupts, two's complement arithmetic. Programming experience with most of standard instruction set, possibly some with decimal opcodes and conversions. Privileged operations are not expected to be well-known, and floating point operations will not be used in the course.

S/360 Assembler Language: familiarity with most of the things covered in the first half of the OS Assembler Language manual (sections 1-5). Most of the following terms or operations should be familiar: self-defining terms, location counter, literals, absolute versus relocatable expressions; USING, DROP, START, CSECT, ENTRY; various instruction formats; EQU, DC, DS, TITLE, EJECT, SPACE, PRINT, ORG, LTORG, CNOP, END. Some students may have done something with DSECTs, MACROs, and linkage of FORTRAN and Assembler modules, but this is not necessarily required. It is expected that most incoming students have done most of their programming under ASSIST, and are thus not yet proficient in debugging programs and reading completion dumps under OS/360 directly. Students are expected to have written at least half a dozen or more programs in Assembler, including typically a small two-pass assembler for a simple assembly language.

B. CMPSC 404 or equivalent: data structures: arrays, linked lists; tree structures, queues, stacks; perhaps a little on searching and sorting methods: hash tables, etc.

Anyone who has taken equivalent courses elsewhere or wishes to substitute other programming experience for the above should contact the instructor immediately, to make sure their background is adequate for the course.

II. TEXTBOOKS AND MANUALS

The text materials for the course are listed below. Many of the text items are IBM manuals, which are often in a continual state of change. Particularly, each time IBM offers an updated version of OS/360, many manuals are modified somewhat. It is generally desirable to have the manuals appropriate for the current version of OS/360 (which is Release ___ at this time). However, this is not necessary, as there are often only minor changes between one version of a manual and the next. In addition, there are combinations of differently-named manuals which are equivalent to others. In the list below, the most desirable manuals are given, but equivalents are noted where possible.

Items coded R are definitely required, items coded D are desirable, while the remaining ones are useful, but can be done without, and may not even be available. Abbreviations to be used later are given in in brackets for each one. NOTE: for IBM manuals, first six digits show the specific manual, while the remaining one(s) indicate the version. Normally, (but not always), manuals having close version numbers are not very different.

D <STRUBLE> 1. STRUBLE:ASSEMBLER LANGUAGE PROGRAMMING: THE SYSTEM 360

This text will be referenced at most occasionally, but contains some more readable explanations than some of the manuals below. It is also good for review of CMPSC 102(410).

R <ASM> 2. GC28-6514-8 IBM S/360 OS ASSEMBLER LANGUAGE

This is heavily used throughout the course. By the 5th week of term, students will be expected to understand almost everything in this manual, while knowing offhand much of it. Besides the few sections in the first half not already known to the students, the entire second half (MACROs) will be covered.

R <POP> 3. GC28-6821-8 IBM S/360 PRINCIPLES OF OPERATION

This also will be used heavily. The students should be familiar with much of the material, but it may be needed for review of some operations (such as TR, TRT, ED, EDMK). It will also be needed for the following topics: system structure: protection features, I/O; status switching: program states, protection, PSW, instructions; interruptions: all in this section; input/output operations: most of this material: CAW, CSW, CCW, basic operation of channels.

R <INTRO> 4. GC28-6534-3 IBM S/360 OS INTRODUCTION

OR

R <C&F> GC28-6535-7 IBM S/360 OS CONCEPTS AND FACILITIES

These manuals (INTRO is effectively a reworking of the older C&F), give an overall view of operating system services, using OS/360 terms in particular.They do not explain things in detail, but give general concepts and vocabulary. The student must be familiar with most of the concepts and terms in these manuals by the end of the course.

The following group of manuals is continually changed around by IBM, with various parts shuffled among manuals of same or changing names or numbers. There are effectively 4 distinct modules of information:
a) Supervisor Services - description of general concepts.
b) Supervisor Macro Instructions - coding details for these macros.
c) Data Management Services - description of general concepts.
d) Data Management Macro Instructions - coding forms for these.

All 4 of the above modules are definitely necessary, but they are combined in various ways, with any combination providing all parts being generally acceptable, although the first combination is preferred. Each manual notes which information (a,b,c,d) it contains. Later references may refer to <S&DM>(x), where x is a,b,c, or d. In such cases, the information can be found in any of the manuals which have that section of information.

R <S&DM> 5. XXXX-XXXX-X SUPERVISOR AND DATA MANAGEMENT SERVICES AND MACRO INSTRUCTIONS

R <SS&M> GC28-6646-6 IBM S/360 OS SUPERVISOR SERVICES AND MACRO INSTRUCTIONS
(a,b) This is recommended version, and contains the general methods used for management of programs (including linkage conventions), tasks, and main storage allocation, with the macros for these.

R <DMSG> GC26-3746-1 OS DATA MANAGEMENT SERVICES GUIDE
(c) This is recommended form, and gives the general methods and many examples of processing each of the different types of datasets in different ways. Good explanatory material is given on various characteristics of data sets (record formats, control characters, etc), direct-access devices, magnetic tapes, and general procedures of data management (OPEN, CLOSE, DCB, GET, PUT, READ, WRITE, etc).

R <DMM> GC26-3794-0 OS DATA MANAGEMENT MACRO INSTRUCTIONS
(d) This is recommended form, and describes in detail the various ways of coding the data management macros. The manual just previous is read for understanding; this one is needed for actually writing such programs.

R <SS> GC28-6646-5 S/360 OS SUPERVISOR SERVICES
(a) This is explanatory part of <SS&M>, older version.

R <DMS> GC26-3746-1 S/360 OS DATA MANAGEMENT SERVICES
(c) This is explanatory part of <DMSG>, older version, and not as well written as <DMSG>.

R <S&DMM> GC28-6647-4 IBM S/360 OS SUPERVISOR AND DATA MANAGEMENT MACRO INSTRUCTIONS
(b,d) Older version of material in <SS&M> and <DMM>.

R <S&DMS> GC28-6646-3 IBM S/360 OS SUPERVISOR AND DATA MANAGEMENT SERVICES
(a,c) Older version: same as <SS> and <DMS> put together.

D <LE&L> 6. GC28-6538-9 IBM OS LINKAGE EDITOR AND LOADER

This describes how to use the named programs, and is particularly useful and necessary for anyone writing overlay programs or concerned with management of program libraries. The beginning contains fair descriptions of object and load modules and their processing.

D <JCLR> 7. GC28-6704-2 IBM S/360 OS JOB CONTROL LANGUAGE REFERENCE

This manual completely defines JCL, gives various coding forms and examples of JCL usage. It is somewhat difficult to read as a text, but is valuable as a reference. This manual is a rewritten version of the combination of the two following manuals (which together contain much redundant information). Therefore, any one of these three manuals are acceptable, although this is the best.

OR

D <JCLR1> GC28-6704-1 IBM S/360 OS JOB CONTROL LANGUAGE REFERENCE

Although mostly like <JCLR>, this contains fewer examples.

OR

D <JCLUG> GC28-6703- IBM S/360 OS JOB CONTROL LANGUAGE USER'S GUIDE

This version leans more towards examples rather than a reference.

R <GREEN> 8. GX20-1703-9 IBM S/360 REFERENCE DATA

This is the GREEN CARD, and contains much useful information in a compact form, including information on ED/EDMK patterns, contants, assembler instructions, condition code setting, interrupt codes, radix conversions, formats of PSW, CAW, CCW, CSW, permanent storage assignments, most common CCW opcodes, instructions and EBCDIC bytes.

R <ASSIST> 9. ASSIST INTRODUCTORY ASSEMBLER USER'S GUIDE

This describes usage of the ASSIST assembler. A slightly outdated version (1.0) is available in the CMPSC office (426 McAllister), while an updated version may be available on BAT files (ask instructor).

III. WRITEUPS

This section provides an alphabetic list of various explanatory writeups and assignment handouts which are available for use in CMPSC 411 or other courses. Some of these may be handed out, while others can be accessed by anyone who is interested. Names, number of pages, source, and description are given for each.

The following notations may appear for the SOURCE of each writeup:

BAT The writeup exists as as PSU CC BAT file, accessible from RJE or batch terminals. An otherwise blank card with a comma in column 1 indicates the beginning of each page, including the first. Note that these normally contain both upper and lower case letters.

DTO The writeup exists only as a dittoed handout.

TAP The writeup exists on a tape, in which case the tape name, file number, and file name are given.

CRD The writeup is in a punched card deck.

MTS The writeup is on an MT/ST tape.

If the description of a BAT file begins with (userid), that is the userid under which the file is saved. If not mentioned, the file is saved under the following userid:

JRM02

Several utility programs are available for printing or punching any BAT file(s). JRM02.PRINT prints any input, converting lower case letters to upper case, while JRM02.PUNCH punches its input. Both will run in any category, including category W. The deck setup is:

```
//    JOB CARD
/*INCLUDE JRM02.PRINT         substitute PUNCH if desired
@@INCLUDE userid.filename     1 or more cards like this
@@                            to terminate input
```

The BAT files generally have about 50 lines per page, with a limit of 500 lines in any one file. Some writeups consist of several BAT files together, while other BAT files contain several distinct writeups.

NAME	PAGES	SOURCE	DESCRIPTION
ASBROPS2	3	BAT	assignment using the ASSIST REplace Monitor: each student replaces the base register table portion of ASSIST: exercies in table search or linked list manipulation; USING, DROP, address conversions.
ASPRGTC1	8	BAT	gives S/360 Assembler programming hints: how to use modules, macros, and combined forms; how to set up safe, nondestructive linkage to a module.
ASREPLGD	11	BAT	ASSIST REPLACEMENT USER'S GUIDE : describes in general terms how to use the ASSIST Replace Monitor.
CS411AS1	2	BAT	assignment: linkage between FORTRAN and Assembler, linkage conventions, argument passing, PARM field access.
CS411GI1	10	BAT	the writeup you are looking at.
CS411GI2	##	BAT	
CS411MC1	8	BAT	two assignments on macro-writing: pages 01-03 have one to write own version of CALL, SAVE, RETURN macros and test them; pages 04-08 have combined macro/module writing: hexadecimal conversions and dumping; various macro features illustrated.
CS411MC2	6	BAT	assignment: write package of macros to manipulate one-way linked lists. each macro is fairly easy.
CS411FP1	8	BAT	assignment: write simulation of typical batch
CS411FP2	8	BAT	multiprogramming computer system. Many possible
CS411GP3	8	BAT	combinations of scheduling/allocation/resources are
CS411FP4	7	BAT	available. Uses almost all features in BAL.
CS411FP5	6	DTO	example flowcharts for this project.
DOCUMENT	4	BAT	S/360 Assembler Language documentation hints and good practices.
DSECT	3	BAT	example of use of dsects to trace save areas, also showing assembly listing of program.
DUMPSJCL	3	BAT	gives basic Job Control Language cards for running most typical assembler programs; gives several useful hints on special JCL available at PSU; gives programs to be run to obtain representative system completion dumps.

NAME	PAGES	SOURCE	DESCRIPTION
DUMP1	8	DTO	debugging and dump-reading hints for assembler programmers; concentrates more on use with Link Editor, but has some material with Loader. Gives examples of hunting down causes of errors. (should be upgraded and rewritten)
DUMP2	7	DTO	
HARDWAR1	5	BAT	describes most devices currently part of PSU CC 360/67 system; gives device addresses, speeds, capacities; channel priorities. Page 5 is only DTO, and it contains diagram of 360/67 layout.
LINKAGE	5	BAT	explanation of OS/360 standard linkage conventions used by FORTRAN, Assembler, etc programs for entry, exit, argument passing.
OSHASP	9	BAT	OS/360 with HASP: explains how OS/360 is loaded into memory (IPL-NIP), how it runs, and how it is modified by the use of HASP.
XDUMP	5	BAT	describes use of XDUMP and XSNAP debugging macros (XDUMP is simple form used in ASSIST, XSNAP is more complex). (JRM04.XDUMP)
XGET	2	BAT	describes use of generalized XGET/XPUT I/O macros
XHEXI	3	BAT	describes use of XHEXI & XHEXO hexadecimal conversion macros (available in ASSIST). (JRM04.XHEXI)
XREAD	3	BAT	describes XREAD, XPRNT, XPNCH I/O macros. available in ASSIST. (JRM04.XREAD)
XSAVE	7	BAT	describes XSAVE and XRETURN linkage macros. available in ASSIST only if using the macro library. (JRM04.XSAVE)

In addition to the above writeups, the students will normally purchase a copy of the ASSIST Introductory Assembler User's Guide. The very latest copy of this manual is available in the following JRM04. BAT files:

ASSIST1			
ASSIST1A	14	BAT	Part I: describes language availabel in ASSIST
ASSIST2	8	BAT	Part II: describes debug and I/O instructions.
ASSIST2A		BAT	
ASSIST3			
ASSIST3A	9	BAT	Part III: control cards, parameter options.
ASSIST4			
ASSIST4A	17	BAT	Part IV: output format, error messages, loader

IV. BAT FILES

The following lists BAT files which are NOT text material, i.e., sample programs, test data for some projects, etc.

NAME	CARDS	SOURCE	DESCRIPTION
$BRTEST	178	BAT	test data for the ASSIST base-register module replacement assignment (see assignment ASBROPS2)
ATTACH	???	BAT	multi-tasking example: shows ATTACH, CHAP,DETACH,etc
BDAM1,BDAM2	?	BAT	run ot illustrate Basic Direct Access Method
BPAM	322	BAT	an entire run, with JCL, to illustrate use of BPAM macros for accessing pPartitioned Data Sets. Reads selected macros from macro libraries and prints them. Shows FIND, BLDL, READ, CHECK, etc.
BSAM	157	BAT	entire run to illustrate BASM macros. shows READ, WRITE, CHECK, etc. reads cards, writes on disk, reads from disk, prints.
CS411FPJ	74	BAT	test deck of $$JOB cards for Final Project (see assignment CS411FP1,2,3,4,5)
CS411FPK	110	BAT	same as CS411FPJ, but with different $$JOB cards.
DUMPTEST	84	BAT	contains 4 runs, set up to produce dumps for students to look at. (see DUMPSJCL writeup). note that each run is preceded by BAT file comma card, so that they may be extracted by listing with the PAGE= option to find their starting sequence #'s.
EXCP	102	BAT	contains complete run, set up to read cards and print them using EXCP macro and CCW strings.
FLOTLINK	152	BAT	contains a complete run, set up to show various combinations of FORTRAN and assembler linkage. illustrates most floating point instructions, by computing a function in FORTRAN, then using some equivalent assembler code.
GETMAIN	???	BAT	program to illustrate use of GETMAIN/FREEMAIN
LINKLOAD	???	BAT	program to illustrate load module management: LINK, LOAD, XCTL, DELETE macros, etc.

NAME	CARDS	SOURCE	DESCRIPTION
OVLY1	276	BAT	OVERLAY example: complete run, uses link-editor to edit a single object module several different ways, showing the amount of storage which can be saved by using overlay methods. includes multiple REGIONs.
PTPCHMAC	22	BAT	uses the IBM utility program to print 3 macros from SYS1.MACLIB. illustrates use of IEBPTPCH, allows students to look at macros if they want to. entire run setup. shows SAVE, RETURN, CALL.
QSAM	126	BAT	Queued Sequential Access Method: reads from cards, writes to disk blocked (illustrating use of DCB from JCL DD card), reads from disk, prints. shows GET, PUT, OPEN, CLOSE, DCB. shows all sources of information for DCB.
RECURASM	???	BAT	illustrates recursive assembler program using GETMAIN/FREEMAIN macros
SPIESTAE	???	BAT	shows use of error-interception macros SPIE/STAE
TIME	???	BAT	test program to show use of timing macros: TIME, STIMER, TTIMER
WTOWTL	???	BAT	test run to show use of WTO, WTL macros for communicating with computer operator

V. ASSIGNMENTS/ DUE DATES

The following lists the assignments given, approximate due date of each (in terms of day number within the 30 class days per term), plus comments on each.

#	DUE	NAME/DESCRIPTION/COMMENTS
1	3	DUMPS - run dump programs as described by DUMPSJCL, bring to class: familiarizes students with dump reading.
2	6	LINK - assignment CS411AS1 : gives students practice with linkage, including FORTRAN/aseembler linkage, dumps. gets them programming again quickly.
3	9	SAVE/RETURN - CS411MC1 (pages 01-03) - write own versions of extended SAVE, RETURN, CALL macros: starts students on macros.
4	12	HEX: CS411MC1 (pages 04-08) - write hexadecimal conversion macros & modules, like XHEXI, XHEXO. covers macro/module linkage, TR, TRT, PACK, UNPK instrs.
5	15	BASEREG - ASSIST base register replacement - ASBROPS2, $BRTEST, ASREPLGD . covers base registers well in short program. helps with understanding of DSECTS.
6	18	LINKED-LIST: CS411MC2 - linked-list macro package - more practice on macros, needed for final project.
7	27	FINAL PROJECT: CS411FP1,2,3,4,5, CS411FPJ, CS411FPK - operating system simulation.

VI. COURSE OUTLINE/READINGS

A. OVERALL OUTLINE

This section gives a very brief outline of lecture topics by day. Several empty days are left for exams, problems, and possible expansion of any of the topic areas.

DAY	TOPICS
1	Introduction to Course; Prerequisistes; Review X-Macros
2	OS/360 Linkage Conventions
3	Dump-reading; Debugging & good programming technqiues; Common
4	interrupts and interpretation of dumps.
5	Overall macro concepts & structure; FORTRAN analogies.
6	Details on macros; examples; most statement types.
7	Finish macros; Aspecial argument handling; character scanning.
8	Miscellaneous cleanup on S/360 Assembler. (CNOP, V-cons ,etc)
9	Addressibility; DSECTS; Multiple USINGS; Useful techniques.
	*** at this point, S/360 Assembler Language is finished ***
10	Assembler Comparison: 1,2,4 pass (SPASM, ASSIST, Assembler G)
11	Introduction to operating systems: history, basic types.
12	Architecture summary: CPU, Memory, Devices, Channels; Memory
13	structure and communication; memory protection.
14	I/O devices: sequential, DASD, capacities, characteristics.
15	Finish I/O devices; I/O Channels: types and programming; S/360
16	Channel-level programming; Interrupt handling; show examples.
17	Final Project: explain simulation concepts; system overview.
18	Module management; types of modules (REENTRANT, etc); loader;
19	Link-editor; Program Fetch; Overlays.
20	User overview of OS/360 services.
21	I/O concepts: buffering, record formats, blocking, etc.
22	Survey of common OS/360 macros (except data management)
23	JCL
24	JCL
25	OS/360 and HASP internal structure; IPL, NIP, etc. ASP.
26	Assorted topics: microprogramming; cache and virtual memories.
27	Assorted topics: pipeline and array computers; non-IBM systems
28,29,30	left for exams, problems, expansion, etc.

COMMENTS: the schedule above is fairly tight. It has been useful, especially in the first several weeks, to give 'help sessions' once a week in the evenings, to go over problems, review, etc. This has been useful especially to even out differences among 102/410 courses taken at different times and/or different campuses. Note that the order of topics above seems to work out fairly well, since it gives the students the material needed for the assignments fairly early.

Note that some explanation of JCL should be given thoughout the term, when relevant to specific assignments, etc, so that the two days allotted to JCL contain a unified explanation, but assuming that the students should have some familiarity with it by then.

B. DETAILED OUTLINE, REFERENCES

DAY	TOPICS
1	Outline of course; administrative details; grading; exams; course prerequsisites; review X-Macros: XREAD, XPRNT, XPNCH, XDUMP, XSNAP, XDECI, XDECO, XHEXI, XHEXO, XSET. Note that a few of these may NOT be review, but new material. REFS: X-MACRO Writeups.
2	OS/360 Linkage Conventions: go over in detail;

COMPUTER SCIENCE 411 - FINAL PROJECT

DUE _____

This writeup: CS411FP: 1(01-08), 2(01-08), 3(01-08), 4(01-07), 5(01-06)

I. INTRODUCTION

A. PURPOSE OF ASSIGNMENT

This assignment requires the student to design, code, debug, and test a program which simulates the execution of a fairly general operating system (OS/411) under a variety of circumstances. It requires the student to become familiar with a number of different strategies for implementing operating system components, and provides experience in working with programs of nontrivial size and complexity.

This project provides experience with the following operating system technqiues: job scheduling, storage allocation, processor dispatching, I/O request handling, and general job processing.

The following programming techniques can be included in this program: linked list manipulation, queueing methods, and random number applications.

In particular, the assignment generally requires the use of the S/360 Assembler Language items: multiple CSECTS, DSECTS, heavy use of macro definitions, and use of SET variables among macros and in open code.

B. PROCEDURES FOR WRITING AND RUNNING THIS ASSIGNMENT

1. This writeup describes many alternate ways of performing the actions needed in an operating system. Although students should study the various options performed, they will NOT write the code to implement all of the options. One option (or several, to be compared), will be chosen by the instructor to be coded by the students. THE STUDENTS WILL DEFINITELY NOT BE REQUIRED TO WRITE EVERYTHING DESCRIBED HERE, ALTHOUGH THEY SHOULD ATTEMPT TO BECOME FAMILIAR WITH THE VARIOUS OPTIONS.

2. In some cases, there are variables which are to be given certain values for test purposes, but which are to be represented by GLOBAL SET VARIABLES in the student program. These SET VARIABLES may be referrred to in various parts of this writeup. The values to be used will be given by the instructor(to fill in chart on page CS411FP4 - 06).

3. The student will write ONLY ONE source program to cover all of the different options desired. Alternate versions of a particular method will be selected or deleted using conditional assembly in open code (i.e., using GBLA, GBLB SET VARIABLES and AIF, AGO commands).

4. The instructor will supply test decks, and request that they be run, given 1 or more sets of options/parameters. The student will then generate the required version(s) of his program by changing ONLY a few SETA, SETB, SETC statements at the beginning of his program, which will create the specific version needed. This process of using SET variables to generate the system is referred to as a SYSGEN.

5. It should be noted that some of the terminology employed in this writeup is not exactly standard, in particular the word QUEUE is used quite often. This usually is taken to mean a list (usually ordered on some key), from which only the first item can be removed, and new items only added to the end. Although this may be the most common case, for this writeup the items in some queues may be inspected, modified, added and deleted in any position. For example, it may be necessary to scan a queue in order for the first item meeting a specified test, then removing that item.

C. A BRIEF DESCRIPTION OF THE PROGRAM AND WHAT IT DOES

The program to be written will read an INPUT STREAM, which is made of one or more BATCHes of JOB cards. Each JOB card descibes the needs and characteristics of a single JOB. The program will then simulate the handling of this job by a typical operating system.

OS/411 handles each BATCH of one or more JOB cards as follows:

First, OS/411 initializes itself to a status in which there are no JOBS at all in the system, and all counters, flags, etc are set to their desired beginning values. The JOB cards will then be read (either immediately or at various intervals during the run), until the BATCH of JOBS is finished executing.

Reading a JOB card begins a long sequence of actions which must be performed to simulate the actions needed to run a JOB.

First, the JOB card is scanned, and all information is recorded from it. The JOB is entered on a queue of JOBS which are not yet able to obtain use of a Central Processing Unit (CPU). Basically, this step represents the process of reading in a complete JOB and storing it on magnetic disk before it can be executed.

Whenever it might be possible that some JOB be loaded into memory and EXECUTED, the queue of waiting JOBS is scanned, and a JOB is perhaps selected (according to one of many possible rules). This JOB is then INITIATED, i.e., allocated MEMORY, and made ready for EXECUTION. It is thus free to compete with other such ACTIVE JOBS for the use of the CPU (or one of several, if such exist).

The list of ACTIVE JOBS (ones in memory), is periodically scanned (according to one of several algorithms) and a JOB is selected to receive control of a CPU for some period of time, i.e., the JOB is DISPATCHED.

After some length of time during which the JOB has use of the CPU, it may make an Input/Output Request, in which case it relinquishes control of the CPU (so that another ACTIVE JOB, if any, may get it). The IO REQUEST needs the use of an IO CHANNEL, of which there are one or more. Depending on the needs of the JOB, it either gains control of a CHANNEL, and uses it to perform its IO immediately, or else it enters a queue of JOBS competing for the use of CHANNELS. In the latter case, it may have to wait a while until it is selected to use the CHANNEL it needs. It is said to be in WAIT STATE at any time during which it is READY to use a CPU or CHANNEL, but cannot obtain what it needs.

When a JOB is finished using a CHANNEL, an IO INTERRUPT is said to occur. This means that the JOB relinquishes control of the CHANNEL, becomes READY to use a CPU, and thus enters the list of such JOBS, once more competing with the others. The CHANNEL thus released of course becomes available for use by other JOBS again, and may be used to fill a request from some previous JOB, which is WAITING for the CHANNEL.

Among the values found on the JOB card is a time limit for the execution of the JOB. When this limit is reached, the JOB is considered to be finished. It is then JOBTERMED, i.e., it is removed from the 0 list of JOBS competing for CPU usage, the memory it occupies is freed for possible allocation to other JOB(S) (i.e., the ones waiting on disk to obtain memory).

At various times, calculations are made to determine some values indicating the performance and nature of the particular operating system version being generated. Typical times are the end of a JOB and the end of a BATCH of JOBS.

The process described above occurs for every JOB in a BATCH. Each BATCH is processed, until none remain. CS411FP5-01 has job flow chart).

The remaining portions of the writeup describe: the configuration of the computer system begin run by OS/411, the major options possible, SET VARIABLES which may be needed, and many hints on the implementation of this program. The last includes general methods, ideas on good ways to divide the program into modules, sample data structures and DSECTS, and overall flowcharts which might be useful.

Every option is given a mnemonic name of some sort, and SET VARIABLES are of course named as they might be inside OS/411.

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
*                                                                           *
*         NOTE:  HINTS, DSECTS, AND FLOWCHARTS GIVEN IN THIS WRITEUP  *
*    ARE ONLY SUGGESTIONS.  THE STUDENT MAY BE ABLE TO DO THEM MUCH   *
*    BETTER THAN THE WAYS OUTLINED.  IN PARTICULAR, THESE THINGS ARE  *
*    USUALLY WRITTEN TO COVER ALMOST ALL OF THE POSSIBLE OPTIONS GIVEN*
*    IN THIS WRITEUP.  AS A RESULT, THEY ARE MUCH MORE GENERAL THAN   *
*    WILL EVER BE NEEDED TO COVER ONLY ONE-TWO OPTIONS FROM EACH      *
*    GROUP OF OPTIONS.  THUS THESE TABLES AND FLOWCHARTS ARE USUALLY  *
*    MORE COMPLEX THAN NEEDED FOR THE VERSIONS REQUIRED TO BE TURNED  *
*    IN FOR GRADE, LEAVING MUCH ROOM FOR CHANGES.                     *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

II. COMPUTER CONFIGURATION

The OS/411 operating system is to be designed for use with a paticular model of the famous (or infamous, depending on viewpoint) System/411 series of computers. This section lists the various sizes and nature of systems which may occur.

A. MEMORY SIZE

&MEMSIZE : gives the size of the S/411 computer, in K (1024) bytes. &MEMSIZE may range from 1 to 1024, and OS/411 may or may not be required to run on different-sized machines. Some versions of OS/411 may be written to run only with one memory size, thus allowing for more efficient special-case programming techniques.

B. NUMBER OF CENTRAL PROCESSING UNITS (CPU)

A CPU is one of the most important resources to be needed for the execution of a JOB. A basic S/411 system contains only 1 CPU, while the bigger models may contain more than one. The possible options are:

CPU1 Only 1 CPU exists.

CPUv &NUMCPUS gives the number of CPU'S in the system. &NUMCPUS may range from 1 to an upper limit to be given.

C. NUMBER OF INPUT/OUTPUT CHANNELS

Performing input/output for a user JOB requires the use of an IO CHANNEL. The following are the possible options:

CHN1 the system is a small one with only 1 IO channel, which all user jobs must use when they need IO.

CHNx the number of I/O channels is fixed at the number x. Channels are numbered from 1 to x.

CHNv &NUMCHNS gives the number of channels, with &NUMCHNS varying from 1 to some given upper limit, and OS/411 must be written to handle any possible value. Channels are numbered 1 to &NUMCHNS in this case. No system has more than 15 channels.

III. INPUT JOB STREAM

OS/411 reads a deck of input cards, which contain cards which describe JOBS, and may contain other kinds of cards also. The kinds of cards possible are as follows:

A. JOB CARD - DESCRIBES CHARACTERISTICS OF ONE JOB

Each BATCH of JOBS consists of from 0 - to some maximum number of JOBS. This maximum number may be a constant or be given by &MAXJOBS . Each card is of the following format (starting in column 1):

```
$$JOB jobname T=x,SP=x,IOIN=x,IORL=x,PRIO=x,CHAN=x,CAT=x
```

jobname is a 1-8 character name, separated from other items by 1 or more blanks on each side. It is a unique name which is used to identify the specific job in any messages.

The rest of the JOB card consists of the parameters shown, WHICH MAY OCCUR IN ANY ORDER. The following options are possible:

PARM1 all of the parameters will exist on any JOB card, and no errors need be tested for in them.

PARM2 some parameters may be omitted or in error, in which case DEFAULT values are to be set at SYSGEN time and used instead. The corresponding SET variables are then named by &, the name of the parameter, and DFT, i.e., default T= value is given by &TDFT, etc. So value of &PRIODFT is used if PRIO= is omitted.

PARM3 This includes PARM2 above, except that the defaults are done by JOB CATEGORY (the CAT= value), so that omitted options from JOBS in different categories will have different defaults.

All the values of x above are unsigned decimal numbers, and the specific meaning of each parameter is given below.

T= a number from 1 to 32767, giving the number of milliseconds which is a time limit on the execution of the JOB. As will be given in a later section, this always counts use of the CPU, but may or may not count use of a CHANNEL.

SP= (SPace) - a number from 1 - 1024, giving the number of K-byte blocks of storage required to execute a JOB.

IOIN= (I/O INterval) - a number from 1 - 32767, which describes the interval between I/O requests from a job. value is in millisecs.

IORL= (I/O Request Length) - number from 1 - 32767 describing the duration in milliseconds for one I/O request by the JOB.

PRIO= (PRIOrity) - a number from 0 - 255 specifying the relative priority of this job, with 0 highest and 255 lowest (i.e., if all other things are equal, prioirty 0 gets preference over 255).

CHAN= (CHANnel) _ specifies a number from 1 to the maxium number of channels in the system. This specifies a particular channel which the JOB needs to use to perform I/O.
=0 specifes that the JOB may use ANY available channel in the system when it needs to perform I/O.
Of course, if option CHN1 is used, this parameter can be ignored. If CHAN= specifies a channel number higher than the maximum one, &NUMCHNS, CHAN=0 should be assumed instead, i.e., this is NOT an error, but allows use of any available channel.

CAT= (CATegory) - is a number from 1 to a maximum of 15(although some OS/411's may allow a smaller maximum). Specifies in some way the type of processing a JOB is to receive. Some versions of OS/411 may not utilize this option at all.

The above options include many things often found on JOB cards for various systems. Others MIGHT be required, such as a date or time by which a JOB must be finished, or limits on output (lines printed, cards punched, or limit on the sum of these).

B. $$CLEAR CARD - INDICATES END OF A BATCH

When this card is found, it requests the following actions:

1. The simulation is continued until all JOBS currently executing or waiting for execution are completely processed. However, no more $$JOB cards are read during this period of time.

2. If required, print out a report giving statistics describing the entire previous BATCH of JOBS (watch out for case where there were 0 JOBS in the BATCH, as when $$CLEAR card is first one in deck.

3. Reset all necessary counters, lists, work queues to such status as to allow acceptance and simulation of another BATCH of cards. I.e., OS/411 must be SERIALLY REUSABLE.

C. $$QUIT CARD - FINISH ENTIRE RUN

This card indicates that no more BATCHes of JOBS follow, i.e., so that this functions like the $$CLEAR, except that no more BATCHes are processed. NOTE: an END-FILE INDICATION SHOULD BE TREATED AS A $$QUIT CARD, i.e., if a read finds nothing there, it should take this action.

D. $$DEBUG CARD - DEBUG ACTION CARD

This card has the following format:

$$DEBUG number,number.....

This card is mainly for debugging use, and supplies the various numbers to whatever debug counters/flags you may wish to use. This may be particularly useful to turn on/off trace and debug output.

IV. JOB INPUT AND SCHEDULING TECHNIQUES

Briefly, when OS/411 reads a $$JOB card, it scans it, evaluates the various parameters on it, and makes up an entry for it on the list of JOBS waiting on disk until they can be executed. The list is generally kept in the order in which the JOBS should be executed if possible. The options following describe HOW OFTEN a new JOB arrives (i.e., how often new JOB cards should be read), IN WHAT ORDER the list of WAITING JOBS should be kept, and WHAT STRATEGIES could be followed to decide which (if any JOB) should selected from this list and INITIATED (loaded into memory and executed).

A. JOB READING OPTIONS

These options describe the intervals between successive arrivals of JOBS into the OS/411 system, and are:

RDR1 A $$JOB card is read at an interval fixed at SYSGEN, arriving every &RDRINT milliseconds.

RDRR A $$JOB card is read at intervals which are random according to some probability distribution. Among the possibilities:

- RDRRU &RDRINT is the mean of a UNIFORM distribution, i.e., intervals vary between 0 and 2*&RDRRINT.
- RDRRE the intervals between successive JOBS are obtained from an EXPONENTIAL distribution, with &RDRRINT as a mean (this is called a POISSON ARRIVAL PROCESS, and is actually the most realistic of the arrivals given here).

B. JOB ORDERING OPTIONS

After the parameters of a JOB have been scanned, it joins a list of JOBS (which may of course be empty) which are waiting for execution. In general, the position of a JOB on the list usually determines how soon it can be executed; the first one on the list should be the next one executed (if possible, according to which set of scheduling rules is being used). Combinations of the various options may be used, with the following general philosophy:

If all other things are equal between two jobs, then the one placed earlier on the list should be the one :

which arrived in the system earlier and has thus waited longer.
having the higher priority (lower PRIO= value).
smallest storage requirement (smaller value of SP=)
smaller running time requirement (T= value)
smaller category number (CAT=)

The following order options can be considered:

JO1 FIFO (First In, First Out) or FCFS (First Come, First Served)
A new JOB is always added to the end of the list, so that the earliest arriving JOB is listed earliest.

JO2 STRAIGHT PRIORITY : JOBS are kept in order from highest to lowest (low PRIO values to HIGH PRIO values). If two JOBS have equal priority, then the earlier-arriving one is first.

JO3 CATEGORY PROGRESSION : JOBS are kept in order by CATEGORY, from lowest CAT= to highest. Within each category, JOBS are listed either FIFO or by PRIORITY, or both.

JO4 LENGTH OF TIME REQUIRED BY JOB, either:

JO4S (SJF - SHORTEST JOB FIRST) : in order by T=, small to big.
JO4L (LJF - LONGEST JOB FIRST) : in opposite order from JO4S.

JO5 SPACE REQUIRED BY JOB, either:

JO5S smallest job first
JO5L largest job first

JO6 INPUT/OUTPUT REQUIREMENT SELECTION
Distinguish between I/O BOUND JOBS (IORL large relative to IOIN) and CPU BOUND JOBS (IORL small relative to IOIN) :

JO6I I/O BOUND JOBS earlier (small values of IOIN/IORL).
JO6C CPU BOUND JOBS earlier (small values of IORL/IOIN).

JO7 STATIC ORDERING COMPUTATION
When the JOB enters the system, a single number is calculated from some subset of the JOB'S parameters, thus weighting the various factors according to whatever the designer of OS/411 desires. (Note that all of the JO options above are really special cases of this). The JOB is then ordered according to this number, which is thus a generalized priority. For example: ORDER = (PRIO + T/256 + SP/4 + IOIN/IORL) * CAT favors high priority jobs, shorter jobs, smaller jobs, I/O BOUND jobs, and especially jobs in lower-numbered categories, assuming jobs are ordered from low ORDER to high ORDER.

JO8 DYNAMIC ORDER CALCULATION
This case includes all JO options as special cases, and allows the ordering of the JOBS to be varied dynamically, according to any system conditions. For example, it might be basically a priority system, but check the list of JOBS at intervals, and occasionally raise the priority of a JOB if it has been waiting a long time. It could also take into account the other jobs in a system, such as trying to give equal service to each category of jobs, or deciding to select an I/O BOUND JOB if CPU BOUND ones are in memory, or vice-versa.

C. JOB SELECTION RANGE AND METHODS

Briefly, a job is to be selected from the list of waiting jobs by scanning down the list until the first one meeting desired criteria is found. If no such job is found within the range of the list scanned, then no job is initiated at this time. The following options are a few of the ways in which this range can be specified.

JSR1 Initiate the first job if it fits in memory (according to the memory allocation rules given below). If it does not fit, do not initiate any other job, even though it may fit.

JSRT Initiate the first job on the entire list which fits.

JSRv Initiate the first job which fits, scanning up to &JSRANGE

Note the effects of the above rules: JSR1 ensures that the jobs are initiated in the order given by the JO option. JSRT does the best in keeping memory full, but it may also keep a large memory job waiting for a relatively long time. JSRv is then a compromise between the others.

D. MEMORY ALLOCATION FOR JOB INITIATION

Many different ways exist for allocating storage to jobs, the ways used depend strongly on the purposes of the system, and also on the computer hardware being used. For example, the S/411 computer series includes computers with a wide range of addressing/relocation circuits. The smaller ones require that a program be loaded as one contiguous block in memory, and never moved from that area. The medium systems use bounds-registers, so that although a program must be loaded in one contiguous block, it may be moved around in memory. The large S/411's may contain special page-translation hardware, which permits the program to be loaded as many noncontiguous pieces, and moved around as desired.

It should be noted that OS/411 uses a STATIC allocation scheme, i.e., a JOB requests memory only 1 time, when it is initiated. It does NOT request and return memory areas while executing. Some systems use a DYNAMIC allocation scheme, in which a program can request an area of a given size, and be supplied with the address of such an area, use it, then return it to the operating system later. Some systems combine both of these methods, i.e., they allocate an area of memory when the JOB is INITIATED, but allow programs to allocate/de-allocate space within that area. OS/360 is an example of such a method.

Finally, various methods exist for determining which of several blocks of unused memory should be allocated to a requestor. Each of the methods has advantages and disadvantages, both in implementation difficulties and in statistical properties.

The next pages describe SOME of the possible options.

1. MEMORY DESCRIPTION VALUES

The following SET variables describe the various parameters of memory allocation:

&MEMSIZE = # 1K blocks of memory available for entire system, up to 1024 (also mentioned previously)

&MEMOS = # 1K blocks allocated to operating system, and thus not available to be allocated for user programs.

&MEMPGR = page sized round value,i.e., all requests are to be rounded up to this size. EX: if = 1, every 1K block of memory can be allocated separately, if = 128, then each job is allocated at least 128K, i.e., jobs occupy 128K, 256K, 348K,...
This unit will be referred to as a PAGE.

&JOBLIM1 = an arbitrary limit on the maximum number of jobs in memory at any one time. Note that the special case &JOBLIM1 = 1 implies UNIPROGRAMMING, while &JOBLIM1 > 1 implies that MULTIPROGRAMMING is at least possible.

&JOBLIM2 = a calculated limit on the number of jobs possible in memory, assuming that each job must be allocated at least &MEMPGR K bytes of storage. This can be calculated as follows:
&JOBLIM2 = (&MEMSIZE-&MEMOS)/&MEMPGR

&JOBLIM3 = final limit on number of jobs, = MIN(&JOBLIM1,&JOBLIM2). This variable would be the one used to actually control code generation.

2. MEMORY ALLOCATION ALGORITHMS

The following are common techniques for determining WHICH block of memory will be used to satisfy a request. In each case, a block is given to a job for the duration of its execution, and then returned to unused status when the job terminates. In every case, this block just returned must be MERGED with any contiguous free block(s), so that they can together be used to satisfy a larger request. If this is not done, memory becomes FRAGMENTED into small free blocks.

MAA1 (FIRST FIT) - a table of free blocks is usually kept in order by address. The request is satisfied from the first area of size >= request size, with the unneeded portion remaining in the free table. This is usually the easiest to implement, and has the best statistical properties for most applications.

MAA2 (BEST FIT) - like FIRST FIT, except that storage is allocated from the smallest free area of size >= request size, thus trying to maintain large free areas. May be best under some conditions, although not usually as good as FIRST FIT.

MAA3 (BUDDY SYSTEM) - see KNUTH, Chapter 2 for this method.

3. MEMORY ALLOCATION - CONTIGUITY OF ALLOCATION

MAC1 (CONTIGUOUS, NO MOVEMENT) - the memory for a job must be allocated in one contiguous unit, and a job may never be moved from that area.

MAC2 (CONTIGUOUS, MOVEMENT) - the job must be allocated contiguous memory, but may be moved around in memory if necessary. (NOTE: a system must use bounds registers or more sophisticated setup in order to use this method).

MAC3 (NONCONTIGUOUS) - the required number of pages are allocated anywhere at all in memory. This method definitely requires special hardware, is most commonly used for Time-Sharing systems, and exists in one form or another on: XDS SIGMA 7, RCA SPECTRA 70/46, BURROUGHS B5500, B6500, etc., GE 645, and IBM 360/67 (used as 67 not as 65 like PSU does).
This type of system may be used to allow programs to be written which are not totally in memory, and which can appear to the user to be much larger than the actual physical memory available on the computer system.

4. MEMORY MERGE FOR ALLOCATED MEMORY

These options are only meaningful with option MAC2 above (why?).

MAM1 (AUTOMATIC COMPACTION) - memory is allocated from one end of storage, and whenever a job terminates, any jobs above it are moved down to fill the empty space. Thus at any point in time, the memory consists of 1 region of contiguous jobs, and 1 contiguous region of free space.

MAM2 (COMPACTION WHEN NECESSARY) - rather than compacting always, this method only moves the jobs around when necessary to initiate some desired job which is larger than any 'hole' currently existing, but smaller than the total available space.

With either option, all CPU(s) must be stopped while the programs are being moved around, since they cannot execute a job while in motion In this case, ALL CPU's are stopped from executing any jobs for the following length of time:

time used = (&MAMOVE * # K BYTES MOVED)/&MUMCPUS

i.e., the CPU's share the task of moving the jobs around as needed. Note that the decision on whether to move jobs or not is heavily dependent on the size of the &MAMOVE factor, which is essentially a move cost.

5. INITIATION DELAY FACTOR

&INITDEL (INITiation DELay) - give the number of milliseconds required to load a job into memory, once it has been chosen, i.e., it cannot begin execution until time NOW+&INITDEL.

V. JOB SCHEDULING DURING EXECUTION

When a JOB is INITIATED, it is added to a list of jobs in memory. This list may be called a Ready Job Queue (RJQ), since it is a list of jobs which are in memory, ready to execute, but not currently being serviced by a CPU. The order of this list determines which jobs may gain control of a CPU when one becomes available. Naturally, the list itself can be ordered in any way desired, and generally includes all of the methods described in IV.B. (the JO options), plus some additional others commonly used only for CPU allocation.

Briefly, the process of allocating CPU's is as follows: at any time when a JOB enters or leaves memory, or when the RJQ changes for any reason, the RJQ is examined. If CPU(s) are IDLE (i.e. not being used for ANY job), then the first job(s) on the RJQ are given control of the idle CPU(s). If PREEMPTION is allowed (described below), a job which would appear earlier on the RJQ than a job which currently has a CPU is allowed to seize control of the CPU from the other job, which is then returned to the RJQ. A table is also kept of the JOB(s) currently in control of CPU(s). Whenever a JOB is first given control of a CPU, it keeps control of it for AT MOST the length of time until it must perform input/output (which can be calculated from the IOIN parameter, as described in the IOIN options).

When a JOB requires I/O, it must then compete with other jobs for the use of I/O CHANNEL(s). If it cannot obtain the desired CHANNEL immediately, it must WAIT until it becomes available.

When a JOB obtains control of a CHANNEL, it keeps control of it for the length of time calculated from the IOIN parameter. As soon as its I/O operation is complete , it releases the channel so that other JOBs may use it, and then reenters the RJQ so that it may compete for use of the CPU(s) again. If PREEMPTION is allowed, it may immediately PREEMPT a JOB=CURRENTLY HAVING CONTROL OF A CPU, if it ranks earlier by whatever ordering scheme is being used.

When a job first enters the RJQ, it is allowed a total time from the T=parameter. Whenever a JOB uses the CPU, it is charged the number of milliseconds used, and depending on the option, it may be also charged for use of a CHANNEL.

A. EXECUTION CHARGING METHOD

The following option determines whether a JOB is charged for the use of CPU only or for both CPU and CHANNEL.

&ECCH = 0 The user is charged only for CPU usage. Thus, whenever the JOB is given control of the CPU, the time until the next I/O request is determined, and an interrupt scheduled to occur at that time. However, if the time remaining to be used for the job <= time until I/O, it is used instead, and the job terminated rather than performing I/O at that time.

&ECCH = 1 In addition to the above charging, there is a charge for the use of a CHANNEL, and the I/O duration is then:
time until I/O done = MIN(time left,I/O duration).
A job never gets to use anything if it has 0 milliseconds left

B. EXECUTION ORDERING METHODS

These methods of course are similiar to the JO options for deciding which JOB(s) are to be initiated. For further explanation on similiar ones, refer back to CS411FP1-07. Note that one difference is that the JOB(s) are generally serviced in the order they are on the list, without having to worrying about whether they fit in memory or not (naturally, since they already ARE in memory).

EO1 (FIFO) - the JOBs are ordered on the list according to their arrival there. In this case, they are order according to the FIRST time they entered the RJQ, i.e., just after they were INITIATED, and thus this order does not change. This method favors those jobs which were loaded earliest, and thus tends to minimize the longest time a job must remain in memory.

EO2 STRAIGHT PRIORITY - JOBS are ordered from highest priority to lowest(lowest PRIO=to highest values).

EO3 CATEGORY PROGRESSION - ordered by CATEGORY numbers, this of course favors the lowest numbered categories.

EO4 LENGTH OF TIME REQUIRED BY JOB, either:

EO4S (SJF - SHORTEST JOB FIRST), or
EO4L (LJF - LONGEST JOB FIRST).

Note EO4S tends to expedite the short jobs, and thus move jobs in and out faster, while EO4L tends to minimize the longest time anything ever has to remain in memory.

EO5 MEMORY REQUIRED BY JOB

EO5S (SMF - SMALLEST MEMORY FIRST)
EO5L (LMF - LARGEST MEMORY FIRST)

Note that of these two, EO5L is usually desirable since it will attempt to get rid of large-memory jobs quicker.

EO6 INPUT/OUTPUT REQUIREMENT

These orderings are computed according to IOIN and IORL from JOB cards, and remain constant, regardless of what JOB(s) actually do.

EO6I I/O BOUND JOBS FIRST
EO6C CPU BOUND JOBS FIRST

Of these EO6I is usually desirable, since it allows JOBs performing much I/O to get what little CPU they need, then let others use CPU.

EO7 STATIC ORDERING COMPUTATION

An order number can be calculated for each job at the time it is initiated, and that number used to determine its relative priority for using a CPU. Note that the relative priority of two jobs does NOT change while the two jobs are in memory. Note that all options EO1 - EO6 are examples of this type of scheme.

EO8 DYNAMIC ORDERING COMPUTATION

In this case, the relative ordering of jobs is obtained dynamically and can thus change during execution, according to whatever criteria are desired. The computation of order numbers is typically done either:

EO8F at fixed intervals, such as every 2 milliseconds. All jobs in memory have order numbers computed for them.
EO8V at variable intervals, or whenever a decision must be made to give a job control of a CPU.

The dynamic ordering methods thus include the static ones. They involve more overhead, but can also give improved performance, since they can respond better to the changing state of the system.

A number of typical dynamic ordering methods follow, including some counterparts of the previous static ordering ones.

EO81 FIFO - jobs are ordered not according to FIRST time jobs entered RJQ, but last, i.e., the RJQ is indeed a queue, and a job only enters it at the end, such as when it finishes doing I/O, it rejoins the queue at the end, rather than in possibly the middle. Compared to EO1, this method is somewhat more 'fair' to all jobs in memory, rather than giving the earlier ones more advantage.

EO84 LENGTH OF TIME FOR JOB
These methods compare actual times, rather than the T= form JOB cards.

EO84S SET - SHORTEST ELAPSED TIME - order jobs according to the amount of time a job has had control of CPU, and give CPU to JOB(s) having lesser time. This is very 'fair'.

EO84R SRT - SHORTEST REMAINING TIME - order jobs according to the time estimated to be remaining (i.e., T= - elapsed), and give CPU to jobs having least time left. This favors getting jobs out of memory quickly, but may result in long-running jobs sitting there forever.

EO86 I/O REQUIREMENTS
For each job, maintain totals of the amount of time used for CPU and I/O (necessary only if I/O activity is non-constant, as in random-generation systems). Use this ratio to order.

EO86I favors I/O bound jobs.
EO86C favors CPU bound jobs.

EO86I is preferred, since it utilizes CPU and CHANNELS better.

E090 TIME SLICING

This family of methods attempts to allow short requests to finish quickly, without having to know what the actual time needed is before the job is run. It also attempts to allocate CPU time to ALL of the jobs in memory in a 'fair' way. It is most often used for interactive and Time-Sharing systems, where many users all wish to obtain very fast or short computations, but occasionaly have longer requests.

E090RR ROUND-ROBIN - when a job reaches the head of the RJQ, it is given control of a CPU for a maximum of &TIMSLIC millisec, or until it waits for I/O. In either case, the next job in the list is given control of the CPU, and the job just removed from control of it enters the RJQ at the end. When a job completes, it is removed from memory. Each slice of time is called a QUANTUM. Note that a job requiring only 1 QUANTUM will enter, execute, and be completed quickly, while those needing more must stay longer. This is very commonly used, paticularly in Time-Sharing operating systems, is simple, but can be catastrophically bad if many jobs want to use the CPU, and if the jobs are only partially core-resident.

Many other variations exist on this scheme.

C. PREEMPTION

When a job first enters the RJQ, or when it completes I/O, and thus becomes eligible again for use of a CPU, it enters the RJQ in the appropriate place. If the job enters the RJQ in the earliest place, either one of two actions may occur:

1. The job may wait until a currently executing job releases a CPU by either requesting I/O or terminating execution entirely.

2. If the job has effective priority or order such that it would rank ahead of a currently executing job (if that job were in the RJQ), the new job may seize control of the CPU, i.e., PEEEMPT it. In this case, the old job is removed from the CPU, and placed back in the RJQ, with the following information computed: elapsed time in CPU must be subtracted from both the job time remaining, and the time interval until the next I/O for it, and the fact that this job has been preempted must be noted, so that a new interval until I/O is NOT computed when job regains control of the CPU later.

Two options are thus possible:

&EOP = 0 NO PREEMPTION

&EOP = 1 PREEMPTION ALLOWED (more complicated to program, but usually better for utilizing CPU and CHANNELS, since it avoids having CPU BOUND programs tying up CPU for long periods of time).

VI. INPUT/OUTPUT SCHEDULING

Normally, when a job receives control of a CPU on a REAL computer system, it keeps control of that CPU until it either requests I/O and must WAIT, or a higher-priority job's I/O completes and is allowed to preempt it.

Since the jobs in a simulated system are not actually performing I/O, whenever a job is scheduled to receive control of a CPU by OS/411, the time until an I/O request occurs must be COMPUTED, and an I/O request scheduled to occur at that time. In general, the interval until the job relinquishes the CPU is always the MINIMUM of:

an I/O INTERVAL (from JOB card IOIN=, or computed from it).
JOB time remaining.
QUANTUM (if using Time-Slicing methods).

When the job actually gives up the CPU, it either TERMINATES and is removed from memory (if the time remaining = 0), or else makes an I/O REQUEST, in which case it must obtain a channel for this action.

When the I/O REQUEST occurs, the job either uses a channel right away, if it is free. or else must enter a queue if requests waiting for a desired channel. Sooner or later, it will be able to use the channel, at which time an I/O INTERRUPT must be scheduled, i.e., this is the time at which the I/O is completed. In general, no PREEMPTION is ever allowed for use of a CHANNEL, as it would be disastrous to interrupt the use of a CHANNEL.

When an I/O INTERRUPT occurs, the I/O operation is completed, the job releases the CHANNEL being used, and it returns to the RJQ to once more compete with other jobs for use of a CPU.

A. I/O INTERVAL CALCULATIONS

The length of time until a job gives up the CPU is always found just before the CPU is given to the job, as follows:

IOIC — Constant interval from SYSGEN. For all jobs, &IOINTVL gives the number of milliseconds until an I/O REQUEST.

IOIJ — Fixed interval from JOB card. For EACH job, the time until the next I/O REQUEST is taken from the IOIN= parameter on the JOB card.

IOIR — Random Intervals, using Job Card information. For example:

IOIRU — UNIFORM - the actual interval is computed from a UNIFORM distribution, using the IOIN= parameter as the MEAN, i.e., any value from 0 to 2*IOIN is equally likely.

B. I/O REQUEST QUEUE SCHEDULING

When I/O request occur for channels currently in use, the jobs making the requests must enter queue(s) to obtain channel use. These queue(s) can of course be ordered in various ways, corresponding to the JO and EO options given previously. However, for this type, usually only several of the possibilities are actually done:

IOO1 FIFO - the first job requesting a channel gets it first.

IOO2 PRIORITY - the highest priority (lowest PRIO=) job waiting for a channel obtains it, even if others arrived earlier.

IOO6I I/O BOUND JOBS are favored over CPU bound jobs, with this calculation made either statically or dynamically.

NOTE: FIFO is the most commonly used, but sometimes PRIORITY is used, especially if there are one or more jobs in the system which absolutely require fast I/O service (usually in Real-Time or Process Control systems also containing batched jobs).

C. I/O REQUEST LENGTH CALCULATIONS

Whenever a JOB gains control of a CHANNEL, the time it will use it must be calculated, and an I/O INTERRUPT scheduled to occur at that time, when the job will give up control of the channel. This length of time using the channel may be found in ways analogous to the IOR options for determining the intervals between requests:

IORC Constant interval from SYSGEN. For ALL jobs, &IOREQL gives the number of milliseconds a job uses a channel.

IORJ Fixed interval from JOB card. For each job, the length of tim a channel is used is constant, and taken from the IORL= parameter on the JOB card.

IORR Random lengths, using JOB card information. For example:

IORRU UNIFORM - the actual length is computed from a UNIFORM distribution, using IORL= from JOB card as the MEAN. Thus, any value from 0 to 2*IORL is equally likely.

VII. REPORTS

At various times during a simulation run, various reports may be printed. In some cases, these may be useful for debugging, and others are most useful for obtaining statistics for comparison of different versions of OS/411. Unlike most of the other option groups, the ones in this set are NOT mutually exclusive.

R1 SYSGEN report. This report is printed 1 time only at the beginning of an entire run, even if multiple batches of jobs are run. It describes which options are being used, and also prints the values of various SET variables used to generate the particular version of OS/411 being used.

R2 CURRENT STATUS REPORT. Every &RDRINTV milliseconds, this report is printed, for the prupose of showing the overall status of the system. It might include:

- R2T The current time (0, 1*&RDRINTV, 2*&RDRINTV, etc).
- R2N The current number of jobs waiting on disk for execution and the current number in memory.
- R2SM The current status of each job in memory, with following codes (and other information if applicable):
  - R (Ready) (ready to execute, but does not have CPU).
  - C (Cpu) (executing now. add number of CPU used in multiple-CPU systems).
  - W (Wait) (waiting for channel to become free).
  - I (I/O) (performing I/O. add channel number being used in multiple channel systems).
- R2MJ The current memory status for each job, i.e., which blocks of memory are currently allocated to the jobs.
- R2MM Memory Map - a map of memory, showing which jobs are located in which areas.
- R2MT The total amounts of memory in use and unused.

R3 JOB READ REPORT - printed whenever a JOB card is read, and JOB thus enters system. Is essentially an ECHO of JOB card, preceded by the time it enters system.

R4 JOB INITIATION REPORT - printed whenever a job is selected to be loaded into memory. Gives time and the name of the job from the JOB card. An additional feature might be:

- R4M memory locations in which job is loaded.

****NOTE**** the memory allocation algorithms may be affected by the requirement of reports R2MJ, R2MM, R2MT, and R4M: i.e., it may be the case that it is NOT actually necessary to implement the algorithms in MAA, MAC, and MAM, IF it is NOT required that memory maps be printed. Thus, it may only be necessary to know it there is enough space to load a job, NOT where it actually is.

R5 TERMINATION REPORT - whenever a job has consumed all of its allotted time, it is terminated (removed from memory). This report should show: 0
Job name
Current time.
JOB turnaround time (= current time - time job was read into the system).
Memory residence time (= current time - time when the JOB first entered the AJL).

Depending on the version of OS/411 used, these might be useful:

R5R (especially for Random-type systems) - total time spent using a CPU, total I/O time, total time WAITING for CPU, and total time WAITING for a channel, ratio of CPU time used to total time (showing CPU-I/O BOUNDEDNESS TYPE).

R6 DISPATCH REPORT (each time a job gets control of a CPU it is said to be DISPATCHED). This is done whenever a job gets control of a CPU, and displays time, jobname, and CPU number for multiple-CPU systems. Might show length of time the job is to be given control of the CPU.

R7 I/O REQUEST REPORT - gives time, jobname whenever an executing job stops to do I/O.

R8 CHANNEL ALLOCATION REPORT - done every time a job actually receives control of a channel, and gives time, jobname, and channel number. It might optionally show the length of time the job will use the channel.

R9 BATCH REPORT - this report occurs at the end of a complete BATCH of jobs, when all jobs in the BATCH have been INITIATED, EXECUTED, and terminated,and the system is completely empty. 0
At this time, statistics can be printed about the BATCH. Needless to say, the appropriate counters must have been kept during the simulation.

The following items are to be printed:

Total time for entire BATCH (TOTAL = time last job terminated).
Total number of jobs run (called NJOBS).

For each CPU or CHANNEL in the system:
Total time during which CPU (CHANNEL) was being used to EXECUTE (DO I/O). (called TIMEUSED).
Percentage of time each CPU (CHANNEL) used: 100*TIMEUSED/TOTAL .

Degree of Multiprogramming: gives the average number of jobs in memory over entire batch. Needs an accumulator (MULTI) and a job counter (#INCORE), both initially = 0. Each time a job is initiated or terminated, set:
MULTI = MULTI + #INCORE * (interval since last INIT or TERM).
and increment/decrement #INCORE as is appropriate.
Finally, at the end of a batch, MULTI/TOTAL gives the average number of jobs in memory (usually >1.0 under multiprogramming).

The following other report items might be printed at the end of a batch under some circumstances:

R9TU Average TURNAROUND time for all jobs in a batch.

This option could usefully be generalized to show turnaround by groups of jobs (for example by category CAT=, or by splitting jobs into several groups according to T=, SP= etc). The reason for this is to determine just which types of jobs are being favored, and if so, how much.

RDBG DEBUGGING REPORTS - these may be added as desired. Things which might be useful debug output to build in are:

RDWJ A dump of the list of jobs waitng on disk.

RDMJ A DUMP OF THE LIST IF JOBS IN MEMORY (AJL plus others).

RDST a dump of all the important system variables, such as the 'clock' or current time, the status of CPU(s) and CHANNEL(s), important flags, etc, etc.

Note these reports are not meant to be pretty, but they are probably the MOST IMPORTANT items for getting the program running.

Note that an excellent technique is to embed such debugging reports in a program from the beginning, allowing them to be removed in one of two ways:

ASSEMBLY TIME: if absolute sure that a given section of code is correct, debug output can be temporarily canceled using SET variables. When the inevitable bugs then creep out of hiding, they can be caught by changing the SET variables again, i.e., XSNAP-XSET process.

EXECUTION TIME: suppose that a program runs a reasonable length of time, appears to be running correctly, then bombs in a given situation. If debug output is produced for the earlier part of the run, record limits are exceeded. Therefore a convenient way is needed to determine whether debug output is needed during execution. One such way is to:

1) Read counters/flags, etc into a global area of the program, from the $$DEBUG card.
2) Use XSNAP IF= option to test these counters/flags, etc and either print debug output or skip it.

In particular, the above approach is of considerable merit when OBJECT DECKS are being used, since this way, it is possible to get an object deck which is loaded with debug output code, but never actually prints such output unless requested from input cards.

Note that $$DEBUG cards can then be inserted in the input stream in position just BEFORE trouble occurs, and removed when debugged.

VIII. GENERAL IMPLEMENTATION METHODS

This section presents general methods helpful not only for this project, but for any other assembler language program of more than trivial size. Some of these methods have analogues in high-level languages. These hints may save much time and effort.

A. MODULAR PROGRAMMING

The program should be constructed of a number of different CSECTs. For OS/411, the appropriate number is approximately 6-12 CSECTS. At least SOME of them can be assembled and debugged fairly quickly, object decks obtained for them, and included with the other modules yet to be debugged.

Inside each CSECT there may be INTERNAL SUBROUTINES where useful, rather than complex flag testing/setting code.

B. PROGRAM DESIGN, FLOWCHARTING, DATA STRUCTURING

The program should be fairly well laid out before MUCH code is written for it. In some cases, it is definitely possible to write a complete section of code without having designed the next one. This can be a good procedure, since it allows the programmer to be running and debugging one section while writing the next. However, it is a good idea to be SURE that the current section's design does NOT depend very mcuh on what the next one does, since this will require unnecessary changes: AVOID CONTINUALLY CHANGING CODE AROUND.

Flowcharting is useful because it helps eliminate unnecesaary or redundant code, aids determinging the dependence/independence of code segments, and helps in the designing of good data structures. Heavy use should be made of various forms of subroutines and macros. Big assembler programs are totally unreasonable without them.

C. NAMING CONVENTIONS

Very early in the design process (and definitely BEFORE any code is written), the programer should make up some simple and consistent conventions for names. Typical rules are:

1. Every register EQU begins with R, and symbolic registers are used EVERYWHERE.

2. EAch CSECT and DSECT will have a name such that every label in the section begins with the first 1, 2, or 3 (choice) letters of that section.

3. Important symbols used across many CSECTS might be assigned to begin with particular characters. Of these, $ is a often chosen to mark such symbols.

Why bother with rules such as are given on the previous page? Briefly, it is because such rules cost little effort, but save much time and trouble. Among other things:

1. It is EXTREMELY annoying to lose a run, with 1 STATEMENT FLAGGED because a statement label is duplicated, very easy to do in a large program if no rules are followed.

2. When looking for an error or at a listing for any reason, the NAME of something shows immediately where it is located, what section it belongs to, and perhaps what its purpose is. This is particularly useful for dsect symbols belonging to specific control blocks.

3. The name methods given cause related symbols to appear close together in the CROSS-REFERENCE listing, which also saves time.

D. SYMBOLIC CODING, DUMMY SECTIONS

DSECTS are an absolute necessity. Flag values and switches are also easier set and tested if they have mnemonic EQU values rather than using actual values. For example:

```
$ACTIVE   EQU  X'01'              (AJLSTATS) => job using CPU
          .....
          MVI  AJLSTATS,$ACTIVE   show this job now active
          .....
          CLI  AJLSTATS,$ACTIVE   is this job active?
```

The above is much clearer than MVI AJLSTATS,X'01', etc. Even MORE IMPORTANT is the fact that it is infinitely safer and less error-prone.

A useful method is to have 1 different DSECT for each distinct type of control block, list element, table, etc, plus perhaps 1 DSECT for a global control table. A TYPICAL METHOD IS THEN TO LOAD AN address constant of a table CSECT into a specified register very early in the main program, connect it to the DSECT with a USING, then NEVER modify that register during the program. Registers 10, 11, 12 seem to be the most favored for this type of use.

IX. SPECIFIC IMPLEMENTATION TECHNIQUES

This section discusses the specific project, covering specific modules, data structures, and techniques which may be of use in writing this project. ********** NOTE ********** THIS INFORMATION IS ONLY TO HELP GETTING STARTED. IT IS NOT REQUIRED THAT THE PROJECT BE STRUCTURED IN THE WAY DESCRIBED HERE, ALTHOUGH THE PROJECT MUST BE CAPABLE OF THE REQUIRED ACTIONS, REGARDLESS OF HOW IT IS WRITTEN.

Please note that the data structures and modules described are designed in such a way as to handle any of the combinations of options given previously. As a result, they may be extremely inefficient for a particular set of options. In this case, you should throw out code or variables which are not at all needed. However, be careful about departing too far from the methods shown in the direction of INCREASED COMPLEXITY, because you will NOT get the project finished if you go out of your way to make it extra complicated.

A. PROGRAM MODULES

The required functions must be performed by sections of OS/411, some of which may be implemented as CSECTs or perhaps as internal subroutines of CSECTs. The functions described will are not generally suitable to be written as MACROS, although it may be useful to write macros wherever necessary for support of their functions.

1. JOB CARD SCANNER (also called READER-INTERPRETER) (abbrev RDR)

When this module is called, it:

a) Determines the type of command on the next input card (if any), and then branches to code sections to handle the individual types of cards possible.

b) If command was a $$JOB, it decodes the various parameters, obtains a free UJL element (block for job waiting on disk), places it it in the UJL (Un-initiated Job List) in the appropriate position.

Lookup of parameter options should be done using clean, table-driven code, i.e., for parameters, scan to an = sign, pad parameter name to a standard length, then use a CLC - BXLE loop to find the position of the parameter in a parameter table, which may also contain flags or jump codes to control further processing of the parameter values.

When finished processing a card, this module may request that it be called at some time in the future, and may also set various global flags.

This module essentially implements the following option groups: RDR, JO.

2. JOB INITIATOR - (JBINIT)

This module is called whenever it MIGHT be possible to move JOB(s) from the queue of jobs waiting on disk (UJL) to the queue ready in memory (AJL). This possiblity occurs normally immediately after:

a) A new JOB has just arrived and been placed in the UJL by RDR.

b) A JOB in the AJL has just been completed and removed by the JOBTERM routine.

In either case, it may be possible to INITIATE a job which was not previously present or could not fit in memory.

It removes the selected UJL element from the UJL, obtains an empty AJL element from the list of such elements, fills it in with required times and values from the UJL, marks the element READY, then either adds the AJL element to the AJL immediately, or takes care of requesting it to be added at a later time (i.e., NOW + &INITDEL).

The initiator then loops through the above until it can no longer initiate any more jobs, for any reason whatsoever.

This module implements the following groups: JSR, MA, and probably needs EO also.

JBINIT probably must request that DISP be called whenever there may be a change in the AJL.

3. DISPATCHER (DISP)

Whenever a CPU becomes available (either because of an I/O request or a termination), this module determines which, if any, job should be assigned the free CPU. If PREEMPTION is allowed, this module should determine whether to switch a READY job for an EXECUTING one, which can occur either when a new job is finished being loaded, or when an old one completes I/O and is thus READY again.

It may call IOILENG to determine how long a job being DISPATCHED (given control of the CPU for some time) will execute until it makes an I/O request.

The DISPATCHER probably manipulates various clock values in the AJL elements, decrements time-remaing counters, etc, and generally performs bookkeeping operations needed to generate statistics.

If PREEMPTION is allowed, the DISPATCHER must take care of it, and modify clock elements accordingly, so that a preempted job is treated fairly, does not lose time, and generates an I/O request after the appropriate amount of time.

This module contains code for option group : EO.

4. I/O INTERVAL LENGTH ROUTINE (IOILENG)

This module determines how long a job will be allowed to execute, before being halted for an I/O request. This might reasonably be an internal subroutine of DISPATCHER.

This module implements option group: IOI, and may be involved with &EOP variable.

5. CHANNEL REQUEST PROCESSOR (CRPROC)

This module is called whenever a job is to relinquish control of a CPU and attempt to perform I/O. It removes the job from control of the CPU, and markes the CPU idle, also requesting that the DISPATCHER be called immediately to try to schedule a job into the CPU. If the required CHANNEL is currently idle, CHANALC can be called to allocate the channel to the job (and the job marked as in I/O status). If the channel is not available, it must enter a queue of jobs waiting for channels, (IOR), and wait here until the needed channel becomes free. In this case, the job is marked as in WAIT state.

This module typically implements the option group : IOO.

6. CHANNEL ALLOCATOR (CHANALC)

This routine is supplied with a job and a channel, and essentially gives the channel to the job, determines how long it will be until the I/O interrupt occurs which ends the I/O, and also takes are of checking against time remaining (if user being charged for channel time also,i.e. if the SYSGEN specified &ECCH = 1) .

CHANALC must make sure that CINTR is called whenever the I/O interrupt is to occur. (possibly to call JOBTERM if there is no more 0 time left and channel time is being charged).

This module implements: IOR options.

7. CHANNEL INTERRUPT HANDLER (CINTR)

This module is called whenever a job finishes with a channel. The channel is freed, marked idle, and if any other job is waiting for that channel, it calls CHANALC to give the channel to the job needing it.

The job's status is changed from I/O to READY, it is placed back in the AJL, and a request made that the DISPATCHER be called, so that the job may effectively compete for a CPU again.

8. JOB TERMINATOR (JOBTERM)

This module is called when a job finally consumes the time allotted to it on the JOB card. It removes the job from memory, returns any control elements to free list of such, and also notes that the JBINIT routine be called, since it may now be possible to initiate another job.

9. EVENT EVNQUEUER (EVNQ)

This module (or possibly just a macro) is called whenever it is necessary to schedule some type of event to occur at a certain time. It inserts (or constructs and inserts) an EQ node into the EQ list according to time of event.

10. REPORT MODULE (RPORT)

This module may be called to produce the various types of reports possible at various times. In some cases, the R options may be included as sections of code in other routines (like R3 in RDR, R4 in JBINIT). This module might be written in FORTRAN, at least partially.

11. MAIN PROGRAM

This module first prints the SYSGEN report R1, if required. It then consists of two nested loops.

The outer loop is executed once for each BATCH of jobs, and is made of the following steps: INITIALIZATION (reset flags, counters, put all list nodes into the respective free lists, and also request that the RDR be called at time 0, set clock GNOW = 0, etc); INNER LOOP (the actual simulation); and COMPLETION (print report R9, if exists). If it is discovered at the end of this loop that a $$QUIT card has been found (or end-file), the loop ends and the entire program terminates.

The inner loop is executed one time for each EVENT which occurs, and thus traces the system actions at those times during which the system actually changes status, ignoring times in between. Each loop performs the following actions:

a) POP's the first node from the Event Queue (EQ), removes the data item(s) from the node, and returns the node to the free list of such nodes.

b) Sets the global clock GNOW = time from the node, i.e., the current time is now whatever time the event is to occur.

c) Uses a code value from the node to determine which of several routines (RDR, JBINIT, etc) to call to do required action.

d) Loops through a) until the EQ is empty, i.e., nothing more is to occur.

THE ABOVE LOOP IS AN EXCELLENT CANDIDATE FOR EMBEDDED DUMPING CODE WHICH CAN BE TURNED ON OR OFF ACCORDING TO VARIABLE SWITCHES.

B. DATA STRUCTURES AND TABLES

This program is essentially a huge list-processing program, which of course makes a set of list-processing macros very useful. It is also useful to write simple macros, then combine calls to them in more powerful and easier-to-use macros.

There are two basically different ways of allocating the storage for this program, static, and dynamic.

The static method is the easiest to program. Whenever the system is initialized, all of the nodes in an active list for a specific kind of node can be pushed onto the free list for that kind of node, and thus they will be ready for the next BATCH (although they may not be in exactly the same ORDER as they were before). Since there are upper limits on the number of any type of node, this method should work and be easy to do.

The dynamic method implies maintaining a single FREELIST of all unused memory, then allocating a node of given size whenever anyone needs one, then returning the node when finished with it. This may use less total space, but is slower and more complex to program.

In addition to the list elements, the following are tables and variables which might be useful/necessary. AS USUAL, THESE ARE ONLY HINTS ON POSSIBLE WAYS TO DO THINGS. ACTUAL IMPLEMENTATION METHODS ARE YOUR CHOICE. THE FOLLOWING ARE JUST TO HELP YOU GET STARTED.

1) GLOBAL CONTROL TABLE (GCTB)

This CSECT effectively acts like FORTRAN COMMON, and is used to hold variables and useful constants to be addressed by any section of the entire OS/411, and lets them communicate easily with each other. Typically, the main program would do the following:

```
L     RGCTB,=V(GCTB)              address of the csect
USING GCTB,RGCTB                  note pointer
```

Then each other csect would include the USING stmt, and no one would ever modify register RGCTB, so that it would always point at GCTB and make its variables available.

Among the typical items contained in this common CSECT would be:

USEFUL CONSTANTS (such as big blocks of zeroes, blanks, Edit patterns for conversion routines, Translate tables if needed several places, useful Mask words, etc, etc).

HEADER CELLS and FREELIST HEADER CELLS for all global lists, since several different routines may modify/inspect the same lists. It may also be desirable to include the actual list nodes themselves, near the end of the CSECT, in order to get all important items in one area.

In addition to the above, the following are GLOBAL VARIABLES which might be included in the GCTB :

```
GNOW     DS    F    (clock) current time, initialized to 0
G#INCORE DS    F    number of jobs currently in memory, init =0
GLASTIT  DS    F    time last initiation or termination occurred, useful
GFLAGS   DS    C    flag byte, for such items as whether $$CLEAR found,
                    $$QUIT, etc.
GDEBUGS  DS    C    debug control byte: could be used with XSNAP IF= to
                    control debug output from input $$DEBUG cards
GMULTIP  DS    F    multiprogramming statistic (see report R9)
```

HINTS: group important variables so they can be dumped with a single XSNAP. Consider bracketing them with character DC constants so that they are quickly located in a dump.

2) EVENT QUEUE

This is the most important single queue, having header cell EQHD, free list header of empty nodes EQFREE, and of which &EQ# total nodes should be generated (use &EQ# = 100 unless otherwise necessary). A dsect for each node might look like:

```
EQSECT   DSECT
EQLINK   DS    A    link to next node
EQTIME   DS    F    time this event to occur
EQDATA   DS    F    data for event type (might be addr of control block)
EQTYPE   DS    F    address of routine, or code giving index to it
.......... anything else which might be useful.
```

3) UN-INITIATED JOB LIST (UJL)

A UJL element is needed for each job waiting on disk for execution. There might be up to &MAXJOBS of these needed. EAch node must contain all useful information from a $$JOB card, plus anything else desired. A typical dsect might be:

```
UJLNODE  DSECT
UJLLINK  DS    A    addr of next node
UJLORDR  DS    F    order field, compute according to JO option
UJLNAM   DS    CL8  name of job from $$JOB card
UJL#     DS    H    number of job  (showing order in which read)
UJLENTR  DS    F    time at which job was read
.......  appropriate entries for T=,SP=,IOIN=,IORL=,PRIO=,CHAN=,CAT=
```

4) ACTIVE JOB LIST (AJL)

One AJLNODE is used for each job currently in memory, and would typically include all items present in the UJLNODE (or a pointer to a UJLNODE having them). There are a maximum of &JOBLIM3 of these, and a typical setup might be:

```
AJLNODE  DSECT
....... set of areas like those of UJLNODE
AJLLOAD  DS    F    time when job first loaded into memory (initiated)
AJLTLAST DS    F    time when status of job last changed (useful for
                    statistics handling)
AJLTREM  DS    F    time remaining until job is completed: this is
                    decremented, and job terminated when it = 0
AJLTINTR DS    F    time until a job either requests I/O or is scheduled
                    to be terminated.  useful in I/O handling.
AJLC#    DS    A    address of CPU element or channel element showing
                    which one this job is using, if any.
AJLSTATS DS    B    status byte:  shows condition of job: ready, but not
                    executing, executing, waiting for I/O, doing I/O.
```

5) INPUT/OUTPUT QUEUE

Each node represents a request for a channel which cannot be filled until job using it releases it. Maximum of &JOBLIM3-1 of these needed.

```
IORNODE  DSECT
IORLINK  DS    A    addr of next one in list
IORORDER DS    F    possible ordering field, depending on IOO option
IORAJL   DS    A    address of AJLNODE showing job making request
```

6) CHANNEL/ CPU ELEMENTS

Two lists (or tables, depending on convenience) might exist, with one node for each CPU/CHANNEL, mainly used for keeping statistics for usage of each one. Of course, they might just be variables in systems having either 1 CPU or 1 CHANNEL only. Typical dsect for CHANNEL (CPU node would look the same):

```
CHNNODE  DSECT
CHNAJL   DS    A    addr of AJL of job using this device, = 0 if IDLE
CHNTLAST DS    F    last time status of this device changed
CHNUSE   DS    F    total accumulated usage time during run.
```

X. SYSTEM GENERATION OPTIONS LIST

The following lists the various options possible for a SYSGEN of OS/411. This chart may be filled in when the instructor goes over the project requirements in class. Note that all options are listed, with underlines where things must be filled in. Also note that not all options may be applicable, since some are needed only if some particular previous choice was made. The heading numbers correspond to the numbers of sections in this writeup. Space is left for comments.

II.

&MEMSIZE = _____K bytes
CPU___ &NUMCPUS = _____ (only if variable #)
CHN___ &NUMCHNS = _____ (only if variable #)

III.

&MAXJOBS = _____ jobs at most in system
PARM__ (if PARM2, then need values for following)
&TDFT, &SPDFT, &IOINDFT, &IORLDFT, &PRIODFT, &CHANDFT, &CATDFT
_____ _____ _____ _____ _____ _____ _____

IV.

RDR___ &RDRINT = _____ millisec (if RDR1)

JO____

JSR___ &JSRANGE = _____ (if variable)

&MEMSIZE = _____ K bytes (repeated from II)
&MEMOS = _____ K bytes
&MEMPGR = _____ K bytes
&JOBLIM1 = _____ jobs

&JOBLIM2, &JOBLIM3 calculated by SETA arithmetic

MAA__

MAC__

MAM__ &MAMOVE = _____ (if MAC2, etc)

&INITDEL = _____ milliseconds

V.

&ECCH = __ (1 = charge for channel use, 0 = don't)

EO___

&EOP = ___ (1 = allow preemption of CPU)

VI.

IOI___ &IOINTVL = _____ (if IOLC)

IOO___

IOR___ &IOREQL = _____ (if IORC)

VII.

Since reports are not mutually exclusive, just list ones required below.

&RDRINTV = _____ millisec (if report R2 is required)

******** NOTES ********

COMPUTER SCIENCE 411 - MACRO ASSIGNMENT
DUE _____
This writeup: pages 01 - 03.

I. LINKAGE MACROS - QSAVE, QRETURN, QCALL

This set of macros covers the following items: OS/360 linkage conventions, and some basic items of macro programming: referencing macro arguments, obtaining items in sublists, use of local (and some global) set variables, concatenation of set variables and other items, &SYSNDX, and &SYSECT.

A. QSAVE - macro for entering subroutine

Your macro will be similiar to XSAVE or SAVE, and will accept the following operands: REGS=, BASE=, SA=, as follows:

REGS=(reg1,reg2) will store registers reg1-reg2 at appropriate locations , will be specified as numbers, and default to REGS=(14,12) .

BASE=number will set up register number as a program base register. Defaults to BASE=12. VAlues 13,14,15 are illegal, should be flagged by an MNOTE, and then use 12 instead.

SA=value controls save area linkage and save area name.
SA=NO means the subroutine has no save area, so that R13 should not be modified, and no inter-save area linkage created.
SA=name save area linkage will be done, and the address of name will be placed in register 13 as the save area.
SA=* the macro will make up a unique name composed of the first 3 characters of the current CSECT name, followed by a unique number, followed by 'S', and will refer to this when setting up the usual save area linkage. This name will be saved in a GBLC variable for later use by QRETURN. Defaults to SA=* .

The QSAVE macro will also automatically generate an identification field (as described in the LINKAGE WRITEUP). This identification field will use either the label on the QSAVE statement, or will use the current CSECT name, if there is no label on the QSAVE. It will generate the minimum storage needed i.e., it cannot just generate:

```
         B    14(,15)
         DC   X'9',CL9'name'
```

B. QRETURN - return from a subroutine

This macro is similiar to XRETURN or RETURN, and will accept as arguments: REGS=(reg1,reg2), SA=value.

REGS=(reg1,reg2) registers to be restored. Default:REGS=(14,12)

SA=value controls save area linkage and generation of a save area, and works exactly as does XRETURN. Thus, it accepts SA=NO, SA=*, SA=name, and omitted operand.

C. QCALL - call a subroutine

This macro can be written in any of the following forms:

1. label QCALL ,(arg1,arg2,....)

2. label QCALL entryname

3. label QCALL entryname,address list name

4. label QCALL entry name,(arg1,arg2,...)

Version 1 merely generates a list of addresses of each of the arguments, with the last one flagged appropirately to show that fact.

Version 2 loads the address of the entry name (=V(entryname)) into R15, and BALR's there, assuming R1 is already set correctly.

Version 3 is like version 2, but also does a LA to get the address of the address list into register 1 before calling the routine.

Version 4 combines versions 1 and 3, with the address list created inside the macro expansion (like the CALL macro does). (see CNOP instr)

D. SAMPLE EXPANSIONS OF THE MACROS

These expansions are examples: IT IS NOT NECESSARY TO GENERATE THIS EXACT CODE, AS LONG AS THE CODE MEETS THE REQUIREMENTS GIVEN.

```
 MAIN     CSECT
          QSAVE
+         USING *,15 .             TEMPORARY USING
+         B     10(,15) .          BRANCH AROUND IDENT
+         DC    AL1(5),CL5'MAIN'
+         STM   14,12,12(13) .     SAVE REGISTERS
+         LA    12,MAI0002S .      GET SAVE AREA ADDRESS
+         ST    12,8(13) .         POINTER TO NEW SAVE AREA
+         ST    13,4(12) .         POINTER TO OLD SAVE
+         LR    13,12 .            GET IN RIGHT SAVE AREA
+         BALR  12,0  .            SET UP NEW BASE
+         DROP  15 .               DELETE TEMPORARY
+         USING *,12 .             NEW USING
 CALL     QCALL SUBX,ADDRX
+CALL     DS    0H .               DEFINE LABEL
+         LA    1,ADDRX .          ADDRESS OF ADDRESS LIST
+         L     15,=V(SUBX) .      SUBROUTINE ADDRESS
+         BALR  14,15 .            CALL ROUTINE
 GOBACK   QRETURN SA=*
+GOBACK   DS    0H .               DEFINE LABEL
          L     13,4(13) .         RESTORE PREVIOUS SA PTR
          LM    14,12,12(13) .     RESTORE REGS
          BR    14 .               RETURN
+MAI0002S DC    18F'0' .           SAVE AREA
 ADDRX    QCALL ,(MAIN,GOBACK)
+ADDRX    DS    0F .               DEFINE LABEL
+         DC    A(MAIN)
+         DC    X'80',AL3(GOBACK)
```

E. ADDITIONAL FEATURES

If desired, additional features may be added to these macros, which may receive extra credit. Do not however, spend too much time on this assignment, and especilly do not spend a great deal of time adding exotic features unless they really seem useful.

F. WHAT TO HAND IN

Hand in one run with all the macro listings, well-commented, and showing complete macro-expansions and execution of AT LEAST the following program (you may add more if you want, and should do so to make sure all reasonably different cases are tested) :

```
MAINPRG  CSECT
         QSAVE
         QCALL SUBX,ADDRX
         QCALL SUBY
GOBACK   QRETURN SA=*
ADDRX    QCALL ,(MAINPRG,GOBACK)
         LTORG

SUBXCS   CSECT
         ENTRY  SUBX,SUBY
SUBX     QSAVE SA=SUBXSA,BASE=11,REGS=(14,11)
         cnop  2,4                cnop for nastiness
         QCALL SUB1,(SUBX)
         QCALL SUB2
SUBRET   QRETURN SA=SUBXSA,REGS=(14,11)

SUBY     QSAVE SA=NO
         XPRNT =CL50'0*** AT SUBY *****',50
         QRETURN SA=NO
         LTORG

SUB1     CSECT
         QSAVE BASE=13
         QRETURN SA=*

SUB2     CSECT
         QSAVE BASE=15,REGS=(2,12)
         QRETURN SA=*,REGS=(2,12)

         END
```

COMPUTER SCIENCE 411 - MACRO ASSIGNMENT
DUE _____
This assignment: pages 04 - 08.

II. HEXADECIMAL CONVERSION, DUMPING MACROS: QHEXI, QHEXO, QDUMPO

These macros cover the following topics: macro/module linkage, some macro processing techniques, and hexadecimal conversions. The following instructions may be useful: TRT, PACK, TR, UNPK.

A. QHEXI - MACRO TO SCAN AND CONVERT HEXADECIMAL NUMBERS TO BINARY

1. REQUIRED FUNCTION OF MACRO

QHEXI is to function somewhat like XDECI, but for hexadecimal input rather than decimal. It is to be called as follows:

```
label    QHEXI reg,address
```

label	is an optional statement label
reg	is the name or number of a register
address	is an RX-type address, i.e., anything legal in an LA instr.

Calling QHEXI should cause the following actions to be done:

a. The address should be evaluated and used as a scan pointer to some character string in memory. A scan will be made starting at that address, until the first character is found which is a hexadecimal digit (0-9, A-F).

b. Starting at the first hex digit found, a hexadecimal number of 1-8 digits (followed by any character NOT a hex digit) is to be scanned, and converted to a 32-bit binary value, right-justified, and filled on the left with leading zeroes.

c. Register 1 is set to contain the address of the first non-hex digit following the hexadecimal number.

d. The register specified in the macro call is loaded with the value given by the hex number just converted (this may be any register).

e. The condition code must end up being set according to the sign of the number placed into the register (=0 => 0, <0 => 1, >0 => 2).

2. ASSUMPTIONS PERMITTED

The implementation of this macro depends strongly on the what assumptions are made about error condtions, modifications of registers, etc. The following assumptions might be made. Consider their effects on the code to be generated:

a. Assume registers 0,1,14,15 may be completely destroyed by the execution of the macro.

b. Assume that registers 0,14,15 may be changed by the macro, but must be restored to their original values by the macro.

c. Assume that neither register 14 nor 15 is the current base reg, but that registers 0,14,15 must all be restored by the macro if changed.

d. Assume that input hexadecimal numbers definitely contain no more than 8 digits.

e. Assume that errors may exist in the input (i.e., more than 8 digits). Then either set the condition code to 3 to show error, (e1), convert the first 8 hex digits normally (e2).

f. Assume that R13 contains the address of a usable s/360 savearea.

(Always assume the macro may not create code changing regs 2-13).

FOR THIS ASSIGNMENT, THE ASSUMPTIONS FROM ABOVE TO BE USED ARE:

a, e1, and f

3. USEFUL HINTS ON INSTRUCTIONS TO BE USED

a. SCANNING: the TRT instruction can usefully be employed to do fast general scanning. Also, several tricks exist for setting up the 256-byte table needed to use a TRT instr. The following is an example of code which scans until it finds the first instance of any letter fROM A to R in a string:

```
         LA    1,STRING              init address of string
LOOP     TRT   0(256,1),TAB1         scan
         BNZ   *+12                  skip if any found
         LA    1,256(,1)             increment
         B     LOOP                  go back for next
..... at this pt, R1 => 1st letter
..... remaining code
TAB1     DC    256X'00'              define whole table
         ORG   TAB1+C'A'             back up location counter to
                                     position of A in table
         DC    9X'04'                nonzero: stop on AIJ
         ORG   TAB1+C'J'             now to position J
         DC    9X'04'                nonzero: stop on J-R
         ORG   ,                     set loccntr to end again
```

The above table is used to STOP on any letter A - R. The table below stops on any character EXCEPT letters A - R.

```
TAB2     DC    (C'A')X'01'           stop on anything before A
         DC    9X'00'                skip over A-I
         DC    (C'J'-C'I'-1)X'01'    stop on ones between I-J
         DC    9X'00'                skip over J-R.
         DC    (255-C'R')X'01'       stop on everything after R
```

b. HEX CONVERSION - USING TR, PACK

A simple trick exists for obtaining conversion from hexadecimal to binary, as long as the hexadecimal number ahas no more than 14 digits. As an example, the code below can be used to convert hex numbers of up to 8 digits.

Assume first, that the 1-8 digit hexadecimal number has been moved to an 8-byte work area, right-justified (NOT LEFT, which is easier), and filled on the left with character 0's (hexadecimal X'F0'). Thus, this 8-byte field is the hexadecimal number with leading zeroes. The bytes' values then can be: A-F, 0-9, or X'C1'-X'C6', X'F0'-X'F9'. Then:

```
            TR    WORK,TAB3          convert  C1-C6 to FA - FF
            PACK  FULL5(5),WORK(9)   do funny pack

WORK        DS    CL8,C              area with number, right justfd
FULL5       DS    F,C                fullword plus wiped-out byte

TAB3A       DC    X'FAFBFCFDFEFF'    for converting C'A' => X'FA',etc
TAB3        EQU   TAB3-C'A'          put theoretical table origin
            ORG   TAB3+C'0'          to position for character 0
TAB4        DC    C'0123456789'      leave digits alone
```

Note what'S OCCURRING ABOVE: since UNPK generally converts each pair of bytes into one byte, it is likely we would want to use it for a conversion in this direction. Unfortunately, it also reverses the last two nibbles from the source field. But, if we pack 9 bytes into 5, the extra ninth byte is reversed and placed into the extra fifth byte, then each of the other bytes has the first nibble (X'F') removed, and the second nibbles packed together into a 4-byte area, as desired. *NOTE* if the ORG & EQU games above make little sense, try punching the statements, running them with ASSIST, and checking the location values .

B. QHEXO - CONVERT VALUE IN REGISTER TO HEXADECIMAL

1. REQUIRED FUNCTION OF MACRO

This is like XDECO, but for hexadecimal instead of decimal:

```
label    QHEXO register,address
```

label, register, and address are as described above under QHEXI

Calling QHEXO should cause the following to occur:

a. The value in the register should be converted to printable hexadecimal format, and placed in the 8-byte area at the address given.

b. The condition code is unchanged by the execution of this macro.

2. ASSUMPTIONS PERMITTED

a. Assume registers 0,1,14,15 may be completely destroyed.

b.Regs 0,1,14,15 can safely be destroyed, but restored by the end.

c. Assume that the current base register may be ANY register 1-15.

(Always assume that registers 2-13 may not be changed in the code generated by the macro).

FOR THIS ASSIGNMENT, MAKE THE FOLLOWING ASSUMPTIONS FROM ABOVE:

c (consider using this macro in lowest-level routime with br=15)

3. USEFUL HINTS ON INSTRUCTIONS TO BE USED

a. CONVERSION: the output conversion is much easier than the input one. Briefly, consider the effects of storing a register into a fullword in memory, then unpacking 5 (FIVE) bytes from that area into a 9-byte workarea, translating the first 8 bytes of that area, then moving 8 bytes where desired:

Assume the data areas from the previous example, that the value to be converted is in first 4 bytes of FULL5, and that the following has been added immediately after statement labeled TAB4:

```
TAB4A    DC   C'ABCDEF'              convert X'FA' => C'A',etc
```

Then:

```
         UNPK WORK(9),FULL5(5)       convert
         TR   WORK,TAB3              make printable
```

C. QDUMPO - PRINT LABELED REGISTER VALUE IN DECIMAL OR HEX

1. REQUIRED FUNCTION OF MACRO

```
label    QDUMPO   register,message,type
```

label is optional statement label

register is name or number of register to be printed

message is an OPTIONAL message to be used to identify output

type D or H. D => print in decimal, H => hexadecimal.
if omitted, default to D. If anything else used, issue
severity 4 warning message, then default to D anyway.

Calling QDUMPO should request the following actions:

a. If a GBLB set variable named &QDUMPO currently has the value 1, NOTHING IS GENERATED, EXCEPT POSSIBLY label DS 0H .

b. If &QDUMPO has the value 0, then code is generated to convert the desired register to decimal or hexadecimal, and print it out, with an identifying message preceding it.(Add blank carriage control).

c. The identifying message noted in b is either the message operand (which is always enclosed in quotes), or the message:
REGISTER register AT LABEL label

d. Neither registers nor condition code may be changed by the macro expansion, and must be usable under any USING base from 2 to 13.

D. WHAT TO HAND IN

Add PRINT NOGEN'S to delete X-MACROS, and run program with data:

```
          GBLB  &QDUMPO
CSECT     CSECT
          XSAVE
          LA    11,AREA3
LOOP      XREAD CARD+1
          XPRNT CARD,81
          BM    DONE
          QHEXI 2,CARD+1
          BALR  3,0
          QHEXO 2,AREA2
          QHEXO 3,0(,11)
          QHEXI 4,0(,1)
          QHEXO 4,AREA3+10
          XPRNT AREA,AREA$L
          CALL  SUB1
          B     LOOP
DONE      XRETURN SA=*
CARD      DC    CL81'0',20CL100' '
AREA      DC    C'0 VALUES OF REGISTERS 2,3,4: '
AREA2     DC    CL10' '
AREA3     DC    2CL10' '
AREA$L    EQU   *-AREA                  LENGTH
SUB1      CSECT
          XSAVE SA=NO,BR=15    WATCH OUT FOR THIS BASE REGISTER
          QDUMPO 2,'HEX VALUE OF 2',H
          QDUMPO 2,'DEC VALUE OF 2'
          QDUMPO 4,'DEC VALUE OF 4',C'H'
SUB1H     QDUMPO 4
          QDUMPO 1,,H
          L      0,=F'&QDUMPO'
          QDUMPO 0,'VALUE OF &&QDUMPO='
&QDUMPO   SETB   1
          QDUMPO 4,'ERROR ERROR ERROR ERROR'
          XPRNT =CL30' VALUE OF &&QDUMPO=&QDUMPO',30
          XRETURN SA=NO
          LTORG
          END
DATA CARDS AS FOLLOWS (BEGINNING IN COLUMN 1):
00001234        FFFFFFFF
01ABCDEF,,,,,,,,,,,,,,,,,,,,,,,,,1234567
        12345678.<(+×&$*);-/,>:#@'=ABCDEFf
 ONLY 1  #
```

COMPUTER SCIENCE 411 MACRO ASSIGNMENT
DUE _____

III. LINKED LIST MACROS: QDEFL, QHED, QNEXT, QPOP, QPUSH, QSRCH, QSNAP

This assignment covers the following: one-way linked list handling techniques and some macro instruction character-scan and operand methods.

This assignment involves writing and testing a group of macros to handle one-way linked lists, to be used heavily in the final project. Each LIST to be processed is of the following form:

A HEADER WORD (or HEADER CELL, or just HEADER) is a fullword in length, and contains the address of the first NODE in the list. Each NODE is a block of storage, aligned on a fullword boundary, with a total length a multiple of 4, with 3 basic subfields:

LINK: the first fullword of the NODE (offset 0). It contains either the address of the next following NODE, if any exist, or else a fullword binary zero to indicate that no more NODES follow, i.e., that this NODE is the LAST NODE in the LIST.

KEY : 0 or more bytes (most commonly 4 bytes at offset 0, i.e. the second fullword of the NODE). If present, this is used to ORDER a LIST for insertion and retrieval of NODES. In this assignment, any ORDERED LIST is kept in ASCENDING LOGICAL ORDER by the KEY field. For example, a NODE having a key value of AAAA would precede that having KEY BBBB, and so forth.

DATA: 0 or more bytes (most commonly at offset 8), divided into as many subfields as needed and convenient to store the various values associated with the KEY (if it exists).

An example of such a list is an alphabetical symbol table used in an assembler, in which the KEY of each NODE is the symbol itself, and the DATA bytes contain such things as: location counter value, length attribute, section identifier number, and any other useful items like ENTRY/EXTRN/CSECT flags, etc.

Given the setup described above, note that an EMPTY LIST (one containing NO NODES), is just a HEADER CELL containing 0.

The following example illustrates an alphabetical LIST. Each NODE contains a fullword LINK and KEY, plus 8 bytes of DATA, resulting in a NODE length of 16:

LOCATION (HEX)	LINK (HEX)	KEY (CHAR)	DATA ITEMS (CHAR)
001000	00004000	(this is the header cell)	
002000	00000000	CCCC	3RD ITEM
004000	00006000	AAAA	1ST ITEM
006000	00002000	BBBB	2ND ITEM

The macros are to be written to handle the type of list just described. The exact functions and formats are given below. Briefly, QDEFL creates a LIST of empty NODES linked together, QHED creates a HEADER, QNEXT obtains the address of the NEXT NODE in a LIST, given a HEADER or address of a current NODE. QPOP POPS the first NODE from a LIST (i.e., obtains the address of it and modifies the HEADER to remove it form the list). QPUSH is the opposite of QPOP; it adds a NODE to the beginning of a LIST, modifying the HEADER to do so. QSRCH performs a LIST SEARCH AND INSERT, i.e., it searches an ORDERED LIST for the correct position for a NODE according to KEY value, then INSERTS the NODE in that position, altering the LINK of a NODE already there. QSNAP is a debugging macro which DUMPS a list.

The following sections give example prototype statements for each of the macros. NOTE: IN SOME CASES, DEFAULT VALUES ARE GIVEN FOR KEYWORD OPERANDS. THESE ARE SUGGESTED AS BEING CONVENIENT, BUT THEY ARE NOT, REPEAT NOT REQUIRED. YOU MAY SUBSTITUTE OTHER VALUES IF YOU THINK THEY WILL BE MORE CONVENIENT DEFAULTS.

For the examples given, assume that the following EQU'S are coded. YOU DO NOT HAVE TO USE THESE, THEY ARE ONLY INCLUDED AS EXAMPLES OF MNEMONIC AND ERROR-STOPPING WAYS TO DO CERTAIN THINGS.

```
LINK      EQU       0          offset of the LINK field in a NODE
KEYO      EQU       4          common offset to key field.
KEYL      EQU       4          most common KEY length
RWK1      EQU       5          temporary work reg, cannot be reg 0
RWK2      EQU       6          2nd temporary work reg, cannot be 0

NODE1     DSECT
NODELINK  DS        A          link to next node
NODEKEY   DS        F          KEY ordering field
NODEDAT1  DS        CL8        first data item
NODEDAT2  DS        H          second data item
NODELEN   EQU       ((*-NODE1+3)/4)*4   node length, to 4-multiple
```

Unless otherwise specified, the label &LABEL is an optional statement label, to be generated either on or before the first byte of executable code in a macro expansion, or usually on the first byte of data generated. POSSIBLE (BUT NOT NECESSARILY THE BEST) CODE is given for at least one sample of each macro.

A. QDEFL - DEFINE A LIST

```
&LABEL    QDEFL     &NUMBER,&LENGTH,&MESSAGE
```

This macro generates &NUMBER nodes, each of total length &LENGTH bytes (including 4-byte link), linked together, with key/data areas initialized to blanks, and preceded by an optional message.

&NUMBER is a self-defining term of value >0, giving the number of nodes to be generated.
&LENGTH is any absolute expression giving a length for each node. Code generated must allow for rounding this to 4-multiple.
&MESSAGE is a string enclosed in quotes. If omitted, the list only is created, but if present, the message string is generated as a C-type constant, rounded up to a 4-multiple, and placed on a fullword boundary preceding the list. It is used for debugging (i.e., to locate a list in memory in a dump).
&LABEL is generated on or before the first node generated.

```
 LAVS1     QDEFL      2,NODELEN,'LIST OF FREE NODES FOR NODE1'
+          DC    0F'0',CL28'LIST OF FREE NODES FOR NODE1'
+LAVS1     DC    A(*+((NODELEN+3)/4)*4),CL(NODELEN-4)' '
+          DC    A(0),CL(NODELEN-4)' '
```

B. QHED - DEFINE LIST HEADER

```
&LABEL    QHED      &LISTNAM
```

If &LISTNAM is coded, this macro defines a HEADER containing the address of this list, otherwise it is a header cell having a value of 0, i.e., defining an empty list.

For debug purposes, it may be very useful to let this macro generate message '&LABEL LIST HEADER' or similiar thing immediately preceding the header word.

```
 HDR1      QHED      LAVS1
+          DC        0F'0',CL16'HDR1 HEADER'
+HDR1      DC        A(LAVS1)
 LIST1     QHED
+LIST1     DC        A(0)
```

C. QNEXT - GET ADDRESS OF NEXT NODE IN LIST

```
&LABEL    QNEXT     &RND,&ADDR,&RLK=,&END=
```

This macro sets register &RND to the address of the next NODE in a list, given that &ADDR indicates the address of a NODE whose LINK field points to the NEXT NODE. If desired, the value in &RND may be saved into register &RLK before &RND is changed (useful for list search and insert operations). If &END is specified, a test is made and branch taken if there are NO MORE NODES in the list.

The operands for QNEXT are described as follows:

&RND is a register EQU symbol or number, into which will be loaded the address of the next NODE.
&ADDR if specified at all, is an RX-type address of the LINK field which addresses the next NODE. The word at this address is to be loaded into &RND. If omitted entirely, it is to be assumed that &RND contains the address of the LINK already.
&RLK if specified, gives the name or number of a register into which &RND should be saved before it is changed.
&END if specified, gives a statement label, in which case the value newly loaded into &RND is to be tested, and if found = 0, a branch taken to the given statement label.

```
 LOOP     QNEXT     5,RGL=RWK1,END=ENDLOOP
+LOOP     LR        RWK1,5
+         L         5,LINK(,5)
+         LTR       5,5
+         BZ        ENDLOOP
```

D. QPOP - POP FIRST ELEMENT OF LIST

```
&LABEL    QPOP      &RND,&HDR,&END=
```

QPOP sets register &RND to the address of the first NODE in the LIST begun by HEADER at address &HDR, taking branch to &END= if the list is empty. (IT SHOULD BE OBVIOUS TO USE QNEXT AS AN INNER MACRO).

&RND specifies register to be set to address of NODE.
&HDR usually specifies the name or other RX-address of the header cell of a list.
&END if specified, requests code to test the LINK just loaded into &RND, and branch to the label specified if it = 0.

```
 LOOPA    QPOP      R6,HDR1,END=ENDA
+LOOPA    L         R6,HDR1
+         LTR       R6,R6
+         BZ        END
+         MVC       LINK+HDR1(4),LINK(R6)
```

E. QPUSH - PUSH NODE ONTO BEGINNING OF LIST

```
&LABEL    QPUSH     &RND,&HDR
```

&RND contains the address of a NODE, which is pushed onto the list begun at &HDR.

&RND specifies a register name or number.
&HDR is a name of a header cell.

```
 PUSHX    QPUSH     R8,HDR1
+PUSHX    MVC       LINK(4,R8),HDR1
+         ST        R8,HDR1
```

F. QSRCH - LIST SEARCH AND INSERT

```
&LABEL       QSRCH        &RND,&HDR,&RGW=(RWK1,RWK2),&KO=4,&KL=KEYL
```

This macro searches the list begun by header &HDR, which is linked in ascending order by KEY fields, for the correct place to insert the NODE addressed by register &RND. The KEY fields are at offset &KO, and are &KL bytes long. &RGW gives two registers which may be used if needed for temporary work registers without disturbing anything.

&RND	gives address of the NODE to be inserted.
&HDR	is the name of the list HEADER CELL.
&RGW	gives the names/numbers of 2 registers which can safely be used as temporary work registers, and destroyed by the macro.
&KO	gives a number (or EQU symbol) of the offset in bytes from the beginning of the NODE to the KEY field in the NODE.
&KL	is the length (number or EQU value) of the KEY field.

Note that the NODE to be inserted is inserted AFTER any NODES which have the SAME KEY value.

```
 SEARCH   QSRCH     R10,HDR1
+SEARCH   LA        RWK1,HDR1
+QQ0002AA LR        RWK2,RWK1
+         L         RWK1,LINK(,RWK1)
+         LTR       RWK1,RWK1
+         BZ        *+14
+         CLC       4(KEYL,RWK1),4(R10)
+         BNH       QQ0002AA
+         ST        RWK1,LINK(,R10)
+         ST        R10,LINK(,RWK2)
```

G. QSNAP - PRINT CONTENTS OF LIST

```
&LABEL    QSNAP      &HDR,&MSG,&RGW=(RWK1,RWK2),&COUNT=4095,&LEN=20
```

This macro dumps the list beginning at the HEADER &HDR, using XSNAP if desired with a message &MSG printed. The work registers needed are given by &RGW, and up to &COUNT NODES are printed.

&HDR	is name of LIST HEADER.
&MSG	is quoted string used as title for list output.
&RGW	specifies names/numbers of 2 registers which may be erased.
&COUNT	specifies the maximum number of nodes to be printed.
&LEN	is an absolute expression giving node length in bytes.

```
DUMP      QSNAP     HDR1,'LIST 1',COUNT=4,LEN=NODELEN
+DUMP     LA        RWK1,4
+         LA        RWK2,HDR1
+QQ0008AA   QNEXT       RWK2,END=QQ&SYSNDX.BB      (not   expanded)
+         XSNAP     LABEL=&MSG,T=NO,STORAGE=(*0(RWK2),*&LEN.(RWK2))
+         BCT       RWK1,QQ0008AA
+QQ0008BB EQU       *
```

H. OTHER USEFUL MACROS (OPTIONAL) : LREMV, LNMBR

The following macros may be found to be useful, but are not required:

```
&LABEL    LREMV     &RND,&RLK
```

LREMV removes the NODE addressed by &RND from a LIST, assuming that register &RLK addresses the LINK field of the NODE which points to the NODE addressed by &RND, i.e., the value in &RND and the value in the LINK field addressed by &RLK are the same.

```
&LABEL    LNMBR     &HDR,&MSG,&RGW=(RWK1,RWK2)
```

LNMBR counts the number of NODES present in the list headed by &HDR, placing result in first register given by &RGW, and using second one as a work register. If &MSG is specified also, the number is printed with &MSG as a heading.

I. TESTING THE MACROS: WHAT TO HAND IN

Write a program to test your macros as follows:

1. Define a list of 15 empty nodes called FREE list, each with 8-byte KEY and 12-byte DATA areas. Also define two empty lists, LISTA and LISTB. (i.e. these are names of headers).
2. Read in 10 data cards, each of which contains KEY and DATA for a single NODE in columns 1-20. After each card is read, obtain an empty NODE from the FREE list, fill it with the KEY and DATA just read (and you must use a DSECT to refer to these fields at this point), then enter it in LISTA.
3. Dump lists FREE, LISTA, and LISTB.
4. Read in 5 cards, each with a KEY value on columns 1-8. Search LISTA for the same KEY value. If not found, print a message. If found, first remove the node from LISTA, then push it onto beginning of LISTB.
5. POP each node of LISTA, print each as it is obtained, then place the NODE back onto the FREE list, until LISTA is empty.
6. Peform same action as in 5, but for LISTB.
7. Show macro expansions for any other special cases you may wish to display. Demonstrate that they work only if you feel like it.

USE THE FOLLOWING SEQUENCE OF KEY VALUES AS TEST DATA:

```
CCCCCCCC, AAAAAAAA, DDDDDDDD, FFFFFFFF, BBBBBBBB,
ZZZZZZZZ, XXXXXXXX, GGGGGGGG, KKKKKKKK, EEEEEEEE

     (set to be moved from LISTA to LISTB if found)
NOTFOUND, ZZZZZZZZ, AAAAAAAA, AAAAAAAA, KKKKKKKK
```

COMPUTER SCIENCE 411 - TOPICS COVERED, HANDOUTS
WINTER TERM 1972 - MASHEY

The handouts given out are are described in file CS411HN

#	DATE	topics, handouts
--	--/--/72	----------

1 01/06 introduction to course, covering:
prerequisistes (101, 102, 404, or equivalent)
outline of topics to be covered later (macros, operating systems, input/output, large machines, assemblers, loaders, memory organization, JCL, OS/360, etc)
useful macros (some review): XSNAP, XDUMP, XSTOP, XSET, XREAD, XPRNT, XPNCH, XDECI, XDECO.

HANDOUTS: INFOR411 (basic information & what to expect)

2 01/08 (OMITTED)

3 01/11 administrative details; linkage conventions and macros.
Registers on entry to subroutine (1,13,14,15).
Parameter list format; PARM field setup as special case.
Linkage actions expected of subroutine: save area format.
Code for return from subroutine; function return, return code.
XSAVE: RGS, SA, BR, TR operands.
XRETURN: RGS, SA, TR operands. use of XSET with them.

ASSIGNMENT: run programs as described by writeup DUMPSJCL, bring dumps to next class.
READING: ASM LANGUAGE MANUAL: Sections 1,2; 3 (except on External Dummy Sections, CXD, DXD, COM); 4, 5 (except OPSYN, CCW, ICTL, ISEQ, PUNCH, REPRO).
HANDOUTS: LINKAGE (linkage conventions for S/360)
DOCUMENT (documentation techniques)
INST (hints on machine and assembler instructions)
XREAD (XREAD/XPRNT/XPNCH macro descriptions)
XSNAP (XSNAP/XSTOP/XSET macro descriptions)
XSAVE (XSAVE/XRETURN macro descriptions)
DUMPSJCL (simple JCL, sample runs to get dumps)

4 01/13 interpreting output, dumps, debug techniques.
go through entire run of ASGCG, using MSGLEVEL=1, /*LOG.
System log information - times, addresses, completion code.
listing of JCL cards supplied, note manual Messages/Codes.
Assembly listing: External Symbol Dictionary, listing, Cross-Reference, and uses of each. follow linkage code.
OS/360 Loader MAP - addresses.
information in the dump - how to find where error occurred.
COMPLETION CODE, APSW, SA TRACE, REGISTERS, STORAGE (SP 000)

HANDOUTS: CS411AS1 (pages 01-02) - 1st assignment, linkage, decks
DUE IN ONE WEEK
READINGS: ASSEMBLER LANGUAGE: Sections VI to end.
REFERENCE: Computing Surveys 1,4(Dec 1969), 183-196.

5 01/13 complete dump reading
follow step-by-step procedure for locating errors in dump.
system completions (precise and imprecise), user completion, differences between link-editor and loader dumps.
interrupts: 1,2,3,4,5,6. and their meanings.

6 01/18 begin macro-instruction writing.
basic concepts of macros - comparison with FORTRAN subroutine.

HANDOUTS: CS411MC1 (pages 01-03) - MCALL, MSAVE, MRETURN ASSIGNMEN

7 01/18 details on macro statements.
most of macro statement types (with FORTRAN counterparts), various examples. MACRO, PROTOTYPE, LCLx, GBLx, SETx, SET VARIABLES, AIF, AGO, CONCATENATION, SUBSTRING, &SYSECT, &SYSNDX.

8 01/20 completion of S/360 macro language
Operand processing, postional/keyword, sublists, attributes, accessing of suboperands, usage of &SYSLIST. many examples.
Program design (5 minutes worth): importance of good design, flowcharting, debugging techniques.

9 01/25 macro problems, multiple entry, CNOP, external symbols.
answer questions on macros, note problems with k'&SYSECT.
give examples of ENTRY usage, multiple entry points, and the usual ways to use them. go over CNOP and how to use it.
note differences between external symbols, which exist as symbols after assembly, and internal ones, which do not.
note USING across CSECTS, and fact that CSECTS may not be in same order as in assembly.

HANDOUTS: ASPRGTC1 (01 - 08) - macros, internal subroutines, and extenals, and when to use them.
ASBROPS2 (01 - 03) - ASSIST base register assignment
ASREPLGD (01 - 11) - ASSIST REPLACEMENT USER'S GUIDE.
ASSIGNMENT: write base register routine (ASBROPS2), using either method A (worth 5 points) or method B (worth 10 points), due 02/18.

10 01/25 addressbility, USING, DROP, DSECTS
Methods for addressing large data areas using A-type adcons.
Typical setup of 1-2 local bases plus one for global table.
DSECTS: write a simple DSECT, then reference in code, showing that each reference meerely requests a base-displacement to be computed. Show equivalent code written explicitly, and show why DSECTS are better for readability and ease of modification. also show method of computing length of DSECT using NL EQU *-NAME , and how this can be used in DC and other instructions for real ease of change.
note importance of DROP, especially with different registers and over what BROPS2 does.

11 01/27 types of assemblers, large 4-pass assemblers
answer questions about assignments, base registers, etc
assembler classification by number of passes: 2,4,1, etc
Two-pass:
Pass 1: opcodes, location counter, allocate storage, symbol table, literal table, ORG, START, EQU#
Pass 2: produces object code for machine ops, DC's, etc
base register table, listing, PRINT, TITLE
Four-Pass: (differences between Assemblers F and G)
NOTE/POINT operation; phases versus passes.
F1: initialization, hash opcodes into global dictionary
F2(Pass 1): source program scan, build macro dictionaries, get system macros from libraries.
F3(pass 2): macro expansion, use of NOTE/POINT stack
F7(pass 3): location counter, etc.
FI: ESD written out
F8(pass 4): final assembly, TXT cards (object code)
FPP (post processor): XREF
Global Dictionary: opcodes, macro names(NOTE addrs), GBLx
Local Dictionary: 1 for each macro/open code, sequence symbols (NOTE addrs), ordinary symbols, LCLx, symbolic parameters for macros. only in core when needed.
Values: set variables: pass 2; ordinary symbols: pass 3,4.
One-Pass: FORWARD REFERENCE PROBLEM

HANDOUTS: ASMTUT1 tutorial on assemblers
CS411MC1: macro assignment: hex conversions and dumps
ASSIGN: write macros for CS411MC1, due in 3 weeks+day.

12 02/01 TRT usage, types of operating systems, history.
TRT table manipulation (reference to handout CS411MC1 - HEX conversions). using ORG instructions in tables. how TRT works, including code equivalent.
OPERATING SYSTEM TYPES (BY HISTORY)
1) HANDS ON
2) BATCH PROCESSING (uniprogramming)
3) MULTIPROGRAMMING SYSTEMS (with SPOOLing)
4) TIMESHARING SYSTEMS

Other classifications: A) UNIPROCESSING and B) MULTIPROCESSING

HANDOUTS: CS411MC2 (pages 01 - 06) Linked List assignment
ASSIGN: assignment given by CS411MC2, due 02/22.

13 02/01 operating system types, programs, begin hardware.
give comparison table for OS types.
program attributes: non-reusable, serially resusable, reent.
REAL-TIME SYSTEMS.
COMPUTER ARCHITECTURE: CPU(S), PRIMARY MEMORY, CHANNELS, IO.
memory fetch/store; access/writeback/cycle times.
physical word versus logical word. interleaving.

READINGS: POP: pages 5-7, 15-22, 68-83 (IO, etc).

14 02/03 memory protection; input/output devices (DASDS)
this lecture continues preparation for general concepts of operating systems and I/O system operation.
CPU: note Mode bit for supervisor/problem states
types of memory protection:
1) NONE use in HANDSON systems only
2) PROTECT BIT IN WORD cheap, but inconvenient
3) BASE-LIMIT REGISTERS - 1 pair: movable programs but no reentrant code easily.
4) 2 PAIRS OF BOUNDS REGISTERS - rrentrant code OK.
5) LOCK AND KEY (S/360) - in detail
to be continued: under virtual memories
I/O DEVICES: beginning with DASD's
Drums, fixed-head disks, movable head disks, data cells.

HANDOUTS: HARDWAR1 (01 - 04, A) PSU 360/67 configuration+ information on devices.
CS411FP1 (01 - 08) 1st part of final project.
ASSIGN: final project, due 03/09.
READING: read on privileged operations in POP.

15 02/08 I/O devices completed, I/O processing beginning.
miscellaneous questions on current addignments.
Magnetic TApes: physical records vs logical records, blocking factors, tape gaps, parity bits (EVEN, ODD), comparison of tape usage blocked 80-80 or 80-8000.
Printers: bar, drum, train/chain.
I/O channels: control between CPU and CHANNELS
CYCLE STEAL.
Types of channels: selector (burst mode only)
multiplexor (burst and multiplex modes both)
S/360 actions: interrrupts: old and new PSW's, usage.
Masks: masking off interrupts, using 1st 4K of memory.

HANDOUTS: CS411FP2 (01 - 08) 2nd part of final project
CS411FP3 (01 - 08) 3rd part of final project

16 02/08 debugging,machine-level I/O, overview of operating systm
debug technique: XSNAP IF= option, similair methods.
I/O interrupts adn handling.
CHANNEL cicuitry - shared or separate.
CCW's, CAW, CSW, and how they interact. Expand I/O process, note command and data chanining, SCATTER READ, GATHE WRITE.
use of protection keys like main memory.
OPERATING SYSTEM (FINAL PROJECT) overview. - RDR/INT, INIT, DISPATCHER, I/OHADNLING.

NOTE: half-period quiz next time on hardware, general concepts, especially for final project.

NOTE: XREAD and XPRNT will not be used on final project.

17 02/10 simulation concepts and implementation
continuous versus discrete simulations; clocks, event queues.
follow entire process of final project simulation, outlining lists modified and programs whihc manage them, with overall scheduling structure.
options to be done for this project: 1024K bytes memory, CPU: 1, CHANv (3-15). JOB card: PARM1 (simple one), etc through all options desired.
HANDOUT: CS411FP4 (01 - 08) specific implementation on FP.

18 02/15 module management
(final project discussion): implementation of global table csect/dsect.
source program -> translator -> object program.
object modules -> LOADER -> executable program
object modules -> LINK EDITOR-> load module
load module -> fetch -> executable program
object module parts: ESD, TXT, RLD, END, and purposes of each. usage of loader and waht it does.

HANDOUTS: CS411FP5 (01 - 04) FP flowcharts.
MODULES (01 - 09, 19) object/load module management

19 02/15 link editor, load modules, overlays
differences between loader and link editor, object modules and load modules, advantages and disadvantages.
overlays: concepts, trees, commands to set them up. briefly on implementation of them.
options to be used on loader and link editor.

20 02/17 user overview of OS/360 services.
program managment and design, data managment, job management, task management.
program structures: simple, planned overlay, dynamic serial & dynamic parallel.
management of resources: job, task, and data.
data management: types of data sets (SDS, Direct, PDS, IS) and what they are used for.

21 02/22 overlay methods, input/output concepts and record formats
go over OVLY1 and OVLY2 programs, pointing out size reductions possible and the control cards used, note PROC on OVLY2.
I/O concepts: buffer groups, flip-flop buffers.
record formats: F, FB, V, VB, VBS, U and comparisons of LRECL, BLKSIZE, # records/block, efficiency, ease of use.
1 outline of rest of term.

READINGS: first 10 pages in DATA MANAGEMENT SERVICES
ASSIGN: run QSAM and EXCP files for next time.

22 02/22 OS/360 macros: all except data management

two types: with SVC calls (regs 0, 1, 15 destroyed often) and without (0, 1, 14, 15 usually wiped, need save area).

JOB MANAGEMENT: communication with operator.

- WTL - write to log
- WTO - write to operator
- WTO - ROUTCDE=11 (write to programmer)
- WTOR - write to operator, and get reply

PROGRAM/TASK MANAGEMENT

Program Linkage inside one load module.

- CALL, RETURN, SAVE

Linkage inside load module for overlay modules.

- SEGLD - begin loading a segment
- SEGWT - make sure segment in.

Program Linkage between load modules.

(note responsibility count = # TASKS currently using a load module. if = 0, not needed any more).

- LOAD - bring to memory, CNT = CNT + 1.
- DELETE - CNT = CNT - 1, remove if desired.
- LINK - bring to memoy, CNT = CNT + 1, pass control (CALL between load modules)
- XCTL - CNT (calling module) = CNT - 1, bring called module to memory if needed, its CNT = CNT + 1. pass control to it, no return, BRANCH between modules.

Task Creation and Management

Task is basic resource allocation unit, each has a Task Control Block, and can compete for resources. EAch job has 1 JOB STEP TASK, and 0 or moreSUBTASKS, in tree structure.

- ATTACH - create a task.
- DETACH - destroy task, terminate processing.
- ABEND - abnormal end of task.
- CHAP - change priority of a task.
- EXTRACT - get data from TCB.

Storage Management

EAch job is given REGION, but space is left in it to be used.

- GETMAIN - obtain main storage dynamically, supply length and get back address of area(s).
- FREEMAIN - return storage to the system.

Synchronization

(needed for both TASKs and I/O)

- WAIT for an event to occur
- POST occurrence of event (ECB's)
- ENQ, DEQ (less important to user)

Timing

- TIME - get time and date
- STIMER - set timer, get interrupt, REAL or TASK.
- TTIMER - tests timer, gets amount left.
- hardware types of timers.

Error Handling

- SPIE - get program interrupts.
- STAE - get all ABEND's of any kind

Debug

- SNAP - dump storage and registers.

S/360 Assembler Language
Documentation and Listing Techniques

by John R. Mashey and Andrea Rhodes

Goals of Good Documentation :
1. Aid in designing good programs
2. Aid in debugging programs
3. Make programs clear and understandable once written
4. Make structure of program well-organized

Good documentation is a great aid to producing clear, well-written, and understandable programs, and can save much programming and computing time. Good documentation is especially necessary for programming projects requiring either a long period of time by one programmer, any period of time by more than one programmer, or modifications to any code by anyone other than the original author. Good documentation techniques can be helpful in the following ways:

PROGRAM DESIGN

Many beginning programmers seem to write programs in haphazard and unplanned ways, and often add comments only after the program is running. This method not only leads to poorly-structured programs, but usually results in wasted time, and is not feasible except for relatively trivial problems.

A much better method is to write most of the overall comments with a flow chart first, specifying the structure and conventions of the program, and then writing the program to fit. This usually leads to cleaner-coded, well-structured programs which are produced in less time than those written by most novice programmers.

PROGRAM DEBUGGING

Program debugging is aided by documenting a program before and during its creation, rather than afterward. Many mistakes can be avoided by having programming conventions well-specified before writing the code. The very act of adding a comment to a statement often helps identify errors in the statement, because it forces the programmer to think about the function of the statement. Finally, good documentation is useful if help is required from someone else, since it aids one in understanding the program quickly. (It also makes other people much more willing to look at a program!)

PROGRAM MODIFICATIONS

Clear and complete documentation is absolutely invaluable when a program must be modified, especially if anyone but the original programmer is making the changes. It may be noted that useful programs tend to be modified often.

ASSEMBLY LANGUAGE DOCUMENTATION

The following advantages apply to any computer language. However, they are most important for assembly language, for the following reasons:

1. Assembly language programs typically require many more statements than do high-level language programs for the same task.
2. Assembly language programs are not usually self documenting. Without good documentation, not even the programmer who wrote the code will be able to understand it several months later.
3. Assembly language programs are often very sensitive to minor changes, much more so than higher-level languages.

The remainder of this paper describes a well-documented assembly program, and notes the various techniques which can be used to achieve this result. Briefly, a well-documented program has the following characteristics:

1. The documentation structure mirrors the program structure, and it leads from the general to the specific. Thus, the program begins with a block of comments which describes the overall purpose of the program, and gives some indication of the general structure. Each major section has a block of comments describing it, as does each of the section's subsets.
2. At least 95% of machine-instruction statements have comments.
3. The program is easy to read, and blocked off into logical sections, so that anyone may look at it and understand it easily.
4. Good programs typically have 15-25% of the total statements as comment cards, in addition to the comments on the individual statements.

S/360 ASSEMBLER DOCUMENTATION HINTS--DO'S and DON'TS

DON'T

punch statements in random columns. This makes a program very unreadable. Use a drum card, and if you do not know how, ask your assistant. The following is a defacto standard for S/360 Assembler statements:

Col. 1 : LABELS
Col. 10: OPERATION CODES
Col. 16: OPERAND FIELD
Col. 36: COMMENTS (col. 40 is preferred by some people)
Col. 72: CONTINUATION COLUMN
Col. 73-80: SEQUENCE NUMBERS (very useful--ask your assistant how to sequence a deck if you are unsure)

This layout can be obtained by the use of the following drum card:
Cols. 1,10,16,36,73: punch '1' (gives tab stops at these cols.)
Col. 72: punch '-' (skips col. 72 automatically, unless AUTO DUP/SKIP is off)
All other columns: punch 'A'

If for some reason these columns are not wanted, a standard set should be decided upon, and then held to completely.

DON'T

Place a comment card before every statement. This bad habit makes programs absolutely unreadable. Embedded comments should be used to block programs into logical sections, not to explain the function of individual statements.

DON'T

bury code with too many interspersed comments. If so many comments are necessary, place them in blocks ahead of the program segments and not in the middle.

DO

put a comment on nearly every machine instruction. Comments are also helpful for explanations of variables and flags. Each comment should describe the function of its statement, and generally, it alone. If a comment is needed to describe the function of a block of half-a-dozen cards, it probably should be placed on a comment card preceding the block of code. These comments should be punched when the program is originally punched. A good technique is to add these comments while keypunching the program. Often, this results in catching many mistakes at that point. It is noted that few novice programmers do this, while most experts do. It is also noted that many programmers who do this wish they had started doing so earlier, since they realize how much time they had wasted by not commenting the original deck.

DO

use TITLE, SPACE, and EJECT commands. The command

```
TITLE 'A HEADING MESSAGE'
```

skips the listing to a new page, and prints the heading message at the top of every page until another TITLE command is issued. This not only clearly labels your listing, but it saves time in looking through a listing which is more than a few pages long. The command

```
EJECT
```

skips the listing to a new page, and is useful in blocking off major parts of a program. The command

```
SPACE n
```

inserts n blank lines into the listing at that point. This is useful for blocking off smaller sections of a program, particularly small loops, register equates, etc.

Not only do listing control instructions aid to the readability of a program, but they also save the programmer time in debugging.

DON'T

merely restate an instruction when you place a comment on it. Of the following two examples, which is more explanatory?

```
A     1,VAR       ADD VAR TO REGISTER 1

A     1,VAR       R1=SUMMATION OF ODD PRIME NUMBERS
```

DON'T

put several single comments between statements in an unreadable manner. It is often useful to indent a single comment to column 16. This keeps it from interfering with the reading of labels and opcodes, and thus distinguishes it from the machine instructions.

DO

use comment card blocks which list useful information. For example, a list of register allocation and usage is extremely helpful, not only in debugging, but also in revising a program. Such a list should appear as part of the preface to the appropriate section of code. Another example is a list of calling conventions for subroutines. For extensive programs, lists of the following might be kept at the beginning of each subroutine: MACROS USED, SUBROUTINES CALLED BY THIS SUBROUTINE, SUBROUTINES WHICH CALL THIS SUBROUTINE, VARIABLES USED BY THIS SUBROUTINE, VARIABLES CHANGED BY THIS SUBROUTINE, etc.

DO

block off large sections of comment cards. Large blocks of comments can begin in whatever column is appropriate, but in general, should use most of the card, since they will otherwise add a great deal of length to a program. For the sake of appearance, comments should be blocked off by blank lines (SPACE n) or lines of continuous characters. The most common characters used for this purpose are asterisks (in columns 1-71, or in just the odd columns). An esthetic appearance can be obtained by placing an asterisk in column 71 of each comment card in a major block, with lines of asterisks before and after the entire block of documentation.

DO

flag instructions which will be modified during execution in order to make programming logic obvious. This may be accomplished by using '*-*' or '$', the latter EQU'ed to zero, for any modified field. For example,

```
$        EQU   0                  $ => INST. MODIFIED IN EX

.......... other statements ..........

         STC   2,MVC+1            SET BUFFER LEN. FOR LATER
*                                 USE.

.......... other statements ..........

MVC       MVC    OUTPUT($),0(5)   MOVE VARIABLE  #  BYTES   INTO
*                                 OUTPUT BUFFER.

.......... other statements ..........
```

The above methods have been derived both from the examination of many professionally-written programs and from the authors' own experiences. Thus, they are not arbitrary rules but techniques which have been widely used and proven to be effective aids in programming assembler language.

```
         PRINT NOGEN
         EQUREGS
MAIN     CSECT
         XSAVE .                   ESTABLISH STANDARD LINKAGE
         CALL  NEXT                CALL LOWER ROUTINE
         XRETURN  SA=*              ESTABLISH SAVE AREA
         LTORG
NEXT     CSECT
         XSAVE .                   ESTABLISH STANDARD LINKAGE
         CALL  LAST                CALL LOWEST ROUTINE
         XRETURN  SA=*             ESTABLISH SAVE AREA
         LTORG
LAST     CSECT
         XSAVE .                   ESTABLISH STANDARD LINKAGE
         CALL  TRACE               CALL TRACE RTN TO PRNT S.A.
         XRETURN  SA=*             GENERATE SAVE AREA
         LTORG
*
*  THE ABOVE ROUTINES DO  NOTHING BUT ESTABLISH LINKS TO TRACE
*  THROUGH THE SAVE AREAS
*
*
*  ROUTINE TRACE PROVIDES A PRINTED TRACE OF THE NAMES OF THE
*  CSECTS OF ACTIVE S.A.'S.  IT USES DSECTS SAVEAREA AND NAMECONV
*  TO FORMAT THE SAVEAREA AND FIRST FEW BYTES OF THE PROGRAM.
*
TRACE    CSECT
         XSAVE SA=TRACESA          ESTABLISH LINKS
         USING SAVEAREA,R13
         USING NAMECONV,R15
         XPRNT =CL25'0BACK TRACE OF SAVE AREAS--',25
         L     R13,4(R13)           CONNECT TO FIRST ACTIVE S.A.
LOOP     LTR   R13,R13             CHECK IF END OF CHAIN
         BZ    DONE                IF YES, EXIT
         L     R15,REG15SAV        GET PTR. TO BEGIN. OF CSECT
         CLC   BRANCH,=X'47F0'     CHECK TO SEE IF VALID BRANCH
         BNE   ERROR               IF NOT, ABORT
         IC    R7,LENGTH           PICK UP LENGTH OF NAME
         BCTR  R7,R0               SET UP FOR EXECUTE
         EX    R7,MOVE             MOVE CHARS. OF NAME TO OUTPUT
         XPRNT OUT,40              PRINT NAME OF ROUTINE
         MVC   OUT+1(39),OUT       BLANK OUT OUTPUT AREA
         LM    R14,R11,REG14SAV    RELOAD REGS. (FOR RETURN)
         L     R13,BACKLINK        FOLLOW LAST LINK
         B     LOOP
DONE     XPRNT =CL25'0BACK TRACE COMPLETED',25
          LA    R13,TRACESA
         XRETURN  SA=TRACESA
ERROR    XPRNT =CL25'0ERROR IN TRACE-BACK',25
         ABEND 999                 ABORT
MOVE     MVC   OUT+1(*-*),NAME     INSTR. FOR EXECUTE
OUT      DC    CL40' '
         LTORG
```

```
*
*  THE FOLLOWING DSECT FORMATS THE SAVE AREA
*
SAVEAREA DSECT
UNUSED   DS    F
BACKLINK DS    F                   PTER TO HIGHER S.A.
FORELINK DS    F                   PTER TO LOWER S.A.
REG14SAV DS    F                   SAVE AREA FOR REG 14
REG15SAV DS    F                    START OF S.A. FOR REG 15-12
*
*  THE FOLLOWING DSECT FORMATS THE BEGINNING OF A CSECT.  IF THE
*  NAME CONVENTION IS FOLLOWED, THE FIRST INSTR MUST BE A BR. ON
*  R15 AS A BASE REG. FOLLOWED BY A  LENGTH AND A NAME.
*
NAMECONV DSECT
BRANCH   DS    XL2,XL2             SPACE FOR BSC INSTR(4 bytes)
LENGTH   DS    C
NAME     DS    C                   SPACE FOR NAME (MARK BEGINNING
*                                    ADDR. ONLY)
         END   MAIN
/*
```

Following is the output from this example--

```
*** MAIN ENTERED ***
*** NEXT ENTERED ***
*** LAST ENTERED ***
*** TRACE ENTERED ***
BACK TRACE OF SAVE AREAS
TRACE
LAST
NEXT
MAIN
IEWLCTRL
BACK TRACE COMPLETED
*** TRACE EXITED ***
*** LAST EXITED ***
*** NEXT EXITED ***
*** MAIN EXITED ***
```

Following is the actual assembler listing of the TRACE csect. Notice those instructions which reference labels from the SAVEAREA and NAMECONV dsects. Look at the object code and see what the base register and displacement by which they were assembled is.

```
  LOC  OBJECT CODE    ADDR1 ADDR2  STMT   SOURCE STATEMENT
000300                             192 TRACE    CSECT
                                   193          XSAVE SA=TRACESA
000000                             220          USING SAVEAREA,R13
000000                             221          USING NAMECONV,R15
                                   222          XPRNT =CL15' BACK TRACE
000386 58DD 0004            00004  232          L     R13,4(R13)
00038A 12DD                        233 LOOP     LTR   R13,R13
00038C 4780 C07E            003E0  234          BZ    DONE
000390 58F0 D010            00010  235          L     R15,REG15SAV
000394 D503 F000 C1C6 00000 00528  236          CLC   BRANCH(2),=X'47F0
00039A 4770 C13A            0049C  237          BNE   ERROR
00039E 4370 F004            00004  238          IC    R7,LENGTH
0003A2 0670                        239          BCTR  R7,0
0003A4 4470 C170            004D2  240          EX    R7,MOVE
                                   241          XPRNT OUT,40
0003CE D226 C177 C176 004D9 004D8  251          MVC   OUT+1(39),OUT
0003D4 98EB D00C            0000C  252          LM    R14,R11,REG14SAV
0003D8 58D0 D004            00004  253          L     R13,BACKLINK
0003DC 47F0 C028            0038A  254          B     LOOP
                                   255 DONE     XPRNT =CL20'TRACE COMPL
000406 41D0 C0F2            00454  265          LA    R13,TRACESA
                                   266          XRETURN  SA=TRACESA
                                   285 ERROR    XPRNT =CL20'ERROR IN TR
                                   295          ABEND 999,DUMP
0004D2 D200 C177 F005 004D9 00005  303 MOVE     MVC   OUT+1(*-*),NAME
0004D8 4040404040404040            304 OUT      DC    CL40' '
000500                             305          LTORG


                                   310 *
000000                             311 SAVEAREA DSECT
000000                             312 UNUSED   DS    F
000004                             313 BACKLINK DS    F
000008                             314 FORELINK DS    F
00000C                             315 REG14SAV DS    F
000010                             316 REG15SAV DS    F
                                   317 *


000000                             318 NAMECONV DSECT
000000 47F0 F000            00000  319 BRANCH   B     0(,15)
000004                             320 LENGTH   DS    C
000005                             321 NAME     DS    C
                                   322 *
000000                             323          END   MAIN
```

ASSIGNMENT A - DUMPS AND JCL INTRODUCTION

The first assignment (not to be turned in) is essentially to use the Job Control Language and deck setups to be used most often during term, and also to become familiar with the completion dumps issued by OS/360.

I. JCL AND DECK SETUPS - OS/360

A. LOADER CATALOGED PROCEDURE - ASGCG

The following is the RECOMMENDED PROCEDURE for any assembler run which cannot be run under ASSIST, and which does not require some of the special facilities available using the LINK EDITOR. The procedure ASGCG stands for ASSEMBLER G COMPILE AND GO, and it has two JOB STEPS, SOURCE, and DATA. The typical deck setup is:

```
// EXEC ASGCG,PARM.DATA='MAP'
//SOURCE.INPUT DD *
..... 360 assembly language source deck
/*
//DATA.SYSUDUMP DD SYSOUT=A            (required for a dump)
//DATA.XSNAPOUT DD SYSOUT=A            (required if XSNAPs are used)
```

The above procedure is the most efficient way to assemble and run an assembler program which cannot be run by ASSIST.

B. LINK EDITOR CATALOGED PROCEDURE - ASGCLG

This procedure is somewhat slower than the above, but can be used to run somewhat large programs, and offers additional features. It stands for ASSEMBLER G, COMPILE, LINK, and GO, and contains three JOB STEPS, SOURCE, OBJECT, and DATA.

```
// EXEC ASGCLG,PARM.OBJECT='MAP'
//SOURCE.INPUT DD *
..... 360 assembly language source deck
/*
//DATA.SYSUDUMP DD SYSOUT=A            (required for a dump)
//DATA.XSNAPOUT DD SYSOUT=A            (required if XSNAPs are used)
```

C. EXECUTION ONLY CATALOGED PROCEDURE - ASGG

In some cases, the user may have an OBJECT DECK rather than a SOURCE DECK, in which case he does not need the assembler at all. The procedure ASGG has only 1 step, DATA, and just executes the program.

```
// EXEC ASGG,PARM.DATA='MAP'
//DATA.DECK DD *
.....input object deck.....
/*
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.XSNAPOUT DD SYSOUT=A
```

*****NOTE***** YOU MUST INCLUDE SYSUDUMP CARD TO GET ANY DUMP AT ALL. ALSO, USING XSNAP WITHOUT XSNAPOUT WILL RESULT IN ABEND U0300.

D. ADDITIONAL USEFUL JCL

1. MSGLEVEL=1 . If is often useful to see the JCL of cataloged procedures used. Punch a comma after the programmer name on your JOB card, punch anything in column 72 of you JOB card, then follow the JOB card with the following:

```
//  MSGLEVEL=1
     or for even more information, use:
//  MSGLEVEL=(1,1)
```

2. /*LOG CARDS. A card having /*LOG in columns 1-5 can be put anywhere in your deck except in the middle of continued cards or in data or source decks. It causes additional information to be printed in the system log, which is the very beginning of your output. This is strongly recommended for multi-step jobs, since it shows how far a a program progressed, and how much time each step needed.

3. /*INCLUDE CARDS. Your instructor may sometimes place cards on magnetic disk (such as test programs), and make them accessible to you. Each FILE on disk has a FILENAME, and you can essentially copy each such file by using a /*INCLUDE card referencing that file. Files may contain both data and JCL. Your instructor has an identification which must be used to refererence the files. The form is:

```
/*INCLUDE ident.filename
          for example:
/*INCLUDE JRM02.TESTDECK
```

4. /*DUMP CARDS. If your program runs out of time or records, you do not normally receive a completion dump. Inserting a card with /*DUMP in columns 1-6 anywhere in your deck (like /*LOG) will allow you to get a dump, assuming you have a SYSUDUMP card also in the correct place.

4. PARM FIELDS. PARM fields can be used to pass information to a processing program, with up to 1 PARM for each STEP in a cataloged procedure. The following options may be useful to you:

SOURCE STEP (ASSEMBLER G)

NOESD	Deletes 1st page - External Symbol Dictionary, which is often not too useful.
NOLIST	Deletes entire source listing - especially good for a debugged program when you want to save records.
NOXREF	Deletes the Cross-Reference from end of listing, saves time and records.
DECK	Requests that an OBJECT DECK be punched. You can then run the program (or include it with another one) without having to assemble it again, saving time.

OBJECT STEP (LINK EDITOR)

MAP	Requests a MAP of the modules in your program.

DATA STEP (LOADER)

MAP	Requests a MAP of where modules are loaded into memory.

EXAMPLES

```
// EXEC ASGCG,PARM.SOURCE='NOESD,NOXREF,DECK',PARM.DATA=MAP
// EXEC ASGCLG,PARM.SOURCE=NOLIST,PARM.OBJECT='MAP'  (missing 's ok)
```

II. OS/360 DUMP ASSIGNMENT

This assignment should help familiarize you with the typical cards used to run an assembler program using Assembler G, and show several of the most common causes of error termination, with the effects they have on the completion dumps printed by OS/360.

Run each of the following programs, using appropriate JCL cards. Use MSGLEVEL=1 and /*LOG cards for all of them. Use procedure ASGCG for all programs, and ASGCLG for part C. in addition. (total 4 runs)

A. TYPICAL INTERRUPT DUMP - PRECISE INTERRUPT - 0C6

```
DUMP1    CSECT
         XSAVE ID=NO
         SPACE 2
         L     0,2                cause 0C6
         XRETURN SA=*             return, create save area
         END   DUMP1
```

B. ABEND DUMP - CAUSED BY USER PROGRAM ABEND

```
DUMP2    CSECT
         XSAVE TR=NO
         SPACE 2
         ABEND 400,DUMP           U0400  completion code
         XRETURN TR=NO,SA=*       no trace, generate save area
         END
```

C. INTERRUPT DUMP WITH MULTIPLE CSECTS, IMPRECISE INTERRUPT - 0C4

```
DUMP3    CSECT
         XSAVE
         SR    0,0                0 for main program
         CALL  SUB1               CALL SUBROUTINE
         XRETURN SA=*
SUB1     CSECT
         XSAVE
         LA    0,1                set to 1 for 1st level sub
         CALL  SUB2
         CNOP  0,8                line up for max overrun
         ST    0,20               0C4 - store into protected core
         AR    0,0                get value of 2
         AR    0,0                get value of 4
         XRETURN SA=*
SUB2     CSECT
         XSAVE
         LA    0,2                set to another value
         XSNAP
         XRETURN SA=*
         END   DUMP3
```

```
//*************************************************************
//*
//*   E X C P    P R O G R A M
//*
//*                     CMPSC 411 - 12/7/71
//*************************************************************
// EXEC ASGCG,PARM.DATA='MAP'
//SOURCE.INPUT DD *
         TITLE  'EXCP - EXECUTE CHANNEL PROGRAM - TEST PROGRAM'
CHANPROG CSECT
         PRINT  NOGEN
         XSAVE
         PRINT  GEN
         OPEN  (INDCB,(INPUT),OUTDCB,(OUTPUT))  OPEN DCBS
*                                  I.E.  CONNECT BLOCKS TO
*                                  ALLOW IO TO TAKE PLACE
    SPACE  3
         EXCP  OUTIOB              EXEC. THE CHANNEL PROG.
*                                  BEGINNING AT LOC. OUTIOB
         MVI   OUTCCW,X'11'        CHANGE FROM SPACING TO TOP
    SPACE 3
*                                  OF NEW PAGE TO SINGLE SPACING
READ     EXCP  INIOB               EXEC. CHAN. PGM. TO READ A PIECE
*                                  OF DATA AS SPECIFIED IN INIOB
   SPACE 3
         WAIT  ECB=INECB           GO INTO WAIT STATE AND DO NOT
*                                  PROCESS ANY MORE UNTIL THIS IO
*                                  HAS POSTED COMPLETION IN INECB
         CLI   INECB,X'41'         CHECK FOR GOOD TERMINATION
         BE    EOF                 IF TERM. CODE =X'41', EOF READ
  SPACE  3
         EXCP  OUTIOB              EXEC. CH PGM AT OUTIOB
         WAIT  ECB=OUTECB          WAIT FOR TERMINATION OF IO.
         B     READ                LOOP
         SPACE 5
EOF      CLOSE (INDCB,,OUTDCB)     CLOSE OPEN DATA SETS
  PRINT  NOGEN
         XRETURN  SA=*
         SPACE 5
INAREA   DC    CL80' '             AREA FOR INPUT
INDCB    DCB   MACRF=E,            USING EXCP MACROS                  X
               DDNAME=IN           UNDER THE DDNAME FOR JCL 'IN'
INECB    DC    F'0'                BLOCK TO POST STATUS OF IO
         SPACE 3
INIOB    DC    B'01000010',X'000000'  INFORMATION ON FORM OF
*                                  IO ACTIVITY
         DC    A(INECB)            LOCATION TO POST STATUS
         DC    2F'0'               **MAGIC**USED FOR STATUS
         DC    A(INCCW)            ADDR. OF CHANNEL COMMANDS
         DC    A(INDCB)            PTR. TO DCB ASSOCIATED WITH
*                                  THIS IO TASK
         DC    2F'0'               **MAGIC**USED FOR STATUS
         SPACE 3
*  COMMAND WORD FOR READ FOLLOWS--
INCCW    DC    0D'0'               MUST BE DOUBLE WORD ALIGNED
         DC    X'02'               COMMAND = READ
         DC    AL3(INAREA)         WHERE TO PLACE DATA
         DC    B'00100000'         FLAGS = SUPPRESS INCORRECT LEN.
         DC    X'00'               UNUSED (BUT MUST BE INCLUDED)
         DC    H'80'               LENGTH OF DATA FIELD TO READ
*  EVERYTHING DOWN TO HERE HAS BEEN ONE COMMAND WORD FOR ONE IO
```

```
*  ACTIVITY (A READ)
         SPACE 5
OUTDCB   DCB   MACRF=E,DDNAME=OUT
OUTECB   DC    F'0'                EVENT CONTROL BLOCK FOR OUTPUT
         SPACE 3
*  COMMAND BLOCK FOR OUTPUT FOLLOWS--
OUTIOB   DC    X'42000000'         INFORMATION FOR CHANNEL.
         DC    A(OUTECB)           ADDR. OF ECB FOR OUTPUT
         DC    2F'0'               MAGIC STATUS
         DC    A(OUTCCW)           ADDR. OF CHAN. COMM. WORD(S)
         DC    A(OUTDCB)           ADDR. OF ASSOCIATED DCB
         DC    2F'0'               MAGIC STATUS AGAIN
         SPACE 5
*  OUTPUT CHANNEL COMMAND WORD FOLLOWS--
         DS    0D
OUTCCW   DC    X'8B'               SKIP TO NEW PAGE
         DC    AL3(INAREA)         AREA FROM WHICH TO TRANSFER
         DC    B'00100000'         STATUS FLAGS
         DC    X'00'               UNUSED
         DC    H'80'               LENGTH OF DATA TRANSFER
         SPACE 5
*  JCL USED--
* // EXEC ASGCG,PARM.DATA='MAP'
* //SYSIN DD *
*    SOURCE DECK
* /*
* //DATA.OUT DD SYSOUT=A  FOR THE DATA SET 'OUT' CREATED AT THE
*                         DATA LEVEL, TO BE ROUTED TO A PRINTER
* //DATA.SYSUDUMP DD SYSOUT=A  JUST IN CASE
* //DATA.IN DD *          THE DATA SET 'IN' CREATED AT THE
*                        DATA STEP, WILL FOLLOW DIRECTLY
*   DATA CARDS
* /*
         SPACE 5
         END   CHANPROG
//DATA.OUT DD SYSOUT=A
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.IN DD *
  THIS IS DATA CARD #1
  THIS IS THE SECOND DATA CARD
  THIS IS THE LAST DATA CARD--AFTER IT THE DCB'S WILL BE CLOSED
```

```
/*LOG
//*
//*      PROGRAM  FLOTLINK:  THIS PROGRAM ILLUSTRATES THE FOLLOWING:
//*      1. JOB CONTROL LANGUAGE FOR LINKING FORTRAN/ASSEMBLER
//*      2. CALLING LINKAGE AMONG FORTRAN/ASSEMBLER MODULES
//*      3. FLOATING POINT INSTRUCTIONS.
//*
//*      FLOTLINK CONSISTS OF A FORTRAN MAIN PROGRAM, A FORTRAN
//*      FUNCTION (F2A), AND A FORTRAN SUBROUTINE (F3A), PLUS SEVERAL
//*      ASSEMBLER LANGUAGE SUBROUTINES (A1, A2).
//*
//*      THE EXEC FGC REQUESTS A FORTRAN G COMPILATION.
//*
// EXEC FGC
//SOURCE.INPUT DD *
C        MAIN PROGRAM - READS VALUES, CALLS A1.
      WRITE(6,9000)
 1000 READ(5,9020,END=8000) A,B,C,I
      WRITE(6,9040) A,B,C,I
C        EXAMPLE:  CALL ASSEMBLER SUBROUTINE FROM FORTRAN.
      CALL A1(A,B,C,I)
      GO TO 1000
 8000 WRITE(6,9060)
      STOP 20
 9000 FORMAT('1***** FLOTLINK - FORTRAN/ASSEMBLER FLOATING PT/LINKAGE')
 9020 FORMAT(3F10.0,I10)
 9040 FORMAT('0***** INPUT VALUES: A,B,C,I = ',3F20.10,I10)
 9060 FORMAT('0***** END EXECUTION OF FLOTLINK')
      END
      FUNCTION F2A(A,B,C,I)
C        THIS FORTRAN FUNCTION COMPUTES THE FOLLOWING VALUES:
C        F2A = (A/2.) * B + (C**I)/10. - 2.
C        IT GENERATES CODE RETURNING RESULT IN FLOATING PT REGISTER 0.
C        IF RESULT WERE FIXED PT, IT WOULD BE IN GP REGISTER 0.
      WRITE(6,9000)
      F2A = (A/2.) * B + (C**I)/10. - 2.
      WRITE(6,9020) F2A
      RETURN
 9000 FORMAT(' *** FUNCTION F2A ENTERED ***')
 9020 FORMAT(' *** RETURN FROM F2A, RESULT = ',E15.8)
      END
      SUBROUTINE F3A(VALUE,TITLE,NTITLE)
C        THIS SUBROUTINE IS USED TO PRINT THE SINGLE PRECISION FLOATING
C        POINT  VALUE, FOLLOWED BY TITLE  HAVING NTITLE CHARACTERS.
      LOGICAL*1 TITLE(NTITLE)
      WRITE(6,9000) VALUE,TITLE
      RETURN
 9000 FORMAT(E15.8,60A1)
      END
//*
//*      THE EXEC ASGCG REQESUTS FIRST AN ASSEMBLY AND THEN EXECUTION.
//*      EXECUTION WILL BEGIN AT ROUTINE NAMED 'MAIN'.
//*
//STEP2 EXEC ASGCG,PARM.SOURCE=NOXREF,PARM.DATA='MAP,EP=MAIN'
//SOURCE.INPUT DD *
         TITLE 'MODULE A2 OF FLOTLINK'
A2       CSECT
*        WITH ARGUMENTS (A,B,C,I), A2 COMPUTES SAME AS F2A:
*        F2A = (A/2.) * B + (C**I)/10. - 2.
*        RESULT IS RETURNED IN FLOATING POINT REGISTER 0.
         PRINT NOGEN
```

```
         EQUREGS
         EQUREGS L=F,DO=(0,6,2)    FLOATING POINT EQUATES
         SPACE
         XSAVE
*             R1 CONTAINS ADDRESS OF ADDRESS LIST OF ARGUMENTS.
         LM    R2,R5,0(R1)         GET ADDRESSES OF A,B,C,I
         LE    F0,0(,R2)           LOAD VALUE OF A
         HER   F0,F0               = A/2.  USING HALVE SHORT INSTR
         ME    F0,0(,R3)           = (A/2.) * B, MULTIPLY SHORT
*             NOW COMPUTE C**I / 10. PART
         LD    F2,=D'1'            INITIALIZE, WILL MULITPLEY IN HERE
         LE    F4,0(,R4)           F4 = C, WILL SAVE IN REGISTER
         L     R6,0(,R5)           R6 = I, FOR LOOP COUNTER
         XSNAP T=FL,LABEL='REGISTERS IN A2'
         LPR   R0,R6               R0 = ABS(I)
         BZ    A2NOMUL             IF I =0, C**I = 1.0 ALWAYS
         MER   F2,F4               F2 * C, LOOPING
         BCT   R0,*-2              LOOP, MULTIPLYING TO GET C ** ABS(I)
         SPACE
         LTR   R6,R6               WAS I POSITIVE
         BP    A2NOMUL             YES, SO CAN SKIP, F2 IS OK
         LD    F4,=D'1'            GET VALUE OF 1 FOR DIVIDE
         DER   F4,F2               DIVIDE TO GET 1.0 / (C ** ABS(I))
         LER   F2,F4               GET CORRECT VALVE OF C ** I
A2NOMUL  DE    F2,=E'10'           F2 = (C ** I) / 10.
         AER   F0,F2               F0 = (A/2.)*B + (C ** I)/10.
         SE    F0,=E'2'            F0 = (A/2.)*B + (C ** I)/10. - 2
*             RESULT HAS BEEN COMPUTED AT THIS POINT.
*             SAVE ACROSS CALL (FLOAT REGS NOT PROTECTED), AND PRINT.
         STE   F0,A2RESULT         SAVE THE RESULT
         PRINT GEN
         SPACE 2
         CALL  F3A,(A2RESULT,A2MSG,A2LEN)
         SPACE 2
         PRINT NOGEN
         LE    F0,A2RESULT         RELOAD RESULT
         XRETURN SA=*,TR=NO        RETURN, NO TRACE
A2RESULT DS    E
A2MSG    DC    C' COMPUTED BY A2 AT EXIT'
A2LEN    DC    A(L'A2MSG)
         LTORG
         TITLE 'MODULE A1 OF FLOTLINK'
A1       CSECT
*             THIS MODULE ILLUSTRATES THE CALLING SETUPS FOR CALLING
*          BOTH FORTRAN AND ASSEMBLER ROUTINES.  IT CALLS A2, F2A.
           SPACE
*          ON ENTRY, R1 CONTAINS THE ADDRESS OF AN ADDRESS LIST, FOR
*          ARGUMENTS A, B, C, I,  AS FOLLOWS:
*  R1 ==> (0(R1), ADDRESS OF A) ==> VALUE OF A
*          (4(R1), ADDRESS OF B) ==> VALUE OF B
*          (8(R1), ADDRESS OF C) ==> VALUE OF C
*          (12(R1),ADDRESS OF I) ==> VALUE OF I
         XSAVE
*             DUMP THE ADDRESS LIST FOUND ON ENTRY.
          XSNAP STORAGE=(*0(R1),*16(R1)),                              #
               LABEL='R1 ==> 4 FULLWORDS OF ADDRESS LIST'
*             MOVE THE ARGUMENTS OVER.  THIS IS NOT REALLY NECESSARY,
*             IT IS JUST DONE TO SHOW ACCESSING OF ARGUEMNTS.
         LM    R2,R5,0(R1)         LOAD PTRS TO VALUES OF A,B,C,I
         MVC   A,0(R2)             GET LOCAL COPY OF A
         MVC   B,0(R3)             GET LOCAL COPY OF B
```

```
         MVC   C,0(R4)             GET LOCAL COPY OF C
         MVC   I,0(R5)             GET LOCAL COPY OF I
*              AT THIS POINT,LOCAL A,B,C,I HAVE SAME VALUES AS THOSE
*              SUPPLIED IN ORIGINAL PROGRAM.
*              NOW MAKE CALLS TO ROUTINES AND SEE HOW THEY RETURN VALS.
         LA    R1,ADCONS           GET ADDRESS OF ADCON LIST
         L     R15,=V(F2A)         ADDRESS OF FORTRAN ROUTINE
         BALR  R14,R15             CALL THE ROUTINE
         XSNAP T=FL,LABEL='F0 CONTAINS RESULT OF FUNCTION F2A'
*              NOW CALL EQUIVALENT ASM ROUTINE, USING CALL MACRO.
         PRINT GEN
        SPACE 2
         CALL  A2,(A,B,C,I),VL
        SPACE 2
         PRINT NOGEN
         XSNAP T=FL,LABEL='F0 CONTAINS RESULT OF FUNCTION A2'
         XRETURN SA=*
A        DS    E
B        DS    E
C        DS    E
I        DS    F
ADCONS   DC    A(A,B,C),X'80',AL3(I)    ADCON LIST
         LTORG
         END
//DATA.XSNAPOUT DD UNIT=AFF=FT06F001    XSNAPS ON FT06F001
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.FT05F001 DD *
       10.        5.       10.         2
        1.       10.        .1        -1
/*
```

```
//*
//*      THIS JOB WILL RUN WITH TIME = 25 SECONDS
//*                             RECORD = 600
//*
// EXEC ASGCG
//SYSIN DD *
         PRINT NOGEN
         EQUREGS
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*   THE PURPOSE OF THIS PROGRAM IS TO DEMONSTRATE THE GETMAIN AND
*   FREEMAIN MACRO USING THE REGISTER CONVENTION
*   FIRST READ IN THE NO OF BYTES TO BE OBTAINED FROM THE OS  THEN
*   READ IN NUMBER TO FILL THESE BYTES AND PLACE THEM IN THE AREA
*   OBTAINED NEXT SNAP THESE NUMBERS  AND FREE THE AREA WITH A
*   WITH A FREEMAIN MACRO
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
MAIN     CSECT
         XSAVE
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*   THE FIRST SECTION PREPARES FOR THE GETMAIN MACRO
*   FIRST READ IN THE NUMBER OF BYTES TO BE OBTAINED FROM THE OPERATING
*   SYSTEM THEN ECHO PRINT THE  NUMBER
*   MOVE THE NUMBER OF BYTES TO BE OBTAINED INTO REG 0 AND MAKE A
*   COPY OF THIS NUMBER IN R9
*   DIVIDE R4 BY 4 TO GET THE NUMBER OF NUMBERS TO BE READ IN
*   THEN PLACE THE SUBPOOL NUMBER IN R0 ALONG WITH THE NUMBER OF BYTES
*   REQUESTED
         XREAD WORD               READ IN THE NUMBER OF BYTES TO BE
*                                 OBTAINED FROM THE OPERATING SYSTEM
         XPRNT WORD-1,81          ECHO PRINT THE NO OF NUMBERS TO BE R
         XDECI R4,WORD            CONVERT TO INTERNAL FORM
         LR    R0,R4              MAKE A COPY OF R4 FOR GETMAIN
         LR    R9,R4              MAKE A COPY OF THE NUMBER  OF BYTES
         SRL   R4,2               DIVIDE THE NUMBER OF BYTES BY 4 TO
*                                 GET THE NUMBER OF WORDS TO BE READ
         LA    R3,1               PLACE A 1 IN BIT 31 OF R3
         SLL   R3,24              MOVE THE BIT TO BIT 7 OF R3
         OR    R0,R3              PLACE SUBPOOL IN R0
         PRINT GEN
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*   USING A GETMAIN MACRO OBTAIN THE NUMBER OF BYTES REQUESTED
*   THE REGISTER CONVENTION REQUIRES THAT THE SUBPOOL NUMBER BE PLACED
*   IN THE FIRST BYTE OF REG 0
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
         GETMAIN R,LV=(0)
         PRINT NOGEN
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*   MAKE TWO COPIES OF THE ADDRESS OF THE AREA OBTAINED FROM THE
*  OPERATING SYSTEM THEN USING R4 FOR LOOP READ IN THOSE NUMBERS
*   AND PLACE THEM IN THE AREA OBTAINED
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
         LR    R6,R1              MAKE A COPY OF ADD OF NEW STORAGE
         LR    R7,R1              MAKE A COPY OF ADD OF NEW STORAGE
LOOP     XREAD WORD               READ IN THE NUMBERS
         XPRNT WORD-1,81          ECHO PRINT THE NUMBERS
         XDECI R5,WORD            CONVERT THE NUMBERS TO INTERNAL FORM
         ST    R5,0(R6)           PUT NEW NUMBER IN STORAGE
         LA    R6,4(R6)           INCREASE POINTER TO NEXT NEW WORD
         BCT   R4,LOOP            IF NOT LAST NUMBER RETURN FOR NEXT
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*   FINALLY  SNAP THE AREA OBTAINED AND THEN FREE THE AREA OBTAINED
```

```
*   PLACE THE SUBPOOL NUMBER AND THE NUMBER OF BYTES TO BE FREED IN
*   R0 AND THEN USING THE REGISTER CONVENTION FREE THE AREA OBTAINED
*   WITH THE REGISTER CONVENTION PUT THE ADDRESS OF THE AREA TO BE
*   FREED IN A REGISTER
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
         XSNAP STORAGE=(*0(7),*4(6)),T=NOREGS,                          X
               LABEL='THIS IS A SNAP OF THE AREA OBTAINED FROM GETMAIN'
         OR    R9,R3               PLACE SUBPOOL NUMBER IN FIRST BYTE
         LR    R0,R9               PUT THE PS NUMBER AND LENGTH IN R0
*                                  FOR THE MACRO CALL
         PRINT GEN
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*   USING A FREEMIAN MACRO FREE THE AREA OBTAINED WITH THE REGISTER
*   CONVENTION THE SUBPOOL NUMBER IS PLACED IN THE FIRST BYTE OF R0
*   THE ADDRESS OF THE AREA TO BE FREED IS PLACED IN A REGISTER
*   DESIGNATED BY THE A= PARAMETER
*   .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
         FREEMAIN R,LV=(0),A=(7)
         PRINT NOGEN
         XRETURN SA=*
         DC    F'0'
WORD     DC    20F'0'
         END
/*
/*LOG
//DATA.XSNAPOUT DD UNIT=AFF=FT06F001
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.INPUT DD *
40
0
1
2
3
4
5
6
7
8
9
/*
```

PENN STATE UNIVERSITY COMPUTATION CENTER
360/67 CONFIGURATION
this writeup: pages 01 - 04, plus Diagram A (separate).

INTRODUCTION

This writeup briefly describes the devices included in the PSU 360/67 system, and shows how they are connected together. Each device is described below, and diagram A shows the connections.

References are made to DEVICE ADDRESSES. Each individually addressable device (such as a single disk drive, card reader, etc) has a 3 digit (hexadecimal) number which uniquely identifies it to the system, and is used in all input/output operations. The DEVICE ADDRESS is of the following form:

abc where:
a gives the CHANNEL NUMBER (from 0 up)
b specifies a CONTROL UNIT attached to that channel
c notes which device attached to a given control unit.

Since each digit can have the value 0-F, theoretically it would be possible to attach 16 devices to each of 16 control units attached to 16 channels, for a maximum of 4096 separate devices. In practice, this number is much less, since most S/360's allow a MAXIMUM of 7 channels or less.

The devices follow, more or less in order from the CPU outward.

CENTRAL PROCESSING UNIT

2067-1 (a single 360/67 CPU). uses 200 nanosec (.2 microsec) cycle Read Only Storage (ROS) of 88 bits/word to implement S/360 instruction set (Universal plus special model 67 instructions) includes a HIGH RESOLUTION TIMER (13 microsec cycle). includes a BCU (Bus Control Unit), which is connected to all memory modules, and determines which channel or CPU gets to use a given memory module.

PRIMARY STORAGE

2365 III (4 units) each unit contains 256K bytes. Physically each 2365 contains 2 arrays of 128K bytes, with physical word size of doublewords, i.e., each has 2 arrays of 16K doublewords, and is thus 2-way interleaved at this level. Each 2365 is independent of the others.
CYCLE TIME: 750 nanosec / ACCESS TIME: 375 nanosec

2361 II (1 unit) - Large Core Storage (LCS) - 2048K bytes, organized physically of 2-way interleaved doublewords.
CYCLE TIME: 8000 nanosec (8 microsec) / ACCESS TIME: 3.2 mic

Of the two types of storage, the first contains user programs, and heavily used parts of system programs, while the LCS contains less-used system programs, tables, and buffer areas.

CHANNELS

2870 MULTIPLEXOR CHANNEL - includes 2 SELECTOR SUBCHANNELS (used for magnetic tape drives). generally handles LOW-SPEED devices (card readers, printers, etc)
MAXIMUM TOTAL TRANSFER RATE: 426 KB (kilobytes) per second

2860 SELECTOR CHANNELS - 5 total (2 in 2860 II, 3 in 2860 III). used for HIGH-SPEED devices (disk, drum, etc)
MAXIMUM DATA TRANSFER UNIT, EACH SELECTOR: 1250KB

All CHANNELS and the CPU contend for use of memory modules. The BCU arbitrates among them using a simple priority scheme in following order:

```
          SERVED EARLIER   --->  SERVED LATER
CHANNEL # :    1    2    0    3    4    5    CPU
             drums disk  mx  disk disk  ADAGE
```

The above order is used since the drums cannot wait very long and have the highest transfer rate, the multiplexor channel (0) is fairly early because it may have a large number of things to do, and the CPU is always last because it never hurts it to wait.

CONTROL UNITS

Each control unit can attach to a number of devices, and it is used to control greatly different devices in a such a way as to make them appear more alike, as far as the channels are concerned. Each device must be attached to a particular type of control unit, and each control unit normally can control a group of related devices.

2820 STORAGE CONTROL UNIT - controls the 2301 drum units, attached to channel 1 .

2821 CONTROL UNIT - controls UNIT RECORD devices (card readers, printers, punches). attached to multiplexor channel.

2848 DISPLAY CONTROL - controls the 8 2260 scopes which display system status to the operators.

2701 DATA ADAPTOR - controls a small number of high-speed transmission lines, i.e. high speed terminals (4800 bits/sec transmit rate), such as 360/20's at various locations.

2703 TRANSMISSION CONTROL - controls a larger number of lower-speed terminals, including typewriter/teletype terminals and read/print/punch terminals at Commonwealth Campuses (such as IBM 2780, DCS CP-4, etc).

DISPLAY DEVICES

1052 CONSOLE TYPEWRITER - messages are printed here requiring action by computer operators, and they can enter commands to the system at this location.

2260 ALPHAMERIC DISPLAY SCOPES (8 units) - these display current system status (jobs, disk usage, etc), and also are used to display requests for magnetic tapes to be mounted, etc.

SECONDARY STORAGE - DIRECT ACCESS STORAGE DEVICES (DASDs)

2301 MAGNETIC DRUMS (2 drums) - attached to channel 1 via 2820. Each holds 4.09 megabytes (million bytes) of data, rotates once each 17.5 milliseconds, with average rotational delay (latency time) of 8.6 milliseconds. Records data 4 bits in parallel (for high transfer rate). Has 200 conceptual TRACKS, each of 20,483 bytes maximum size. EACH DRUM IS UNREMOVABLE. MAXIMUM TRANSFER RATE: 1.2 megabytes/second (FASTEST DEVICES USED ON THIS SYSTEM).
These hold most heavily-used compilers and system programs.

231x (2314, 2319) MAGNETIC DISK STORAGE FACILITIES - total of 22 disk drives (including 2 spare ones). Each DRIVE holds one 2316 DISK PACK: 29.17 megabytes maximum, on 20 disk surfaces (11 plates - outside ones not used). Uses MOVABLE HEADS to access information. Each CYLINDER (of which there are 200 usable at any one time) contains the 20 TRACKS accessible at one time without moving the READ/WRITE HEADS. Each track can record at most 7294 bytes of information.
NOTE: unlike drums, each DISK PACK can be removed, and another one mounted in is place if desired.
ROTATION TIME: 25 millisec, AVERAGE LATENCY: 12.5 millisec.
SEEK TIMES (time to move HEADS to correct cylinder):
MIN = 25, AVERAGE = 60 or 75, MAX = 130 or 135 millisec.
MAXIMUM DATA TRANSFER RATE: 312,000 bytes/sec.

NOTE: each of the three storage facilities contains its own control unit, and each drive is numbered accordingly, i.e., 230-237, 330-337, 430-433, on channels 2, 3, 4.

TOTAL DASD STORAGE IS AS FOLLOWS:

2314 (8 drv)	233	megabytes
2319 (8 drv)	233	megabytes
2314 (4 drv)	116	megabytes
2301 (2 drums)	8	megabytes
TOTAL	590	megabytes (approx)

SECONDARY STORAGE - SEQUENTIAL DEVICES

240x (2402 III, 2403 III) MAGNETIC TAPE DRIVES - read/write tape at maximum density of 800 BPI (bits/inch), 9 tracks per tape (2 of the drives also read/write 7-track tapes). Each group of 4 drives is connected to one SELECTOR SUBCHANNEL of the MULTIPLEXOR CHANNEL. The control units for these drives are contained in the 2403 units.
MAXIMUM TRANSFER RATE: 90,000 bytes/sec (90KB), using tape speed of 112.5 inches per second, tape gaps of .6 inch between blocks of data.

UNIT RECORD DEVICES

1403 LINE PRINTERS (of various models), printing with maximum rated speed of 1100 lpm (lines/minute) for 1403 N1, 600 lpm for others. Use removable TRAINs, so that different character sets can be obtained (upper case only: QN, upper/lower: TN). Attached to 2821 control units (on multiplexor).

2540 CARD READ/PUNCH - one unit contains a card reader and card punch (treated logically as separate addresses: for example: 00C for reader, 00D for attached punch).
READS cards (optically) at 1000 cpm (cards/minute) maximum.
PUNCHES cards at 300 cpm maximum.
Attached to 2821 control unit.

2671 PAPER TAPE READER - reads punched paper tape at up to 1000 cps (characters per second). attached also to 2821 control unit.

SUMMARY OF DEVICE CHARACTERISTICS

DEVICE TYPE		CAPACITY PER UNIT (megabytes)	TRANSFER RATE (KB/second)	AVERAGE DELAY (seek)	(latency) ms.
2301	DRUM	4.09	1200	0	8.6
2319	DISK	29.17 per pack	312	60	12.5
2314	DISK	29.17 per pack	312	75	12.5
2400	TAPE DRIVE	varies, 20 per 2400-ft tape OK	90	-	-
1403	PRINTER	132 bytes/line	2.4	-	-
2540	READER	80 bytes/card	1.3	-	-
2540	PUNCH	80 bytes/card	0.4	-	-
2671	PAPER TAPE	--	1.0	-	-

REFERENCES: GA22-6810 IBM S/360 SYSTEM SUMMARY
GA27-2719 IBM S/360 MODEL 67 FUNCTIONAL CHARACTERISTICS

CMPSC 102 - INDEX OF BAT FILES - J R MASHEY

The following provides a brief index to BAT files available for CMPSC 102. Unless otherwise specified, these files are kept under the following RJE ID: JRM02.

CS102AS1 - first assignment, mainly for arithmetic operations

CS102FP1 - final project writeup - write assembler/interpreter for
CS102FP2 - what is basically an XDS SIGMA 5 subset computer

CS102M1 - two writeups: beginning run setup; explanation of the conventions used to make up S/360 opcode mnemonics

CS102TPA - day-by-day outline of most of CS 102 course

FLOTLINK - sample program illustrating FORTRAN/ASSEMBLER LINKAGE and floating-point instructions.

A number of files used in CMPSC 411 may also be suitable for CMPSC 102(410). See also file JRM02.CS411GI1.

Some of the files mentioned in CS411GI1 include:

DOCUMENT - hints on good documentation for assembler
DSECT - sample DSECT usage
DUMPSJCL - typical dump setups; common ASM G JCL setups
LINKAGE - explanation of OS/360 linkage conventions

CMPSC 411 - INDEX OF BAT FILES - J R MASHEY

The following provides a brief index to materials useful for CMPSC 411 (11).

CS411GI1 - contains general information about CMPSC 411, text
CS411GI2 - materials, and also has further index to BAT files contained in CS411GI1. Approximately 45 files of sample programs, assignments, writeups, etc are listed here.

CS411TPA - contains detailed descriptons of day-by-day lectures

INDEX102 - contains index to CMPSC 102 files, which in some cases may overlap with 11 or 411.

*****NOTE: above files are held under RJE ID JRM02.

STANDARD LINKAGE CONVENTIONS

Charles Pfleeger

Under OS/360, certain conventions have been established regarding the use of registers. These conventions will have been followed when you, the problem programmer, receive control from the system; they should be followed for any routines which you call, or for communicating with the system (e.g. system macro calls, SVC's, returning control, etc.). Following these conventions will make your code easier for someone else to follow. Certain debugging aids are also available for those who adhere to standard conventions. In general, unless there is a strong reason to deviate, these conventions should be employed.

REGISTER 14 is called the return register and contains the address to which this routine is to return upon exit.

REGISTER 15 is called the entry point register, and contains the address through which this routine was entered. Note that temporary addressability may be established by

```
        USING   entrypointname,15
```

If this routine calls no other routines, register 15 may be used as a permanent base register. If this routine calls any other routines, however, register 15 will be changed, and should not be used as a permanent base register. In this latter case, the sequence

```
        LR      BASEREG,15
        USING   entrypointname,BASEREG
```

(where BASEREG is any of registers 2-12) may be used to establish permanent addressability.

On return, register 15 may be used to return a code to indicate normal or error return. One frequently-used technique is to set R15 zero on a normal return and set it non-zero if some error condition occurred prior to return.

REGISTER 0 is used to return the single result from some process (as in a Fortran function subprogram). Note: although you will probably not use this convention much, it is heavily used by the operating system. Register 0 cannot be guaranteed to be intact after executing some call to the system, as a system macro, or an SVC.

REGISTER 1 is the pointer to an argument list. It contains the address of the first of one or more full word entries (on consecutive f.w. boundaries). These entries are the addresses of arguments to be used by the calling routine.

If there may be an indefinite number of arguments, (as with a routine which would accept one, two, or any number of arguments--c.f. Fortran MAX0), the first bit of the last address is set to a 1. (This bit will not interfere with ordinary S/360 addresses, since an address can be fully specified in 3 bytes; byte 1 is ignored on an address constant.)

The following example illustrates how to use the address list passed through register 1.

```
            LA   1,ARGLIST      get argument list address
            L    15,=V(CALLRTN) get entry address
            BALR 14,15          call routine
             . . .
ARGLIST     DC   A(ARG1)
            DC   A(ARG2)
             . . .
            DC   X'80',AL3(ARGn)  Note the length factor
                                  does not provide auto-
                                  matic alignment.
             . . .
CALLRTN     CSECT
             . . .
            L    2,0(1)         get addr. of next arg.
            LTR  1,1            last arg. in list?
            BM   RETURN         if yes, return
            LA   1,4(1)         else get addr. of next arg.
```

When a programmer receives control from the system, information from the PARM field of his EXEC card is passed via register 1. Register 1 points to a fullword of storage. Bit 0 of this fullword is set to 1 (to indicate the last--only--argument of the list). This fullword contains the address of a halfword. The halfword is a count of the number of characters in the parm field message, and these characters follow immediately after the halfword count field. The contents of the halfword may be picked up to use as a length count in an execute instruction, and the address of the halfword may be used as a base to move the information characters of the PARM field.

REGISTER 13 is called the save area register. It contains the address of an 18 fullword area (on a f.w. boundary) within the calling routine. The routine called will use this area to save the contents of registers, to be able to return the registers intact to the calling program. This save area has a set format:

```
Word 1    Used by PL/I and FORTRAN
Word 2    address of the save area used by the calling
          program.
Word 3    address of the save area set up by the called
          program.
Word 4    address to which to return (reg. 14).
Word 5    address of entry point (reg. 15).
Word 6    contents of register 0.
. . .
Word 18   contents of register 12.
```

Save areas are chained in a doubly-linked list. At any low-level routine, by tracing back through a chain of save area links, one can eventually return to the system at the original point of call.

When your routine is entered, first you should save registers and then establish and link your own save area.

```
          STM  14,12,12(13)   save regs. 14, 15, and 0-12 in
                              calling program's save area.
          LA   5,MYSAVE       get addr. of my save area
          ST   5,8(13)        link calling pgm. s.a. to mine
          ST   13,4(5)        link my s.a. to calling pgm's
          LR   13,5           transfer pointer to s.a.
```

On return:

```
          L    13,4(13)       retrieve addr. of calling pgm's
                              save area
          LM   14,12,12(13)   restore registers as they were
          BR   14

MYSAVE    DC   18F'0'
```

A calling program is known as a "higher routine", and the routine called is the "lower routine". Register 13 is always to point to an area whose contents may be destroyed.

An exception to the requirement that a routine must always establish a save area is that the lowest-level routine (the one which calls no others) need not set up a save area. The reason for this is the save area is for the use of any called routines, but that the lowest-level routine will have no called routines.

It is important to know the conventions on save areas, but the use of XSAVE AND XRETURN (consult appropriate documentation) can reduce the problems in coding and linking save areas.

THE NAME CONVENTION is a means of having the EBCDIC form of the name of a routine appear at certain key places on dumps. To use this convention, the first four bytes of a routine must be a branch, on 15 as a base register, which passes over a series of bytes.
These bytes contain the EBCDIC form of the name of a routine, and also a length count for this name area. This example shows how to code a name field.

```
     name      CSECT
               B    m+1+4(,15)
               DC   X'm'
               DC   CLm'name'
               next instruction.
```

The value of m must be odd, in order to have the next instruction properly aligned. An alternate approach uses the convention on register 15:

```
     name      CSECT
               USING     name,15
               B         NEXTINST
               DC        X'm'
               DC        CLm'name'
     NEXTINST  next instruction
```

Notes:

O/S follows these conventions strongly. In particular, the system often destroys the contents of registers 0, 1, 14, and 15 when it returns control from a system macro, an SVC, or another system function. One must SAVE THE CONTENTS of these registers BEFORE executing one of these functions; hard-to-locate errors will frequently occur after failure to do so.

It is a good idea to mark a save area upon exit. This is usually done be moving X'FF' into the first byte of the fourth word of the save area (the place register 14 was stored). Although this technique does not seriously affect the contents of the save area for reading in a dump, this technique quickly shows what save areas are active and which are not active when reading a dump.

Register 13 must be kept as the save area pointer; however, by careful programming, it can also double as a base register. Consult the appropriate section from XSAVE and XRETURN documentation for the coding sequence using these macros. You may set up your own save area for this purpose by setting it high in a program, and following it by a USING on register 13, referencing the name of the save area.

For reserving the 18 fullwords of storage for a save area, use DC instead of DS. A constant of F'0', or F'-1' will quickly show in a dump if the save area was ever used.

SAVE and RETURN are two system macros which will eliminate much of the coding for saving and returning conventions. SAVE generates the code necessary to save a specified series of registers. The registers are specified as they would be for a STM instruction. In addition, the operand T will cause registers 14 and 15 to be stored, regardless of what other registers may also be saved from the pair specified. The following example will cause registers 5, 6, ... 10 and 14 and 15 to be saved.

```
SAVE (5,10),T
```

The RETURN macro will generate code to restore registers, insert a return code in register 15, flag the save area (X'FF' in wd. 4), and branch back via register 14. The registers to be restored are coded as with SAVE. If 15 already has a return code in it and should not be restored, it is coded as RC=(15); else RC=n may be coded, where n is some value to insert into register 15. The operand T causes the flag X'FF' to be inserted in the save area. The following code will restore registers 5, 6, ... 10 to be reloaded, the save area to be flagged, and 15 to be loaded with a value 16.

```
RETURN   (5,10),T,RC=16
```

NOTE Both of these macros expect that register 13 will already be loaded with the address of the appropriate save area.

The use of the PSU macros XSAVE and XRETURN can provide added flexibility in saving and restoring registers. Both can generate code to print a trace message showing entry and exit from a module; XSAVE can be used to establish and load a base register or to print a snap of the registers saved; XRETURN can create a save area. NOTE that as with RETURN, XRETURN assumes that register 13 still points to the relevant save area.

For most uses, the code XSAVE alone can be used to save registers. For a routine with only one return point, XRETURN SA=* will suffice; if a routine has more than one return point, however, XRETURN alone should be coded at all return points except one, and at that one XRETURN SA=* should be coded. The reason for this is that SA=* will cause a save area to be created; only one should be created per module. For further details on the parameters involved in these two macros, see the appropriate PSU documentation.

The following example causes register 12 to be established as a base register, causes all registers to be saved on entry, cuases no trace messages to be printed on entry or on exit, and causes R15 to be loaded with the return code value 8.

```
        MAIN    CSECT
                XSAVE    BR=12,TR=NO   (Note--default is for all
                                        registers to be saved)
                XRETURN  SA=*,TR=NO,RC=8
```

```
//*      THIS JOB WILL RUN  WITH TIME = 110
//*                              RECORDS = 800
//*
// EXEC ASGCL
//SOURCE.INPUT DD *
*
*        .         .         .         .         .         .         .
*        THE PURPOSE OF THIS PROGRAM IS TO DEMONSTRATE THE MACROS
*        GIVEN BELOW
*                  LOAD
*                  DELETE
*                  LINK
*                  XCTL
*        .         .         .         .         .         .         .
*
*
*        .         .         .         .         .         .         .
*        THE OVERALL FLOW OF THIS PROGRAM IS AS FOLLOWS:
*        1         THE FOLLOWING CSECTS ARE ASSEMBLED AND LINK EDITED
*                  SECOND, THIRD, AND FOURTH.
*        2         MAIN IS ASSEMBLED AND LINKEDITED THEN IT IS GIVEN
*                  CONTRLL AND EXECUTES.
*        3         DURING EXECUTION OF MAIN SECOND IS LOADED USING
*                  THE LOAD MACRO THEN SECOND IS CALLED USING CALL
*                 MACRO.   CONTROL IS THEN RETURNED TO MAIN AND
*                  SECOND IS DELTED USING THE DELTTE MACRO.
*        4         THIRD IS LOADED AND CONTROL PASSED TO IT USING THE
*                  LINK MACRO.  CONTROL IS RETURNED TO MAIN THROUGH
*                  THE LINK MACRO CONTROL PROGRAM, AND THE OONTROL
*                  PROGRAM DELETES THIRD.
*        5         FOURTH IS LOADED AND GIVEN OONTROL THROUGH THE XCTL
*                  MACRO.  THE XCTL MACFO DELETES MAIN AND THEN CONTROL
*                  IS RETURNED TO THE OPERATING SYSTEM USIGN
*                  IS RETURNED TO THE OPERATING SYSTEM.
*        .         .         .         .         .         .         .
*
         EJECT
         PRINT NOGEN
*
*        .         .         .         .         .         .         .
*        THE PURPOSE OF THIS CSECT IS TO BE LOADED USING THE LOAD
*        MACRO AND THEN TO BE CALLED USING CALL MACRO.  THEN IT PRINTS
*        A MESSAGE AND RETURNS TO MAIN.  NOTE THE NORMAL SAVE AND
*        RETURN CONVENTIONS.
*        .         .         .         .         .         .         .
*
SECOND   CSECT
         XSAVE  TR=NO
         OPEN  (OTPT,OUTPUT)
         PUT   OTPT,SHEAD
         CLOSE (OTPT,)
         XRETURN SA=*,TR=NO
SHEAD    DC    CL80'0SECOND HAS BEEN LOADED AND CALLED RETURN TO MAIN'
OTPT     DCB   DSORG=PS,MACRF=PM,LRECL=80,BLKSIZE=80,RECFM=FA,         X
               DDNAME=FT06F001,EROPT=ACC
         PRINT GEN
         END
/*
//OBJECT.SYSLMOD DD DSNAME=&&LOADMOD(SECOND)
// EXEC ASGCL
//SOURCE.SYSGO DD DISP=(OLD,PASS)
```

```
//SOURCE.INPUT DD *
*
*        .         .         .         .         .         .         .
*        THE THIRD CSECT IS ENTERED VIA THE LINK MACRO.  THE LINKAGE
*        CONVENTION APPEARS STANDARD, BUT ACTUALLY REGISTER 14 POINTS
*        TO AN ADDRESS IN THE CONTROL PROGRAM OF THE LINK MACRO.
*        THEREFORE, THE LINK CONTROL PROGRAM GETS CONTROL WHEN CONTROL
*        IS PASSED FROM THIRD TO MAIN AND FROM MAIN TO THIRD.
*        .         .         .         .         .         .         .
*
         PRINT NOGEN
THIRD    CSECT
         XSAVE TR=NO
         OPEN  (ATPT,OUTPUT)
         PUT   ATPT,THEAD
         CLOSE (ATPT,)
         XRETURN SA=*,TR=NO
THEAD    DC    CL80'0THIRD ENTERED VIA LINK MACRO RETURN TO MAIN'
ATPT     DCB   DSORG=PS,MACRF=PM,LRECL=80,BLKSIZE=80,RECFM=FA,         X
               DDNAME=FT06F001,EROPT=ACC
         PRINT GEN
         END
/*
//OBJECT.SYSLMOD DD DSNAME=&&LOADMOD(THIRD),DISP=(OLD,PASS,DELETE)
// EXEC ASGCL
//SOURCE.SYSGO DD DISP=(OLD,PASS)
//SOURCE.INPUT DD *
*
*        .         .         .         .         .         .         .
*        THE FOURTH CSECT IS GIVEN CONTRLO THROUGH THE XCTL MACRO.
*        WHEN CONTROL IS PASSED VIA THE XCTL MACRO, CONTROL IS NOT
*        RETURNED TO THE STEP ISSUING THE XCTL MACRO, AND THE STEP
*        ISSUING THE XCTL MACRO IS DELETED BY THE XCTL MACRO.  THEREFOR
*        ISSUING THE XCTL MACRO IS DELETED BY THE XCTL MACRO.
*        THEREFORE, THE REGISTERS ARE RELOADED IN THE ISSUING STEP SO
*        THAT WHEN THE STEP SHICH IS XCTLED TO RETURNS IT RETURNS TO
*        THE PROPER POINT
*        .         .         .         .         .         .         .
*
         PRINT NOGEN
FOURTH   CSECT
         XSAVE TR=NO
         OPEN  (ETPT,OUTPUT)
         PUT   ETPT,FHEAD
         CLOSE (ETPT,)
         XRETURN SA=*,TR=NO
FHEAD    DC    CL80'0FOURTH LOADED AND ENTERED VIA XCTL MACRO'
ETPT     DCB   DSORG=PS,MACRF=PM,LRECL=80,RECFM=FA,BLKSIZE=80,         X
               DDNAME=FT06F001,EROPT=ACC
         PRINT GEN
         END
/*
/*LOG
//OBJECT.SYSLMOD DD DSNAME=&&LOADMOD(FOURTH),DISP=(OLD,PASS,DELETE)
// EXEC ASGCLG
//SOURCE.SYSGO DD DISP=(OLD,PASS)
//SOURCE.INPUT DD *
*
*        .         .         .         .         .         .         .
*        THIS IS THE MAIN JOB STEP.  WHEN IT RECEIVES CONTROL, IT FIRST
*        LOADD SECOND AND CALLS SECOND, THEN WHEN CONTRO IS RETURNED
```

```
*        IT PASSES CONTROL TO THIRD VIA THE LINK MACRO, FINALLY IT XCTL
*        TO FOURTH AND CONTROL IS NOT RETURNED.
*        .         .         .         .         .         .         .
*
         PRINT NOGEN
MAIN     CSECT
         XSAVE TR=NO
         PRINT GEN
*
*        .         .         .         .         .         .         .
*        THE LOAD MACRO INSTRUCTION IS CODED SO THAT THE LOAD MODULE
*        WITH ENTRY POINT SECOND IS LOADED INTO CORE.  SINCE DCB IS
*        OMMITED IT SEARCHES THE STEPLIB WHICH IS INCLUDED BY
*        //DATA.STEPLIB DD DSNAME=&&LOADMOD,DISP=(OLD,PASS)
*        THE ENTRY ADDRESS OF SECOND IS RETURNED IN REGISTER 0 SO THAT
*        SECOND CAN BE CALLED
*        .         .         .         .         .         .         .
*
         LOAD  EP=SECOND
         PRINT NOGEN
         LR    15,0                GET ADD OF SECOND CSECT IN 15
         CALL  (15)                BRANCH TO SECOND
         PRINT GEN
*        NOW THAT WE HAVE FINISHED WITH SECOND WE DELETE LOAD MODULE.
*        AGAIN THE ENTRY POINT IS SPECIFIED.    THE DELETE MACRO MUST
*        BE ISSUED IN THE SAME TASK AS THE LOAD MACRO.
*
*        .         .         .         .         .         .         .
         DELETE EP=SECOND
*
*        .         .         .         .         .         .         .
*        IN THE LINK MACRO WE CODE THE ENTRY POINT OF THE LOAD MODULE
*        TO WHICH WE SISH TO LINK.  THE PARAM SPECIFIES A PARAMETER
*        LIST WHICH IS PASSED IN REGISTER 1.  THE VL = 1 INDICATES A
*        VARIABLE NUMBER OF ARGUMENTS.  THESE ARGUMENTS WILL NEVER
*        BE USED IN THIRD THEY ARE FOR EXAPMLE.
*        .         .         .         .         .         .         .
*
         LINK  EP=THIRD,PARAM=(PARM1,PARM2),VL=1
*
*        .         .         .         .         .         .         .
*        BEFORE WE CAN PASS CONTROL TO FOURTH WE MUST RELOAD REGISTERS
*        13 AND 14 TO POINT THE WAY THEY DID BEFORE ENTRY TO MAIN.
*        THEN SINCE WE USE THE ENTRY POINT CONVENTION TO FIND THE LOAD
*        MODULE WE LET THE MACRO RESOTRE REGISTER 2 THRU 12.
*        THE XRETURN IS INCLUDED TO PROVIDE A SAVEAREA FOR THE XSAVE
*        .         .         .         .         .         .         .
*
         L     13,4(13)            GET ADD OF SYSTEM SAVEAREA
         L     14,12(13)           RESTORE REGISTER 14
         XCTL  (2,12),EP=FOURTH
         PRINT NOGEN
         XRETURN SA=*,TR=NO
PARM1    DC    F'0'
PARM2    DC    F'1'
         END
/*
//OBJECT.SYSLMOD DD DSNAME=&&LOADMOD(MAIN),DISP=(OLD,PASS,DELETE)
//DATA.STEPLIB DD DSNAME=&&LOADMOD,DISP=(OLD,PASS)
//DATA.SYSUDUMP DD SYSOUT=A
```

OVERVIEW OF OS/360 WITH HASP

This writeup gives a quick overview of the process by which any OS/360 sytstem is initialized, how storage is used (particularly in OS/360/MVT), and describes how OS/360 is modified by the use of HASP (Houston Automatic Spooling Priority system). The storage layout is described for the PSU CC 360/67 system.

I. INITIALIZATION - getting a system up and running

Consider a computer with no operating system currently in it. The first necessity is to get a workable operating system in it, so that jobs can be run. This is NOT a trivial process: note that there is no Program Fetch resident in the machine, no I/O Access Method routines, and not even a correct set of PSW's in low core for directing interrupt actions.

For OS/360, the initialization process is composed of two parts: IPL and NIP. IPL (Initial Program Loader) initializes memory and some other things, and brings the NUCLEUS (the core of the OS) into memory. NIP (Nucleus Initialization Program) performs the remaining actions required to set up a specific NUCLEUS to be ready to execute.

A. IPL - Initial Program Loader

The process of getting an OS/360 system running is called IPLing, and includes the following main steps:

1. The operator makes sure the disk pack called SYSRES (SYStems RESidence) is mounted on a disk drive. The LOAD UNIT switches are set to show the device address of the SYSRES disk pack, and the LOAD button pressed. This causes the CONTROL RECORD to be read from the first record on the disk pack, consisting of a PSW and two CCW's, placed at location 0 in memory. Execution of this record causes the IPL BOOTSTRAP record to be read into memory. The BOOTSTRAP record consists of a set of CCW's which are used to read the IPL program into memory, beginning at location 0. It ends with a LPSW to give control to the IPL program.

2. IPL selects which NUCLEUS will be loaded (there may be a choice which can be given by switches on the operator console).

3. IPL clears all memory above itself to zeroes, also obtaining the size of memory; i.e., it stores until addressing interrupt occurs.

4. IPL clears the floating point registers, thus finding out if the floating point feature is installed.

5. IPL brings the NUCLEUS into memory. First, it relocates the part of itself not yet executed into high memory (near 252K), so that the NUCLEUS can be placed beginning at 0. It then simulates Program Fetch, loading the csects of the NUCLEUS load module into memory. The first csect loaded is the NIP, loaded just below IPL, followed by the I/O Interrupt Handler at 0 (which thus defines all of the special PSW's in low core). IPL then passes control to NIP.

B. NIP - Nucleus Initialization Program

The IPL process described above applies to all versions of OS/360. The NIP is generated in different ways, depending on the specific type of system and choice of options desired. Note: NIP is a csect which is link-edited with the nucleus, so that it can refer to sections of the nucleus via address constants, and provide efficient and specific initialization services. It includes the following steps:

1. The CVT (Communications Vector TAble) is initialized, and its location placed at location 16, so that it can be accessed from any routine, whether part of the nucleus or not.

2. NIP determines whether the computer has Large Core Storage (LCS) attached to it or not. This is particularly necessary for those systems which include HIARCHY SUPPORT, i.e., the ability to usefully distinguish between main core and LCS, perhaps splitting programs into heavily-used and lesser-used sections.

3. NIP checks the workability of operator console(s), and also checks the workability of ready direct-access devices (using TIO instructions). It particularly checks that the SYSRES volume is mounted and contains certain datasets needed by the system.

4. NIP performs various housekeeping actions, such as checking and setting the timer to make sure it is working correctly, initializing some pointers for storage management, initializing the SVC table (which gives a pointer to each routine associated with a defined SVC number). It also sets up to be able to obtain modules from the SYS1.LINKLIB, which contains the heaviest-used load modules for the system, and also establishes communications with the operator.

5. For any system having one, NIP loads reentrant modules into the LINK PACK. These modules can be used during following execution, and are located at the high end of memory. In a system with fast core+LCS, the LINK PACK can be split, residing at both the high end of fast core and the high end of LCS.

6. With the addition of various other miscellaneous operations, NIP prepares a REGION which will contain the MASTER SCHEDULER, which is the program doing overall job scheduling and operator communication. It then can pass control (LINK or XCTL) to the MASTER SCHEDULER, and the system is finally ready to run jobs.

At this point, memory layout (fast core only) is as follows:

```
HIGH ADDRESS        LINK PACK           (reentrant modules)
                    MS (MASTER SCHEDULER)
                    FREE AREA           (dynamic for problem programs)
                    SQS (SYSTEM QUEU SPACE) (contains space for system
                                         control blocks - TCB's, etc)
LOW ADDRESS         NUCLEUS
```

NOTE: in systems with HIARCHY SUPPORT, FREE AREA, MS, and LINK PACK would also have areas in LCS.

II. RUNNING JOBS IN AN OS/360 SYSTEM

This section describes how jobs are run in a standard OS/360 system, using either OS-MFT or OS-MVT. Note that OS-PCP runs jobs one at a time (sequential scheduling, uniprogramming), with no SPOOLing of jobs to and from disk before and after execution. OS-MFT and OS-MVT are generally similiar in that they both can SPOOL input onto DASD, execute jobs in priority order, and write the output out later. The main difference is in the handling of storage, in which OS-MVT is much more dynamic. Note that all scheduling of jobs and communication with the operator is effectively under the control of the MASTER SCHEDULER.

A. READING INPUT STREAMS

For each existing input stream (card reader, or input on tape), the operator can issue a START RDR command. This causes a copy of the READER/INTERPRETER program (referred to herafter as a RDR) to read cardimages from the requested input device.

During its operation, a RDR reads an input stream, scans JCL cards and converts them to a standard internal text form, and also obtains cataloged procedure definitions from the procedure library (PROCLIB). From the internal text, it builds INPUT QUEUE entries, representing the information on the user JCL cards. It also writes any input data cards onto disk, while placing pointers to the data into the INPUT QUEUE entries so that it can be found later. The job's INPUT QUEUE entry is enqueued in priority order with other jobs awaiting execution.

When all of the cards for a job have been read, it has in effect been split up into the following:

1. INPUT QUEUE ENTRIES, in priority order, in a special system data set used only for work queue entries, referred to as SYSJOBQUE.
2. INPUT STREAM DATA SETS, placed on DASD, using normal OS/360 Direct Access Device Storage Management (DADSM) routines. NOTE: DADSM routines are themselves kept on DASD, nonresident, and allocating disk space often requires a fair number of accesses to disk to look for free space on one, and to allocate the space appropriately. The DADSM routines are quite general and powerful, but also create some overhead.

B. INITIATING JOB STEPS

The operator may START one or more INITIATORs, each of which can initiate jobs from one or more classes(categories) of jobs. Each initiator will then attempt to initiate the highest priority job from the first class of jobs which has a ready job. If there are no jobs awaiting execution in its allowed classes, it WAITs for one to become available. Note that it essentially removes input queue entries from SYSJOBQUE. Like every RDR, each INITIATOR is executed as a separate task. (INITIATOR may be abbreviated INIT).

When an allowable job becomes available, the initiator obtains a REGION for the job (from the FREE area, also called the DYNAMIC area), uses the information from the RDR to allocate DASD storage, tape drives, and other I/O devices. It then ATTACHes the first module of the program to be executed (thus creating the JOB STEP TASK), and WAITs until the job step completes.

When a job step is finished, the TERMINATOR (part of the INITIATOR really, so that the whole unit is called an INITIATOR-TERMINATOR) effectively cleans up, performing disposition of I/O devices (DISP parameter in JCL), and releasing the REGION which had been acquired for the job step.

During this process, job steps are essentially independent, i.e., they could require different sizes of regions, and might execute in different locations. Note that the INITIATOR-TERMINATOR must also control the skipping of steps as controlled by the JCL COND option.

During execution, SYSOUT datasets are written to DASD, to be printed/punched later. When the last job step of a job completes, the INIT creates a work queue entry calling for the job's output to be printed/punched.

C. WRITING SYSTEM OUTPUT

A program called a SYSTEM OUTPUT WRITER (WTR) can be STARTed by the operator to transcribe output from DASD to printers or punches, or even tapes to be printed/punched later. Output can originally be grouped into CLASSes, which can be written according to priority or otherwise treated differently as desired.

COMMENTS ON THE PROCESS ABOVE

The process described above is quite flexible and general. However, it does require a fair amount of time to set up any job, even a small one. As such, it is quite satisfactory for any installation which runs jobs which require a fair amount of time, since then the setup time is negligible. However, due to the use of OS DADSM for PSPOOLed input and output, DASD space can become fragmented, disk head movement can become excessive, and much time can be used up allocating and deallocating disk space. Although OS/360 is quite reasonable in a commercial installation, or in one running a few large jobs, it seems to have too much overhead for university or other installations which often run many small jobs. For this reason, most larger S/360 computers (i.e., models 75,67,65, and larger 50's) typically use some method to reduce the overhead in running small jobs. All of the methods involve 'faking out' OS/360 in some aspect or other. The method emphasized here (which happens to be the most popular one) is HASP (Houston Automatic Spooling Priority) system.

III. RUNNING AN OS/360 HASP SYSTEM

In any OS/360 system, it is fairly typical to have one or more special jobs in the system, which are loaded before normal user jobs. and typically remain resident from one IPL to the next. Such jobs may control remote batch terminals, timesharing typewriter terminals, or provide any other service which the installation desires. Such jobs are normally placed into the high-address sections of the FREE area (or of the two FREE areas, if the system has both main core plus LCS). When HASP is used, it is normally the first job submitted to OS/360, and it essentially takes over the system, even though it appears to OS/360 as just another job.

A. HASP INITIALIZATION

There are two possible cases when starting HASP up after an IPL. A COLD START occurs when the system is completely empty, i.e., there are no jobs already enqueued on disk which can be executed. If there are disk packs on the system containing previously-read jobs, the start is called a WARM START. A WARM START normally occurs if the system was previously taken down on purpose, such as for systems programming, or if enough information had been saved previous to a 'crash'. A COLD START only occurs when the system crashes badly, and destroys records of jobs already SPOOLed onto disk. In this case, the jobs must be read in again.

When HASP first gains control, it issues a special SVC call, which returns to HASP with protect key 0 and supervisor state, also supplying HASP with some useful pointers to control blocks in the nucleus. NOTE: this special SVC can only be called 1 time, since it locks after its first usage after an IPL.

UCB's (Unit Control Blocks) exist for every device connected to the computer system. HASP now scans these, and essentially allocates to itself:

1. All real unit-record devices (readers, punches, printers).

2. All disk packs which have volume label names beginning SPOOL.

It also obtains effective control of the operator's console(s), plus remote terminals, if any.

Finally, HASP modifies the SVC table (which contains pointers to the routines which are called for each specific SVC number), so that the following ones go into HASP, rather than to the original routines (also saving these addresses for later use for itself):

```
SVC  0    (EXCP - all input/output)
SVC 34    (WTL - write to log)
SVC 35    (WTO, WTOR - write to operator, with/withput reply)
```

B. RUNNING NORMAL USER JOBS UNDER OS/360 WITH HASP

1. Input Stage - HASP continually reads cards from whatever card readers are active in the system. It checks for JOB cards, performs various accounting checks on input jobs, and transcribes the jobs to disk. In this stage, each job is split up into two sections: the JCL cards (with certain modifications), and the input data cards. It enqueues the jobs according to a priority scheme, which can be found from many different sources of information. These include category, time, output, storage requirements, originating site of job, and commands from the operator to change priority of either single jobs or entire groups of jobs. The disk allocation scheme used is quite efficient, and is described later.

2. Execution Stage - HASP has the ability to control which jobs may execute in which portions of the OS FREE area, and using the various priority and storage requirements, it selects jobs from its queue to be executed. One OS RDR exists, permanently STARTed to a card reader. This card reader does not actually exist (i.e., it has a device address which does not correspond to a real card reader). Since SVC 0's are intercepted by HASP anyway, HASP effectively selects a job and feeds it to the OS RDR, which thinks the job is coming across a real card reader. The OS RDR includes an EXIT LIST, which allows it to call some routine after it has scanned each JCL Card, but before the JCL card's data is actually recorded. HASP is entered, and takes this opportunity to modify any JCL that it wishes to, for example, removing any REGION= requests on JOB or EXEC cards. HASP has special treatment for any system input or output data sets, as described below:

//XXXXXXXX DD * or DATA : the OS RDR would normally expect data to follow such a card, and would normally thus SPOOL such to disk itself. HASP does not want this to occur, since it has already SPOOLed the data. It happens that there are large number of UCB's for pseudo card readers already in the system. HASP selects one of these UCB's which is not being used, and effectively changes the tables for this type of card so that it appears as:

//XXXXXXXX DD UNIT=xxx

As a result, the OS RDR thinks that the data set will be read from unit xxx, so that it does not try to SPOOL the input. In any case, the input no longer follows that JCL card, because HASP feeds the RDR only the JCL cards of a user job. During this process, HASP connects up the device address xxx to the specific input data set which had been previously SPOOLed.

//XXXXXXXX DD SYSOUT=x : HASP also has a large number of UCB's for nonexistent, pseudo printers/punches. It does the same thing to this kind of card as it does to DD * cards, except that it only allocates the pseudo devices, and will later save the output which is written to them.

As soon as the RDR finishes reading a job, an initiator can immediately initiate it, since HASP chooses jobs appropriately. When the initiator chooses i/o devices, it finds that it can always allocate devices for unit-record i/o, since HASP had already checked to make sure a pseudo reader/printer/punch was available for each SYSIN or SYSOUT data set.

Finally, a job step of the user job executes. When it wishes to read cards or print lines, it acts as though it were using a real device attached to the system, and OS/360 accepts this. Whenever an SVC 0 is issued to request such I/O, HASP intercepts it.

HASP may be entered for any of the following reasons:

1. WTO, WTOR, WTL - HASP adds own processing as desired.
2. I/O to disk, drum, tape, terminals, etc - HASP does not interfere, but passes these on to the real I/O Supervisor.
3. I/O to real unit-record devices - these have probably been issued by HASP in the first place, so it passes control to the real I/O routines to let them perform the I/O.
4. I/O to a pseudo device - these must be caused by user program. For input, HASP fetches the cardimages from disk into memory (if they are not already present), and feeds requested cardimage(s) to the user program by MVCing them there (using user protect key for safety). For output, it blocks up output and eventually writes it to disk. In all cases, HASP simulates the effect of having real card readers/printers/punches, which are odd only in possessing great speed; i.e., the effect on OS/360 is of having issued an I/O request and having had it complete immediately.

During execution, HASP can also provide extra services, such as monitoring time used, output records, etc. It also reorders priorities of executing user tasks so that I/O bound jobs have higher priorities than do CPU-bound ones. This action (which is unknown to OS/360) helps minimize time spent waiting .

3. Output Stages - Print and Punch - after a job has been executed, it enters the Print queue, is printed, enters the Punch queue, and has punched output (if any) actually punched. This activity occurs without the knowledge of OS/360, which believes the job disappeared whenever it finished execution. Only when a job is finished punching is its disk space released. This allows for jobs to be saved across system crashes, and for such useful services as repeating output by operator control.

C. DASD STORAGE MANAGEMENT IN HASP

HASP manages its DASD storage quite efficiently, not only needing NO accesses to DASD to allocate or deallocate space, but also doing a good job of minimizing arm movement on moveable-head devices. HASP requires the use of entire volumes (normally 2311 or 2314 disks). For example, the PSU CC's 360/67 has 3 2314 disk packs for HASP. The management of this storage works as follows:

A MASTER CYCLINDER BIT-MAP is maintained in HASP. This is a string of bytes, in which each bit represents 1 CYLINDER on the SPOOL disks (for example, 600 bits for the cylinders on 3 packs). A one-bit represents a FREE CYLINDER, while a zero-bit shows that the given cylinder is allocated to some job. HASP also remembers for each disk which cylinder was the last referenced, thus always noting the current position of the read/write heads.

Two JOB BIT-MAPS exist for each job, one for SYSIN data and the other for SYSOUT data. Whenever a cylinder is required for a job, HASP searches for a free one in the following fashion:

1. It first searches the master bit-map for a free cylinder at the current position of any read/write head, i.e., where it can read or write without even moving a head.
2. It then searches for a free cylinder at +1 from current head positions, then -1 from each, followed by +2, -2, etc up to +8, -8 cylinders away from current head position.
3. If the above fail, it searches sequentially through all cylinders in the master bit-map.

When a cylinder is found, its bit is turned off in the master bit-map, and turned on in the appropriate job bit-map. The overall effect of this process is to minimize head movement.

When disk storage for a job is to be released, the deallocation scheme is extremely fast and efficient: the job bit-maps are just OR'd into the master bit-map, thus returning all of the cylinders to free storage.

IV. OTHER PSEUDO-DEVICE SYSTEMS FOR USE WITH OS/360

The following are other systems which are based on OS/360, but use some kind of pseudo-devices to make it run faster.

A. ASP - ATTACHED SUPPORT PROCESSOR

In this system, 2 computers are used. All unit-record devices are attached to the multiplexor channel of a medium-sized 360, along with some disk. It performs all SPOOLing, control of remote terminals, etc. It is connected to a larger system via a chnnel. OS/360 is in the large system, and it reads its input and sends its output along the channel-channel hookup between the two CPU's. A typical setup would use a 360/50 hooked to a 360/75.

An advantage over HASP is that ASP offers somewhat better setup facilities for optimizing use of tapes and non-SPOOL disks. A disadvantage is the requirement of two CPU's, either of which may have problems, and thus stop the entire system.

B. LASP (LOCAL ASP) or CLASP (CLOSELY LINKED ASP)

These are versions of ASP in which the code from the smaller computer is moved over into a region on the larger machine. This allows an ASP system to be run on one processor. If the system is also run under straight ASP, it requires switches to switch the unit-record devices over to the bigger machine. It also requires more memory than HASP, but does allow the system to run even with one CPU down.

C. TUCC HYPERDISK

This method uses LCS plus part of a 2314 disk pack to simulate the entire disk pack containing heavy-used systems programs. The most recently used tracks of this disk are kept in LCS, thus making the disk effectively faster, without changing the internals of OS/360.

V. PSU CC 360/67 SYSTEM - OS/MVT WITH HASP

The following tables gi$e the current layout (with no guarantee of future appearance) as of 6/12/72, for the 360/67 at the PSU CC. The system has both fast core (1024K) and Large Core Storage (2048K).

	LOW	HIGH	K	LOW	HIGH
MS	2928	3072	144	2DC000	300000
HASP	1968	2928	960	1EC000	2DC000
FMGR	1628	1968	340	197000	1EC000
RJE	1346	1628	282	150800	197000
WATFOR	1336	1346	10	14E000	150800
RASP	1236	1336	100	135000	14E000
FREE	1024	1236	212	100000	135000
MS	964	1024	60	0F1000	100000
HASP	876	964	88	0DB000	0F1000
RDR	866	876	10	0D8800	0DB000
FMGR	852	866	14	0D5000	0D8800
RJE	832	852	20	0D0000	0D5000
WATFOR	704	832	128	0B0000	0D0000
FREE	122	704	582	01E800	0B0000
NUC	0	122	122	000000	01E800

NOTES

MS (Master Scheduler) includes the Link Pack areas. The fast core section contains mainly modules for the various I/O Access Methods, while the LCS part contains reentrant parts of INITIATORS, RDRS, plus other routines (overlay supervisor, special tables,etc).

HASP Fast core section is most heavily-used sections. LCS part has lesser-used sections, plus such items as in-core SYSJOBQUE (HASP intercepts all RDR and INIT reads/writes to SYSJOBQUE, and keeps such information in about 600K of LCS). Also has HASP buffers for all devices, plus tables of tape names/locations for user tapes.

FMGR File Manager - manages, synchronizes RJE, BAT files.

RJE Remote Job Entry - handles typewriter terminals.

WATFOR WATFOR REgion - RPSS - manages CAtegory W fast processors swapped in and out of memory (WATFOR, ASSIST, PL/C, etc).

RASP Interface between 360/67 and ADAGE AGT/30 graphics computer.

FREE fast core - 560K for user programs (4x140, 2x280, 1x280+2x 140, occasionally 1x560), rest for Sytem Queue Space.
LCS - currently unused, except for systems programs.

```
//JOBLIB DD UNIT=SYSDA,DSN=&&LOADMOD,SPACE=(CYL,(5,1,3)),DISP=(,PASS)
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//*                                                                     *
//*                       OVERLAY TEST PROGRAM                          *
//*                                                                     *
//*            THIS PROGRAM ILLUSTRATES THE USE OF THE OVLY OPTION      *
//*      FOR THE LINKEDITOR.  IT FIRST PRODUCES AN OBJECT MODULE        *
//*      OF A NUMBER OF CSECTS, THEN USES THE LINKE EDITOR TO PUT       *
//*      THEM TOGETHER IN VARIOUS WAYS, USING:                          *
//*      ASMLINK:   ASSEMBLE, THEN LINK MODULES 4 WAYS, PLACING THEM    *
//*                 IN &&LOADMOD AS MODULES MOD0 - MOD3.                *
//*            THE REMAINING STEPS EXECUTE THE MODULES MOD0-MOD3.       *
//*      STEP0:     NO OVERLAY WHATSOEVER                               *
//*      STEP1:     SIMPLE OVERLAY WITH THREE SEGMENTS                  *
//*      STEP2:     COMPLEX 1-REGION OVERLAY WITH 10 SEGMENTS           *
//*      STEP3:     THREE REGION OVERLAY - MOVES SUBROUTINES OUT        *
//*            OF THE ROOT SEGMENT.                                     *
//*                                                                     *
//*                 CALLING HIERARCHY CHART FOR THIS PROGRAM            *
//* LEVEL MODULE    CALLS ROUTINES AT LEVEL SHOWN (CSECTS  ONLY).       *
//*  5   MAIN                                                           *
//*            4    SUB1                                               *
//*            2    SUB1C, SUB2                                        *
//*                                                                     *
//*  4   SUB1                                                           *
//*            3    SUB1B                                              *
//*            2    SUB1C                                              *
//*            1    MSUB1, MSUB2                                       *
//*            0    MSUB3, MSUB4, SUB1A                                *
//*                                                                     *
//*  3   SUB1B                                                          *
//*            2    SUB1C                                              *
//*            0    SUB1D                                              *
//*                                                                     *
//*  2   SUB1C                                                          *
//*            1    MSUB2                                              *
//*      SUB2                                                           *
//*            1    MSUB1                                              *
//*            0    SUB2A, SUB2B, SUB2C                                *
//*                                                                     *
//*  1   MSUB1                                                          *
//*            0    MSUB3, MSUB4                                       *
//*      MSUB2                                                          *
//*            0    MSUB3, MSUB4                                       *
//*                                                                     *
//*  0   MSUB3, MSUB4, SUB1A, SUB1D, SUB2A, SUB2B, SUB2C                *
//*                                                                     *
//* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
//*
//*            STEP ASMLINK(SOURCE,OBJECT) : ASSEMBLE THE PROGRAM,
//*      THEN USE LINKEDITOR TO PRODUCE 4 LOAD MODULES, WITH THE
//*      MODULES LINKED IN INCREASING ORDER OF OVERLAY COMPLEXITY.
//*
//ASMLINK EXEC ASGCL,PARM.SOURCE='NOXREF',PARM.OBJECT='MAP,LIST,OVLY'
//SOURCE.SYSGO DD DSN=&&STUPID
//SOURCE.INPUT DD *
OVLY     TITLE 'TEST PROGRAM FOR OVERLAY OPTIONS'
         MACRO
&LABEL   OCALL &ENTRY
.*--> MACRO: OCALL      SPECIAL VERSION OF CALL TO SHOW V-A-ADCONS. . .
&LABEL   L     15,=A(&ENTRY) .     A-TYPE ADCONS
```

```
         L     15,=V(&ENTRY) .     V-TYPE ADCON
         BALR  14,15 .             CALL THE ROUTINE
         MEND
         SPACE 4
         XSET  XSAVE=OFF,XRETURN=OFF     ZAP THE TONS OF MESSAGES
MAIN     CSECT
         PRINT NOGEN
         XSAVE
         L     15,=V(SUB1)         GET @ WHERE ENTAB IS
         XSNAP T=NO,LABEL='THIS IS ENTAB',STORAGE=(*0(15),*40(15))
         SPACE 2
         OCALL SUB1
         OCALL SUB2
         OCALL SUB1CE              ENTRY POINT OF CSECT SUB1C
         OCALL SUB2C
         XSNAP T=NO,LABEL='A-V-ADCONS FOR SAME',STORAGE=(MAIN1,MAIN2)
         SPACE 2
         XRETURN SA=*
MAIN1    DS    0D                  BEGINNING ADDRESS FOR LITERAL DUMP
         LTORG
MAIN2    EQU   *                   ENDING ADDRESS FOR LITERAL DUMP
         ORG   MAIN+X'1000'        MAKE SIZE NIZE
         TITLE 'CSECTS SUB1, SUB2, SUB1A, SUB1B'
SUB1     CSECT
         XSAVE
         SPACE 1
         CALL  MSUB1               CALL LOW LEVEL ROUTINE
         CALL  MSUB2               "
         CALL  MSUB3               "
         CALL  MSUB4
         CALL  SUB1A
         CALL  SUB1B
         CALL  SUB1C
         SPACE 1
         XRETURN SA=*
         ORG   SUB1+X'2000'
         SPACE 3
SUB2     CSECT
         XSAVE
         SPACE 1
         CALL  MSUB1
         SPACE 1
         CALL  SUB2A
         CALL  SUB2B
         CALL  SUB2C
         SPACE 1
         XRETURN SA=*
         ORG   SUB2+X'3000'
         SPACE 2
SUB1A    CSECT
         XSAVE
         SPACE 1
         XRETURN SA=*
         ORG   SUB1A+X'4000'
         SPACE 1
SUB1B    CSECT
         XSAVE
         SPACE 1
         CALL  SUB1C
         CALL  SUB1D
         SPACE 1
```

```
         XRETURN SA=*
         ORG   SUB1B+X'5000'
         TITLE 'CSECTS  SUB1C,D   SUB2A,B,C'
SUB1C    CSECT
         ENTRY SUB1CE
         XSAVE
         SPACE 1
         CALL  MSUB2
         SPACE 1
         XRETURN
SUB1CE   XSAVE
         SPACE 1
         CALL  MSUB2
         SPACE 1
         XRETURN SA=*
         ORG   SUB1C+X'1000'
         SPACE 2
SUB1D    CSECT
         XSAVE
         XRETURN SA=*
         ORG   SUB1D+X'1000'
         SPACE 3
*             SUB2A,B,C - SUBROUTINES CALLED BY SUB2 OR MAIN.
SUB2A    CSECT
         XSAVE
         XRETURN SA=*
         ORG   SUB2A+X'3000'
         SPACE 2
SUB2B    CSECT
         XSAVE
         XRETURN SA=*
         ORG   SUB2B+X'2000'
         SPACE 2
SUB2C    CSECT
         XSAVE
         XRETURN SA=*
         ORG   SUB2C+X'2000'
         TITLE 'MSUB1,2,3,4 CSECTS - CALLED FROM ALL OVER'
MSUB1    CSECT
         XSAVE
         SPACE 1
         CALL  MSUB3
         CALL  MSUB4
         SPACE 1
         XRETURN SA=*
         ORG   MSUB1+X'3000'
         SPACE 2
MSUB2    CSECT
         XSAVE
         SPACE 1
         CALL  MSUB3
         CALL  MSUB4
         SPACE 1
         XRETURN SA=*
         ORG   MSUB2+X'4000'
         SPACE 2
MSUB3    CSECT
         XSAVE
         XRETURN SA=*
         ORG   MSUB3+X'1000'
         SPACE 2
```

```
MSUB4    CSECT
         XSAVE
         XRETURN SA=*
         ORG   MSUB4+X'1000'
         END
//*
//*            THE FOLLOWING STMTS ARE JCL KLUDGES REQUIRED TO
//*       GET EVERYTHING IN THE RIGHT PLACE, GIVEN THE WAY THE CAT.
//*       PROCEDURES ARE SET UP.
//*
//OBJECT.SYSLIN DD *
         INCLUDE OBJ               GET FOR MOD0 (NO OVERLAY
         NAME    MOD0              (OVERLAY OPT WILL BE NULLED OUT)
         INCLUDE OBJ               GET ANOTHER COPY OF THE OBJECT
         OVERLAY ALPHA             DEFINE BEGINNING
         INSERT  SUB1,SUB1A,SUB1B,SUB1C,SUB1D   SUB1&ITS SUBRS
         OVERLAY ALPHA             BACK TO SAME PLACE AS ABOVE
         INSERT  SUB2,SUB2A,SUB2B,SUB2C         SUB2& ITS SUBRS
         NAME    MOD1              3-SEGMENT OVERLAY MODULE
         INCLUDE OBJ               COPY FOR 10-SEGMENT OVERLAY
         OVERLAY ALPHA             ORIGIN AS BEFORE
         INSERT  SUB1              POSITION SUB1 AFTER ROOT
         OVERLAY BETA1             ORIGIN FOR SUB1A,B
         INSERT  SUB1A             PUT SUB1A AT END OF SUB1
         OVERLAY BETA1             BACK TO END OF SUB1
         INSERT  SUB1B             PUT SUB1B AT END OF SUB1 ALSO
         OVERLAY CHI1              END OF SUB1B
         INSERT  SUB1C             PUT SUB1C AT END OF SUB1B
         OVERLAY CHI1              BACK TO END OF SUB1B
         INSERT  SUB1D             PUT SUB1D BEGIN LIKE SUB1C
         OVERLAY ALPHA             BACK TO WHERE SUB1 BEGAN
         INSERT  SUB2              BEGIN SUB2 WHERE SUB1 DID
         OVERLAY BETA2             ORIGIN FOR SUB2'S SUBROUTINES
         INSERT  SUB2A             PUT SUB2A AT END OF SUB2
         OVERLAY BETA2             BACK EVEN WITH SUB2A
         INSERT  SUB2B             PUT SUB2B IN SAME PLACE AS SUB2A
         OVERLAY BETA2             ONCE MORE
         INSERT  SUB2C             PUT C IN SAME AS A AND B
         NAME    MOD2              10-SEGMENT SINGLE REGION MODULE
         INCLUDE OBJ               COPY FOR 3-REGION OVERLAY
         OVERLAY ALPHA             ORIGIN AS BEFORE
         INSERT  SUB1              POSITION SUB1 AFTER ROOT
         OVERLAY BETA1             ORIGIN FOR SUB1A,B
         INSERT  SUB1A             PUT SUB1A AT END OF SUB1
         OVERLAY BETA1             BACK TO END OF SUB1
         INSERT  SUB1B             PUT SUB1B AT END OF SUB1 ALSO
         OVERLAY CHI1              END OF SUB1B
         INSERT  SUB1C             PUT SUB1C AT END OF SUB1B
         OVERLAY CHI1              BACK TO END OF SUB1B
         INSERT  SUB1D             PUT SUB1D BEGIN LIKE SUB1C
         OVERLAY ALPHA             BACK TO WHERE SUB1 BEGAN
         INSERT  SUB2              BEGIN SUB2 WHERE SUB1 DID
         OVERLAY BETA2             ORIGIN FOR SUB2'S SUBROUTINES
         INSERT  SUB2A             PUT SUB2A AT END OF SUB2
         OVERLAY BETA2             BACK EVEN WITH SUB2A
         INSERT  SUB2B             PUT SUB2B IN SAME PLACE AS SUB2A
         OVERLAY BETA2             ONCE MORE
         INSERT  SUB2C             PUT C IN SAME AS A AND B
         OVERLAY REGION2(REGION)   NEW REGION
         INSERT  MSUB1             TAKE OUT OF ROOT SEGMENT
         OVERLAY REGION2           REPOSITION (DON'T NEED (REGION)
```

```
         INSERT  MSUB2             PUT MSUB2 SAME AS MSUB1
         OVERLAY REGION3(REGION)   NEED ANOTHER REGION FOR MSUB3,4
         INSERT  MSUB3             PUT IN THIS REGION
         OVERLAY REGION3           BACK TO BEGINNING OF REGION
         INSERT  MSUB4             PUT MSUB4 SAME AS MSUB3
         NAME    MOD3              3-REGION, 14-SEGMENT OVERLAY
//*
//*            THE FOLLOWING SECTIONS MODIFY THE SYSLMOD CORRECTLY AND
//*      DEFINE &&STUPID SO THAT IT CAN BE INCLUDED.
//*
//OBJECT.SYSLMOD DD VOL=REF=*.JOBLIB,DSN=&&LOADMOD,DISP=(OLD,PASS)
//OBJECT.OBJ DD DSN=&&STUPID,DISP=(OLD,PASS)
//*
//*            NOW EXECUTE THE LOAD MODULES PRODUCED BY ASMLINK.
//*            STEPS STEP0-STE3 FOR MODULES MOD0-MOD3
//*            THESE ARE IN OUR JOBLIB, SO WE CAN JUST CALL THEM OUT
//*            WITH EXEC STMTS.
//*
//STEP0 EXEC PGM=MOD0              NO OVERLAY
//XSNAPOUT DD SYSOUT=A
//*
//STEP1 EXEC PGM=MOD1              3-SEGMENT OVERLAY
//XSNAPOUT DD SYSOUT=A
//*
//STEP2 EXEC PGM=MOD2              10-SEGMENT, 1-REGION OVERLAY
//XSNAPOUT DD SYSOUT=A
//*
//STEP3 EXEC PGM=MOD3              14-SEGMENT, 3-REGION OVERLAY
//XSNAPOUT DD SYSOUT=A
//*
//*            END OF EXAMPLE ON OVERLAY PROGRAMS
//*
```

```
//*
//* 1. SAMPLE PROGRAM - EXAMPLE OF IEBPTPCH, ON PDS, SHOWING HOW TO
//* PRINT SEVERAL MACROS FROM SYS1.MACLIB (CALL,SAVE,RETURN)
//*
//*
//*      PRINT SEVERAL MACROS, USING PTPCH
//*
//PRINTMAC EXEC PGM=IEBPTPCH
//SYSPRINT DD SYSOUT=A
//SYSUT2 DD SYSOUT=A
//SYSUT1 DD DSN=SYS1.MACLIB,DISP=SHR
//*         DSN=CMACLIB,DISP=SHR     FOR PSU LOCAL MACROS
//SYSIN  DD *
  PRINT TYPORG=PO,MAXFLDS=6,MAXNAME=3
  TITLE ITEM=('MACRO LISTINGS',40)
  TITLE ITEM=(' ',40)
  MEMBER NAME=CALL
  RECORD FIELD=(72,,,10),FIELD=(8,73,,90)
  MEMBER NAME=SAVE
  RECORD FIELD=(72,,,10)
  MEMBER NAME=RETURN
  RECORD FIELD=(72,,,10)
```

```
//*
//*            SAMPLE PROGRAM - INPUT/OUTPUT - QSAM
//*      1. READ FROM CARD READER, ECHO TO PRINTER, WRITE ON DISK.
//*      2. READ FROM DISK, WRITE TO PRINTER.
//*      ILLUSTRATES GET-MOVE, PUT MOVE, GET-LOCATE, PUT-LOCATE.
//*
//STEP1 EXEC ASGCG,PARM.SOURCE='NOESD,NOXREF',PARM.DATA='MAP'
//SOURCE.INPUT DD *
         TITLE 'QSAM SAMPLE PROGRAM'
IOTESTQS CSECT
         PRINT NOGEN
         EQUREGS
         XSAVE TR=NO               NO TRACING TO BE DONE
         PRINT GEN                 GEN SO CAN SEE OPENS, ETC
         SPACE 1
*        USE THE OPEN MACRO TO INITIALIZE FOR INPUT/OUTPUT.
         SPACE 1
         OPEN  (IOCRDDCB,(INPUT),IOPRTDCB,(OUTPUT))
*              THE ABOVE GIVES 2 DCB NAMES AND DIRECTIONS FOR I/O.
         OPEN  (IODSKDCB,(OUTPUT)) OPEN WHERE WE WILL PUT DATA
         SPACE 1
*              PRINT A TITLE BEFORE DOING ANYTHING ELSE
         PUT   IOPRTDCB,IOTITLE1   PUT THE MESSAGE - PUT-MOVE FORM
         SPACE 2
*              THE FOLLOWING LOOP READS A CARD FROM CARD READER,
*              PRINTS IT AS AN ECHO CHECK, THEN WRITES IT ON DISK.
*              CONTROL IS TRANSFERRED TO IOEOF1 WHENEVER THERE ARE
*              NO MORE CARDS LEFT.
IOREAD   GET   IOCRDDCB,IOCARD     MOVE NEXT CARD TO IOCARD#
         PUT   IOPRTDCB,IOCARD     PRINT IT (IT HAS CARRIAGE CONTROL)
         SPACE 1
*              NEXT PUT ILLUSTRATES PUT-LOCATE.  OS/360 RETURNS IN
*              R1 THE ADDRESS OF NEXT BUFFER IN WHICH TO PLACE OUTPUT
*              CARD.  WE MOVE IT THERE OURSELVES.
         PUT   IODSKDCB            2ND OP OMITTED SINCE PL
         MVC   0(80,R1),IOCARD     MOVE THE CARD THERE
         B     IOREAD              GO BACK FOR MORE
         SPACE 1
IOEOF1   EQU   *                   BRANCH HERE - SEE EODAD=IOEOF1
*              USE CLOSE MACRO TO CLEAN UP AT END OF PROCESSING.
*              WILL ALSO WRITE OUT LAST BUFFER CREATED.
         CLOSE (IOCRDDCB,,IODSKDCB)  NOTE EXTRA COMMA REQUIRED.
*              WE NOW REOPEN IODSKDCB FOR INPUT THIS TIME.  NOTE THAT
*              THE NAME ONLY IS INCLUDED.  IF THE OPTION IS OMITTED,
*              INPUT IS IMPLIED.
         OPEN  (IODSKDCB)          OPEN IT FOR INPUT
         PUT   IOPRTDCB,IOTITLE2   PRINT SECOND TITLE
         SPACE 1
*              AT THIS POINT, WE CLOSE PRINT DCB, DYNAMICALLY CHANGE
*              LRECL AND BLKSIZE TO 80, SO WE DON'T HAVE TO PAD WITH
*              BLANKS THIS TIME, THEN RE-OPEN.
         CLOSE (IOPRTDCB)          CLOSE IT, FLUSH BUFFERS
         MVC   IOPRTDCB+X'3E'(2),=H'80'  MAKE BLKSIZE 80
         MVC   IOPRTDCB+X'52'(2),=H'80'  MAKE LRECL 80 ALSO
         OPEN  (IOPRTDCB,(OUTPUT)) REOPEN NOW
*              THE ABOVE KLUDGE ONLY NECESSARY BECAUSE LAZY PROGRAMMER
*              DOESN'T WANT TO PAD WITH BLANKS AND MOVE CARDS AROUND.
         SPACE 1
         SPACE 1
*              FOLLOWING LOOP READS THE RECORDS BACK FROM DISK, THEN
*              PRINTS THEM OUT AGAIN.
```

```
IOREAD2  EQU   *                  LOOP HEAD FOR READING
         GET   IODSKDCB           GET-LOCATE - R1= @ NEXT RECORD
         LR    R0,R1              MOVE TO PLACE BEST FOR PUT
         PUT   IOPRTDCB,(0)       ILLUSTRATE REGISTER FOR FOR PUT-MOVE
         B     IOREAD2            LOOP UNTIL DONE
         SPACE 1
IOEOF2   EQU   *                  BRANCH HERE - SEE EODAD=IOEOF2
         CLOSE (IODSKDCB)         DONE WITH IT - CLOSE IT
         LA    R2,IOTITLE3        PUT IN @ TITLE AREA
         PUT   IOPRTDCB,(2)       ILLUSTRATE REGISTER FORM
         CLOSE (IOPRTDCB)         MAKE SURE LAST PRINTED -DONE
         SPACE 1
         PRINT NOGEN
         XRETURN SA=*,TR=NO
         SPACE 1
IOCARD   DS    0D,CL80            80 BYTES, D ALIGNED
         DC    CL53' '            PAD TO 133 BYTES FOR PRINTING
IOTITLE1 DC    CL133'1***** ECHO-CHECK OF INPUT CARDS BELOW *****'
IOTITLE2 DC    CL133'0***** FIRST PASS DONE, RECORDS FROM DISK FOLLOW  #
               BELOW *****'
IOTITLE3 DC    CL133'0***** END OF EXAMPLE - LAST LINE PRINTED *****'
         SPACE 1
         PRINT GEN
*              DATA CONTROL BLOCKS FOLLOW.
         SPACE 1
*              DATA CONTROL BLOCK FOR THE CARD READER
IOCRDDCB DCB   DDNAME=CARDS,       JCL DDNAME DEFINING THE DATA        #
               DSORG=PS,           PHYSICAL SEQUENTIAL DEVICE          #
               MACRF=GM,           GET-MOVE MACRO FORM USED            #
               EODAD=IOEOF1,       END-OF-DATA EXIT ADDRESS TO GOTO    #
               RECFM=F,            RECORD FORMAT - FIXED -NOT BLOCKED  #
               LRECL=80,           LOGICAL RECORD LENGTH = CARD SIZE   #
               BLKSIZE=80          BLOCK SIZE = CARD SIZE, UNBLOCKED
*              NOTE, THE LAST 3 PARAMETERS COULD BE LEFT OFF HERE
         SPACE 1
*              DCB FOR THE PRINTER, PSECIFYING ALL NEEDED.
IOPRTDCB DCB   DDNAME=PRINT,       JCL DDNAME FOR THE PRINTER          #
               DSORG=PS,           DATA SET ORGANIZATION-PHYS SEQ.     #
               MACRF=PM,           MACRO FORMAT IS PUT-MOVE            #
               RECFM=FA,           RECORD FORMA -FIXED, HAS CARR CONTRL#
               LRECL=133,          LOGICAL RECORD LENGTH- PRINT LINE   #
               BLKSIZE=133         BLOCK SIZE SAME AS LRECL, UNBLOCKED
         SPACE 1
*              NOW HAVE DCB FOR DISK, WILL SPECIFY SOME DCB VALUES IN
*              THE JCL CARDS (DD CARD FOR WORKDISK).
IODSKDCB DCB   DDNAME=WORKDISK,    JCL DDNAME USED                     #
               DSORG=PS,           ALSO PHYSICAL SEQUENTIAL DATASET    #
               MACRF=(GL,PL),      BOTH GET- AND PUT- LOCATE MACRO FORM#
               EODAD=IOEOF2        FOR WHEN END-OF-DATA
         END
//*
//*             NOW EXECUTE THE PROGRAM
//*
//DATA.WORKDISK DD UNIT=SYSDA,     GIVE ME SPACE ON ANY DASD           #
//              SPACE=(160,(20)),  SPACE FOR 20 RECORDS OF 2 CARDS     #
//              DCB=(RECFM=FB,LRECL=80,BLKSIZE=160), SUPPLY DCB        #
//              DSNAME=&&TEMP,     GIVE DATA SET A NAME (UNNEC HERE)   #
//              DISP=(NEW,DELETE)  CREATE IT NOW, GET RID WHEN DONE
//DATA.PRINT DD SYSOUT=A           PUT THIS ON A PRINTER SOMEWHERE
//DATA.CARDS DD *                  WE HAVE A SET OF CARDS FOLLOWING
0 FIRST INPUT TEST CARD
```

  SECOND INPUT TEST CARD - GOES IN BLOCK WITH FIRST.
0 THIRD INPUT TEST CARD
  FOURTH INPUT TEST CARD - GOES IN BLOCK WITH THIRD
0 FIFTH INPUT TEST CARD - WILL BE IN TRUNCATED BLOCK BY ITSELF.

```
//*
//*      THIS JOB WILL RUN WITH   TIME = 25 SECONDS
//*                               RECORDS = 600
//*
// EXEC ASGCG
//SYSIN DD *
*
*        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*        THE PURPOSE OF THIS PROGRAM IS TO DEMONSTRATE THE USE OF
*        GETMAIN AND FREEMAIN WITH THE E CONVENTION
*        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*
RCALL    DSECT
SAVEAREA DS    18F
NUMBER   DS    F
         PRINT NOGEN
         EQUREGS
*
*        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*        THE MAIN PROGRAM ONLY CALLS THE RECURSIVE SUBROUTINE RECRSIVE
*        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*
MAIN     CSECT
         XSAVE
         CALL RECRSIVE,(NMBR)
         XRETURN SA=*
NMBR     DC    F'5'
*
*        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*        THE CSECT RECRSIVE IS A RECURSIVE CSECT THE SAVEAREA IS OBTAIN
*        FROM THE OPERATING SYSTEM BY A GETMAIN  THEN THE PARM HAS 1
*        SUBTRACTED FROM IT AND IF IT IS 0 THEN WE OUTPUT THE SAVEAREAS
*        IF NOT 0 THEN WE CALL RECRSIVE THE LAST WORD THAT IS OBTAINED
*        HAS THE VALUE FOR THE NEXT CALL TO RECRSIVE
*        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*
RECRSIVE CSECT
         USING *,15
         LR    R11,R15             MAKE A COPY OF ENTRY ADDRESS
*
*        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*        FIRST WE GET A COPY OF THE ADDRESS OF THE PARM LIST THEN
*        OBATIN THE DESIRED STORAGE FROM THE OPERATING SYSTEM
*        NEXT SET R2 FOR A DSECT THEN DO AN XSAVE  WITH SA = SAVEAREA
*        NEXT TEST THE PARM TO SEE IF IT IS 0
*        .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .  .
*
         LR    R3,R1               MAKE A COPY OF PARM LIST  ADDRESS
         PRINT GEN
         GETMAIN EU,LV=80,A=ADDRESS,SP=1
         PRINT NOGEN
         LR    R15,R11             SET R15 FOR USING IN XSAVE
         L     R2,ADDRESS          SET R2 TO BEGINNING OF AREA OBTAINED
         XC    0(80,R2),0(R2)      CLEAN OUT THE AREA
*                                   THIS WILL BE USED WITH DSECT
         USING RCALL,2
         STM   R14,R12,12(R13)     SAVE REGS OF CALLING PGM
         LA    R5,SAVEAREA         GET ADD OF CALLED PGM SAVEAREA
         ST    R5,8(R13)           SET LINK FOR LSA OF CALLING PGM
         ST    R13,4(R5)           SET HSA OF CALLED PGM
         LR    R13,R5              SET R13 TO CALLED PGM SAVEAREA
```

```
         BALR  R12,R0              SET R12 FOR BASE FEG
         DROP  R15
         USING *,R12
         L     R4,0(R3)           GET ADDRESS OF PARM
         L     R5,0(R4)           GET PARM
         XDECO R5,N               CONVERT TO OUTPUT FORM
         XPRNT OUT,52
         LA    R6,1               SET R6 TO 1 FOR SUBTACTION
         SR    R5,R6              DECREASE RECURSIVE COUNTER BY ONE
         BZ    OUTPUT             IF RECURSIVE COUNTER = 0 THEN OUTPUT
*
*        . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
*        AT THIS POINT WE HAVE NOT DONE ENOUGH RECURSIVE CALLS SO CALL
*        RECRSIVE AGAIN WITH THE NEW PARM IN NUMBER WHICH WAS OBTAINED
*        FROM THE OPERATING SYSTEM
*        . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
*
         ST    R5,NUMBER          SET VALUE FOR NEXT CALL TO RECRSIVE
         LA    R9,72(R2)          PUT ADD OF NUMBER IN R9
         ST    R9,76(R2)          PUT ADD OF PARM IN PARM LIST
         LA    R1,76(R2)          PUT ADDRESS OF PARM LIST IN R1
         CALL  RECRSIVE
         B     DONE               GO TO RETURN FROM THIS CALL
*
*        . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
*        AT THIS POINT WE HAVE DONE 5 RECURSIVE CALLS TO RE RSIVE
*        SO OUTPUT THE SAVEAREAS AND AND RETURN
*        . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
*
OUTPUT   LA    R6,5               SET R6 TO 5 FOR COUNTER ON LOOP FOR
*                                 OUTPUTTING THE SAVEAREAS
         LR    R11,R13            GET R13 TO R11 FOR LOOP
         LA    R10,80(R11)        GET THE END OF THE AREA OBTAINED
         XPRNT HEADING,47
LOOP     XSNAP STORAGE=(*0(11),*0(10)),T=NOREGS
         L     R11,4(R11)         GET ADD OF HSA
         LA    R10,80(R11)        GET ADD OF END OF AREA
         BCT   R6,LOOP            RETURN FOR NEXT SAVEAREA
         PRINT GEN
*
*        . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
*        AT THIS POINT WE HAVE FINISHED WITH THIS VERSION OF THE CSECT
*        SO FREE THE STORAGE OBTAINED WITH A FREEMAIN AND RETURN TO
*        CALLING PROGRAM
*        . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
*
         FREEMAIN E,LV=80,A=ADDRESS,SP=1
DONE     ST    R2,ADDRESS         SET ADD FOR FREEMAIN
         PRINT NOGEN
         LA    R5,1(R5)           INCREASE R5 FOR OUTPUT
         XDECO R5,NN              PLACE IN OUTPUT STREAM
         XPRNT IN,52
         L     R13,4(R13)         GET ADD OF HIGHER SAVE AREA
         LM    R14,R12,12(R13)    RESTORE THE REGISTERS
         BR    R14                RETURN TO CALLING PROGRAM
HEADING  DC    C'0THE SAVEAREAS AND NUMBERS ARE OUTPUTTED BELOW '
ADDRESS  DC    F'0'
OUT      DC    CL40' RECURSIVE CALLED WITH VALUE'
N        DC    3F'0'
IN       DC    C' RETURNING FORM RECRSIVE WITH NUMBER ='
NN       DC    3F'0'
```

```
         LTORG
         END
/*
/*LOG
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.XSNAPOUT DD UNIT=AFF=FT06F001
```

```
//*
//*      THIS JOB WILL RUN WITH   TIME = 35 SECONDS
//*                               RECORDS = 1000
//*
//*
// EXEC ASGCG
//SOURCE.INPUT DD *
*
*        .         .         .         .         .         .         .
*        THE PURPOSE OF THIS PROGRAM IS TO DEMONSTRATE THE FOLLOWING
*        MACROS:
*                  SPIE
*                  STAE
*        .         .         .         .         .         .         .
*
*
*        .         .         .         .         .         .         .
*        THE BASIC FLOW OF THIS PROGRAM IS AS FOLLOWS:
*        1         FIRST ISSUE A SPIE MACRO FOR ABEND CODES 1 - 5.
*        THEN CAUSE AN ABEND S0C1 AND INTERCEPT IT WITH A SPIE.
*        2         RETURN CONTROL TO THE NEXT INSTRUCTION  FROM THE
*        SPIE EXIT ROUTINE.
*        3         CANCEL THE SPIE MACRO.
*        4         ISSUE A STAE MACRO.  THEN CAUSE A SOC6 AND
*        INTERCEPT THE ABEND WITH THE STAE EXIT ROUTINE.
*        THEN ALLOW THE ABEND TO CONTINUE.
*        .         .         .         .         .         .         .
*
*                 PROGRAM INTERRUPTION CONTROL AREA
*
*   DISPLACEMENT
*
*   BYTES
*   0         1         2         3         4         5         6
*   *---------------------------------------------------------*
*   ×         ×         ×         ×         ×         ×         ×
*   ×   0000  ×PROGRAM  × EXIT ROUTINE      ×INTERRUPTION       ×
*   ×         ×  MASK   ×         ADDRESS   ×      TYPE         ×
*   ×         ×         ×         ×         ×         ×         ×
*   *---------------------------------------------------------
        EJECT
*                        PROGRAM INTERUPTION ELEMENT
*
*        DISPLACEMENT
*            BYTES 0         1         2         3
*                  *-----------------------------------------*
*        0         ×RESERVED ×        PICA ADDRESS          ×
*                  *-----------------------------------------*
*        4         ×OLD PROGRAM        ×INTERUPTION CODES  ×
*                  ×STATUS WORD        *-------------------×
*                  ×                                       ×
*                  *-----------------------------------------*
*        12        ×              REGISTER 14              ×
*                  *-----------------------------------------*
*        16        ×              REGISTER 15              ×
*                  *-----------------------------------------*
*        20        ×              REGISTER 0               ×
*                  *-----------------------------------------*
*        24        ×              REGISTER 1               ×
*                  *-----------------------------------------*
*        28        ×              REGISTER 2               ×
```

```
*                *-------------------------------------*
         EJECT
         PRINT NOGEN
         EQUREGS
MAIN     CSECT
         XSAVE SA=NO
*
*        .         .         .         .         .         .         .
*        THE SPIE MACRO BELOW INDICATES THAT IF AN ABEND SOC1 THRU
*        ABEND SOC5 OCCURS THAT FIX IS TO BE GIVEN CONTROL.
*        .         .         .         .         .         .         .
*
         PRINT GEN
         SPIE FIX,((1,5))
         PRINT NOGEN
         ST    R1,HOLD             SAVE ADDRESS OF PREVIOUS PICA
         DC    F'0'
         XPRNT MHEAD,80
         L     R5,HOLD
*
*        .         .         .         .         .         .         .
*        THE SPIE MECRO IS THE EXECUTE FORM IT CANCELS THE PREVIOUS
*        SPIE MACRO.
*        .         .         .         .         .         .         .
*
         PRINT GEN
         SPIE  MF=(E,(5))
*
*        .         .         .         .         .         .         .
*        THE STAE MACRO BELOW INDICATES THA THE FIXSTAE ROUTINE IS
*        TO BE GIVEN CONTROL ON AN ABEND AND TO ALLOW THE I/O TO
*        CONTINUE EVEN THOUGH THE ABEND HAS CCCURED. ALSO CREATE THIS
*        STEA AREA NOT OVERLAY ANY PREVIOUS
*        .         .         .         .         .         .         .
*
         STAE  FIXSTAE,CT,PURGE=NONE
         LA    R3,HALFWORD         GET ADDRESS OFHALFWORD
         L     R2,0(R3)            IMPROPER ALIGNEMNT CAUSE ABEND
         PRINT NOGEN
         XPRNT MHEAD1,80
         PRINT GEN
         STAE  0
         PRINT NOGEN
         XRETURN SA=NO
         DROP  12
         USING *,15
FIX      STM   R0,R15,SAVE
         L     R2,8(R1)            GET IL CC AND NEXT INSTRUCTION
*                                   ADDRESS
         LA    R11,255             SET R11 TO HEX 000000FF
         LR    R10,R11             PUT HEX FF IN R10
         SLL   R11,8               MOVE OVER 1 BYTE
         OR    R10,R11             MAKE R10 HEX 0000FFFF
         SLL   R11,8               MOVE OBER 1 MORE BYTE
         OR    R10,R11             MAKE R10 HEX 00FFFFFF
         NR    R10,R2              GET INSTRUCTION ADDRESS
         SR    R9,R9               ZERO R9 FOR IC
         IC    R9,0(R10)           GET OPCODE FOR NEXT INSTRUCTION
         LA    R8,64               SET TO HEX 40
         CR    R9,R8               CHECK FOR RR INSTRUCTION
         BM    RR                  GO TO SET NEW PSW
```

```
         LA    R8,192 SET R8 TO CHECK FOR RX OR RS
         BM    RX                  GO TO SET PSW
         LA    R7,6                GET INSTRUCTION LENGTH
         B     ILCSET              GO TO SET IL
RR       LA    R7,2                SET ILC FOR RR
         B     ILCSET              GO TO SET IL
RX       LA    R7,4                STE IL OF 4
ILCSET   AR    R10,R7              GET NEW ADDRESS FOR PSW
         SLL   R7,29               SET R7 TO IL CODE
         LA    R8,63               SET R8 TO HEX 0000003F
         SLL   R8,24               SET R8 TO 3F00000000
         NR    R8,R2               GET CC AND PROGRAM MASK
         OR    R10,R7              GET IL CODE IN NEW PSW
         OR    R10,R8              NOW WE HAVE NEW PSW
         ST    R10,8(R1)           SET NEW PSW
         PRINT NOGEN
         XSNAP T=NOREGS,STORAGE=(*0(1),*36(1)),                        X
               LABEL='PROGRAM INTERUPT ELEMENT IN SPIE'
         L     R2,0(1)             GET PICA ADDRESS
         XSNAP T=NOREGS,STORAGE=(*0(2),*6(2)),                         X
               LABEL='PICA FOR SPIE MACRO'
         LM    R0,R15,SAVE         GET ADDRESSES TO RETURN
         BR    R14                 RETURN TO CONTROL
         DROP  15
FIXSTAE  XSAVE SA=NO
         XSNAP STORAGE=(*0(1),*104(1)),T=NOREGS,                       X
               LABEL='THIS IS THE 104 BYTE WORKAREA PROVIDED BY STAE'
         LA    R15,0
MMDONE   L   R14,12(13)            RESOTRE REGISTER 14 FROM SAVEAREA
         LM    R0,R12,20(R13)          RESTORE REG 0 THRU 12
         BR    R14                 RETURN TO PPERATING SYSTEM
MHEAD1   DC    CL80'0STAE MACRO HAS ISSUED AND RETURNED CONTROL'
SAVE     DC 18F'0'
HOLD     DC    F'0'
         DS    0F
         DC    H'0'
HALFWORD DC    C'NONO'
MHEAD    DC    CL80'0THE INTERUPT HAS OCCURED AND FIX CALLED ON SPIE'
         END
/*
/*LOG
//DATA.SYSUDUMP DD SYSOUT=A
//DATA.XSNAPOUT DD SYSOUT=A
```

```
//*
//*      THIS JOB WILL RUN WITH   TIME= 35 SECONDS
//*                               RECORDS = 600
//*
// EXEC ASGCG
//SOURCE.INPUT DD *
/*LOG
*
*        .         .         .         .         .         .         .
*        THE PUT
*        THE PURPOSE OF THIS PROGRAM IS TO DEMONSTRATE THE FOLLOWING
*        MACROS:
*                  TIME
*                  TTIMER
*                  STIMER
*        .         .         .         .         .         .         .
*
*
*        .         .         .         .         .         .         .
*        THE BASIC FLOW OF THIS PROGRAM IS AS FOLLOWS:
*        1         GET THE CURRENT TIME AND DATA USING THE TIME MACRO
*        THEN CONVERT IT TO OUTPUT FORM AND PRINT IT OUT.
*        2         THEN DO 10,000 CALLS TO A RANDOM NUMBER GENERATOR
*        WHICH USES THE TIME MACRO IN BINARY TO SET THE BASE.
*        THEN OUTPUT THE RESULTS OF THE CALLS TO THE RAMDOM NUMBER
*        GENERATOR
*        3         ISSUE A STIMER MACRO FOR 5 SECONDS.  THEN DO
*        1000 BCT TO *. TO DETERMINE THE NUMBER OF MICRO SECONDS THAT
*        A BCT INSTRUCTION TAKES.  THEN OUTPUT THE RESULT.
*        4         THEN USE AN STIMER WITH AN EXIT ROUTINE TO CHECK
*        A LOOP.  THAT IS ISSUE AN STIMER FOR 1.04 SECONDS WITH AN
*        EXIT ADDRESS.  WHEN THE EXIT ADDRESS GETS CONTROL IT CAUSES
*        THE LOOP TO COME TO A HALT AND OUTPUTS THAT THE LOOP TAKES
*        LONGER THAN 1.04 SECONDS.  THEN CANCEL THE STIMER WITH A
*        TTIMER MACR.
*        .         .         .         .         .         .         .
*
         EJECT
         PRINT NOGEN
         EQUREGS
MAIN     CSECT
         XSAVE
*
*        .         .         .         .         .         .         .
*        WHEN TIME IN DEC IS ISSUED IT RETURNS THE DATE IN REG 1
*        IN THIS FORM 00YYDDDF WHERE YY IS THE LAST TWO DIGITS OF THE
*        YEAR AND DDD IS THE JULIAN DATA.  THEN LEFT JUSTIFY THIS
*        DATE AND UNPACK THE RESULT AND PLACE IN OUTPUT FOR PRINTING.
*        .         .         .         .         .         .         .
*
         PRINT GEN
         TIME  DEC
         PRINT NOGEN
         SLL   R1,8               LEFT JUSTIFY THE DATA
         ST    R1,WORD            PUT DAY AND YEAR IN CORE FOR UNPK
         UNPK  OUT(5),WORD(3)     GET OUTPUT FORM
         MVC   OUTPUT1(3),OUT+2   PUT YEAR IN OUTPUT
         MVC   OUTPUT(2),OUT      PUT JULIAN DATE IN OUTPTU
*
*        .         .         .         .         .         .         .
*        WHEN TIME IS ISSUED WITH DEC IT RETURNS IN R0 THE TIME IN
```

```
*        THE FOLLOWING FORM  HHMMSSTH WHERE HH IS THE HOUR ON A 24
*        HOUR CLOCK,  WHERE MM IS THE MINUTES, HHERE SS IS THE SECONDS
*        WHERE THE T IS THE TENTHS OF SECONDS, AND WHERE THE SINGLE H
*        IS THE HUNDREDS OF SECONDS.
*        DISCARD THE T AND SINGLE H FIELD AND THEN PLACE THE HH MM SS
*        IN CORE AND UNPACK IT AND THEN OUTPUT THE TIME AND DATE.
*        .         .         .         .         .         .         .
*
         LA    R11,240            SET R10 TO JEX 000000F0
         OR    R11,R0             PLACE F IN BITS 24 TO 27 R11
         SRL   R11,4              RIGHT JUSTIFY THE TIME
         ST    R11,WORD           PLACE TIME IN CORE FOR UNPK
         UNPK  OUT(6),WORD(4)     CHANGE TO OUTPUT FORM
         MVC   OUTPUT2(2),OUT     PLACE HR IN OUTPUT
         MVC   OUTPUT3(2),OUT+2   PLACE MINUTES IN OUTPUT
         MVC   OUTPUT4(2),OUT+4   PLACE SEC IN OUTPUT
         XPRNT OUTPUT5,34
         LA    R11,4095           SET R10 TO 4095
         LA    R11,4095(R11)      SET R11 TO 8190
         LA    R11,1810(R11)      SET R11 TO 10,000
         LA    R9,DONE
*
*        .         .         .         .         .         .         .
*        THE TIME MACRO IN THE RANDOM NUMBER GENERATOR RETURNS THE
*        TIME IN REG 0 IN BINARY FORM.  THEN IT IS STORED IN SAVEODD
*        TO BE USED FOR THE BASIS FOR RANDOM NUMBER GENERATOR.
*        THEN OUTPUT THE RESULTS OF 10,000 THROWS OF A TEN SIDED DIE.
*        .         .         .         .         .         .         .
*
LOOP     CALL  IAND,(TEN,MONE)
         LR    R10,R0             SET R10 TO R0
         BCTR  R10,R0             SUBTRACT 1 FROM R10
         SLL   R10,2              MULTIPLY R10 BY FOUR
         AR    R10,R9             GET ADD OF NUMBER TO INCREASE
         L     R8,0(R10)          GET LAST VALUE
         LA    R8,1(R8)           INCREMENT COUNT BY ONE
         ST    R8,0(R10)          PLACE IN COUNTER
         BCT   R11,LOOP           RETURN FOR NEXT CALL
         XPRNT OUTT
         XPRNT OUTTT
         LA    R10,10             SET R10 TO NUMBR OF SIDES ON DIE
         LA    R11,OUTTTT+1
LOOP1    L     R8,0(R9)           GET NUMBER OF TIMES A NUMBER OCCURED
         XDECO R8,0(R11)          PLACE VALUE OF COUNTER IN OUTPUT
         LA    R9,4(R9)           GET ADDRESS OF NEXT COUNTER
         LA    R11,12(R11)        INCREASE OUTPUT POINTER
         BCT   R10,LOOP1          RETURN FOR NEXT COUNTER
         XPRNT OUTTTT
*
*        .         .         .         .         .         .         .
*        WHEN AN STIMER MACRO IS ISSUED WITH TASK IT ONLY DECREMENTS
*        THE TIME INTERVAL WHEN THE TASK IS ACTIVE.  THE TUINTVL
*        IS A FULLWORD ON A FULLWORD BOUNDRY THAT GIVES THE TIME IN
*        TIMER UNITS.  ONE TIMER UNIT = APPROXIMATELY 26 MICRO SECONDS
*        THIS STIMER SETS THE INTERVAL TO APPROXIMATELY 5 SECONDS.
*        THEN 1000 BCT ON R10 TO * ARE DONE.
*        .         .         .         .         .         .         .
*
         LA    R10,1000           SET R10 TO 1000 FOR BCT
         STIMER TASK,TUINTVL=TUNUM
         BCT   R10,*
```

```
*
*        .         .         .         .         .         .
*        WHEN A TTIMER MACRO IS ISSUED IT RETURNS THE TIMER REMAINING
*        IN THE TIMER INTERVAL ISSUED BY THE STIMER MACRO.  THEN
*        COMPUTE THE TIME FOR 1000 BCT INSTRUCTIONS AND OUTPUT THE
*        RESULTS.  NEXT CANCEL THE TIMER INTERVAL USSING THE TTIMER
*        WITH CANCEL SPECIFIED.
*        .         .         .         .         .         .
*
         TTIMER
         L     R9,TUNUM           GET NUMBER OF TIMER UNITS AT START
         SR    R9,R0              GET NUMBER OF REMAINING TIMER UNITS
         SR    R8,R8              ZERO REG 8
         LA    R6,26              SET R6 TO NUMBER OF MICRO SECONDS
*                                 PER TIMER UNIT
         MR    R8,R6              GET THE NUMBER OF MICRO SECONDS
         SR    R8,R8              ZERO R8 FOR DIVIDE
         LA    R6,1000            SET R6 TO 1000 FOR DIVIDE
         DR    R8,R6              GET AVERAGE TIME PER CALL IN MICRO S
*                                 SECONDS
         XDECO R9,OUZZ            PUT R9IN OUTPUT
         XPRNT OUZ,84
         TTIMER CANCEL
*
*        .         .         .         .         .         .
*        NEXT ISSUE AN STIMER WITH AN EXIT ROUTINE.  WHEN AN EXIT
*        ROUTINE IS GIVEN AT THE END OF THE INTERNAL THE ROUTINE IS
*        GIVEN CONTROL.  THEN SET CHECK TO 0 TO STOP THE LOOP.
*        THE TASK INDICATES THE TIME TO BE DECREMENTED ONLY WHEN THE
*        TASK IS ACTIVE.  AGAIN THE INTERVALIS SPECIFIED IN TIMER
*        UNITS.  IT IS 1.04 SECONDS APPROXIMATELY.
*        AFTER THE LOOP IS STOPED THEN CANCEL THE TIME INTERVAL
*        WITH A TTIMER CANCEL.
*        .         .         .         .         .         .
*
         LA    R9,0               SET R9 TO 0 FOR COMPARE
         LA    R10,0              SET R10 TO 0
         BCTR  R10,0              SUBTRACT ONE FROM R0
         STIMER TASK,LPINTVLP,TUINTVL=LPINTVL
LP       C     R9,CHECK           CHECK TO SEE IF INTERVAL IS OVER
         BE    MDONE              IF TIME INTERVAL OVER GO TO DONE
         BCT   R10,LP             RETURN DO LP OVER
MDONE    TTIMER CANCEL
         XRETURN SA=*
         DROP  12
*
*        .         .         .         .         .         .
*        THIS IS THE STIMER EXIT ROUTINE IT ZEROS CHECK AND PRINTS
*        A MESSAGE AND RETURNS CONTROL TO A PROGRAM THAT THEN RETURNS
*        CONTROL TO THE PLACE WHERE IT LEFT OFF WHEN THE INTERVAL
*        EXPIRED.
*        NOTE THE USE OF SAVE AREAS.
*        .         .         .         .         .         .
*
LPINTVLP XSAVE SA=NO
         XC CHECK(4),CHECK
         XPRNT LPINTVLM,80
LPINTVL1 XRETURN SA=NO
LPINTVLM DC    CL80'0LP TAKES LONGER THAN 1.04 SECONDS SO STOP LOOP'
CHECK    DC F'1'
LPINTVL  DC    F'4000'
```

```
OUZ      DC    C'0THE AVERAGE TIME IN MICRO SECONDS FOR A BCT INSTRUC'
         DC    C'TION IS'
OUZZ     DC    CL12' ',C'MICRO SECONDS'
TUNUM    DC    F'20000'
MONE     DC    F'1'
DONE     DC    F'0'
TWO      DC    F'0'
THREE    DC    F'0'
FOUR     DC    F'0'
FIVE     DC    F'0'
SIX      DC    F'0'
SEVEN    DC    F'0'
EIGHT    DC    F'0'
NINE     DC    F'0'
TENT     DC    F'0'
TEN      DC    F'10'
OUT      DC    D'0'
WORD     DC    2F'0'
OUTPUT5  DC    C'0TIME: '
OUTPUT2  DC    C'  :'
OUTPUT3  DC    C'  :'
OUTPUT4  DC    C'    '
         DC    C'DAY'
OUTPUT1  DC    C'    OF 19'
OUTPUT   DC    C'  '
OUTT     DC    X'00'
         DC    CL31' '
         DC    C'THE NUMBER OF TIMES THAT EACH NUMBER OCCURED IS GIVEN'
         DC    C' BELOW:',CL62' '
OUTTT    DC    C'0',CL11' ',C'1',CL11' ',C'2',CL11' ',C'3',CL11' '
         DC    C'4',CL11' ',C'5',CL11' ',C'6',CL11' ',C'7',CL11' '
         DC    C'8',CL11' ',C'9',CL11' ',C'10',CL12' '
OUTTTT   DC    CL133' '
         LTORG
         EJECT
IAND     CSECT
         XSAVE SA=NO,TR=NO
*
*--THIS IS A RANDOM NO GENERATOR
*--IT HAS TWO ARGUMENTS THE FIRST THE MAX VLLUE TO BE RETURNED
*--THE SECOND THE MIN VALUE TO BE RETURNED
*--TI FIRST DETERMINES  THE NO OF BITS IN THE MAX VALUE
*--THEN IT GENERAES THAT MANY RANDOM BITS
*--THESE BITS ARE PLACED IN ONE OF THE REGISTERS
*--THEN IT DETERMINES IF THE NO GENERATED IS WITHIN THE BOUNDS OF THE
*--THE TWO AGUMENTS IT WAS GIVEN IF IT IS NOT THEN IT DOES THE PROCESS
*--ALL OVER AGIAN
*
*
*--FIRST FIND THE MAX AND MIN VLUES TO BE CONSIDERED
         L     R2,0(R1)           LOAD THE ADDRESS OF THE MAXIMUM OF
*                                  THE RANDOM NO. GENERATOR
         L     R8,4(R1)        LOAD THE ADDRESS OF 2ND ARG
         L     R8,0(R8)        LOAD THE MIN VALUE
         L     R2,0(R2)           LOAD THE MAX OF RANDOM NO GENERATOR
*
*--FIND THE NO OF RANDOM BITS TO BE SET
OVERFL   LA    R4,0               PUT A 0 IN R4
         LR    R3,R2              MAKE A COPY OF MAX
OVER     SRL   R3,1               FIND THE NO OF BITS TO BE SET
         LA    R4,1(R4)           R4 IS THE NO OF BITS TO BE SET
```

```
         LTR   R3,R3               DETERMINE IF R3 IS 0
         BNE   OVER                IF NOT 0 THEN SHIFT AGAIN
*
*--PLACE A 1 IN THE ZERO BIT OF R5
         LA    R5,2048         PUT A 1 IN THE O BIT OF R5
         SLL   R5,20               MOVE THE BIT TO BIT 0
         SR    R6,R6               ZERO REG 6
*
*--THE NEXT SECTION GENERATES A RANDOM BIT
AGAIN    L     R0,SAVEODD      GET THE ODD NO.
         LTR   R0,R0           DETERMINE IF SAVEODD=0
         BNE   BEGIN           IF SAVEODD NO 0 THEN BEGIN COMPUTING
         PRINT GEN
         TIME  BIN
         PRINT NOGEN
LOADODD  ALR   R0,R0           MAKE R0 JUST BELOW OVERFLOW
         BNO   LOADODD         IF NOT OVERFLOW MAKE R0 OVERFLOW
         BCTR  R0,0            SUBTRACT 1 FROM R0
BEGIN    LR    R1,R0           LOAD R1 FROM R0
         ALR   R0,R0           CREATE A RANDOM BIT
         BO    ONETWO          BRANCH IF A BIT IS CREATED
         ALR   R0,R1           CREATE A RONDOM BIT
         BO    ONE             IF BIT IS CREATED BRANCH
         LA    R1,0            PUT A 0 IN R1 FOR THE RNADOM BIT WHICH
*                               WAS NOT CREATED
         B     FINISH          GO TO THE END OF THE SECTION
ONETWO   ALR   R0,R1           CREATE A ROANDOM BIT
         BO    BEGIN           RETURN AND START AGAIN
ONE      LR    R1,R5               LOAD THE BIT FROM THE RANDOM CHOICE
FINISH   ST    R0,SAVEODD      SAVE THE ODD NO.
*--SHIFT THE BIT INTO R6 WHICH WILL CONTAIN THE RANDOM NO.
         LR    R7,R1           MOVE THE RANDOM BIT TO R7
         SLDL  R6,1            MOVE THE RANDOM BIT INTO R6
*
*--IF THIS IS NOT THE LAST BIT  GENERATE ANOTHER
         BCT   R4,AGAIN        FIND THE NEXT RANDOM BIT
*
*--DETERMINE IF THE NO EXCEEDES THE MAX VALUE
         CR    R6,R2           DETEMMINE IF THE NO GENERATED EXCEEDES
*                               THE MAX VALUE
         BP    OVERFL          IF MAX EXCEEDED RETURN AND DO AGIAN
*
*--DETERMINE IF THE NO EXCEEDES THE MIN VALUE
         CR    R6,R8           DETERMINE IF THE RANDOM NO IS L5SS TH1N
*                               THE MIN VALUE
         BM    OVERFL          IF OUT OF RANGE DO OVER
         LR    R0,R6           PLACE RESULT IN R0 FOR RETURN
         XRETURN SA=NO,RGS=(14-15,1-12),TR=NO
L        DS    0D
SAVEODD  DC    F'0'
         END
/*
//DATA.SYSUDUMP DD SYSOUT=A
```

```
//*
//*      THIS JOB WILL RUN WITH   TIME= 25 SECONDS
//*                               RECORDS = 600
//*
// EXEC ASGCG
//SOURCE.INPUT DD *
/*LOG
*
*        .         .         .         .         .         .         .
*        THE PURPOSE OF THIS PROGRAM IS TO DEMONSTRATE THE FOLLOWING
*        MACROS:
*        TIME
*        WTL
*        WTO
*        WTOR
*        WTO ROUTCDE = 11
*        .         .         .         .         .         .         .
*
         PRINT NOGEN
MAIN     CSECT
         XSAVE
         PRINT GEN
*
*        .         .         .         .         .         .         .
*        WITH THE WTL MACRO THE MESSAGE APPEARS AT THE BEGINNING OF
*        THE PROGRAM IN THE LOG
*        WITH THE WTO ROUTCDE = 11 WE ARE WRITING TO THE PROGRAMMER
*        BUT ON THIS SYSTEM THIS IS ALSO THE SYSTEM LOG
*        .         .         .         .         .         .         .
*
         WTL   'THIS IS AN EXAMPLE OF WTL,WTO,WTOR, AND WTO RTCDE=11'
         WTO   'THIS IS AN EXAMPLE OF WTO, ROUTCDE =11',ROUTCDE=11
*
*        .         .         .         .         .         .         .
*        IF ACTUALLY CODE THESE WOULD BE EXAMPLES OF WTO AND WTOR
*
*        WTO
*        THE FIRST PARAMETER IS THE MESSAGE TO BE WRITTEN TO THE
*        OPERATOR
*        THE ROUTCDE GIVES THE CONSOLE NUMBER TO BE WRITTEN TO.
*        THE MEANING OF THE DESCRIPTOR IS GIVEN IN APPENDIX THREE TO
*        SUPERVISOR AND DATA MANAGEMENT MACROS
*
*        WTO   'IF JOB DOES NOT TERMINATE IN 10 SECONDS TERMINATE',
*              ROUTCDE=(1,2),DESC=1
*
*        WTOR
*        THE FIRST PARAMETER IS THE MESSAGE TO BE WRITTEN TO THE
*        OPERATOR WITH THE INDICATED REPLY INDICATED
*        THE SECOND PARAMETER IS THE REPLYADDRESS THAT IS WHERE IN
*        THE PROBLEM PROGRAM THE ANSWER GOES
*        THE THIRD PARAMETER IS THE MAXIMUM LENGTH OF THE REPLY
*        THE FOURTH PARAMETER IS THE NAME OF AN EVENT CONTROL BLOCK TO
*        BE POSTED BY THE CONTROL PROGRAM IN THE WTOR MACRO
*        THE ROUTCDE IS THE SAME AS FOR THE WTO MACRO
*        THIS DESC INPLIES THAT SOME IMMEDIETE ACTION IS REQUIRED ON
*        THE PART OF THE OPERATOR.
*
*        WTOR  'IF STANDARD OPERATING CONDITIONS? REPLY YES OR NO',
*              REPLYADD,3,MECB,ROUTCDE=2,DESC=2
*
```

```
*         .         .         .         .         .         .
*
          PRINT NOGEN
          XRETURN SA=*
MEXB      DC    F'0'
REPLYAD   DC C'   '
          END
/*
```