NEUCC NORTHERN EUROPE UNIVERSITY COMPUTING CENTER
TECHNICAL UNIVERSITY OF DENMARK, 2800 LYNGBY, DENMARK, BLDG. 305, TEL. (01) 88 12 77
CABLE: NEUCCDTH - COPENHAGEN

Jan Andersen
Stationsvej 3
Kokkedal
☎ 86 07 18

Introduction to

Assembly Language Programming

With OS/360

A supplemental guide to assembler programmers
new to OS/360

C/114/70
Rennie Petersen
NEUCC
July 1970

CONTENTS

Introduction

This set of notes has been put together as an aid to persons who wish to program the IBM system /360 under OS, and who do not have much or any experience in OS assembler programming. Note however, that these notes are not intended to be a complete course in OS assembler programming, but rather just a supplement to the regular IBM manuals and courses on OS and assembler programming. Furthermore, these notes assume that the user has some previous assembler programming experience, and has some knowledge of the /360 and OS.

The following manuals may be used in conjunction with these notes.

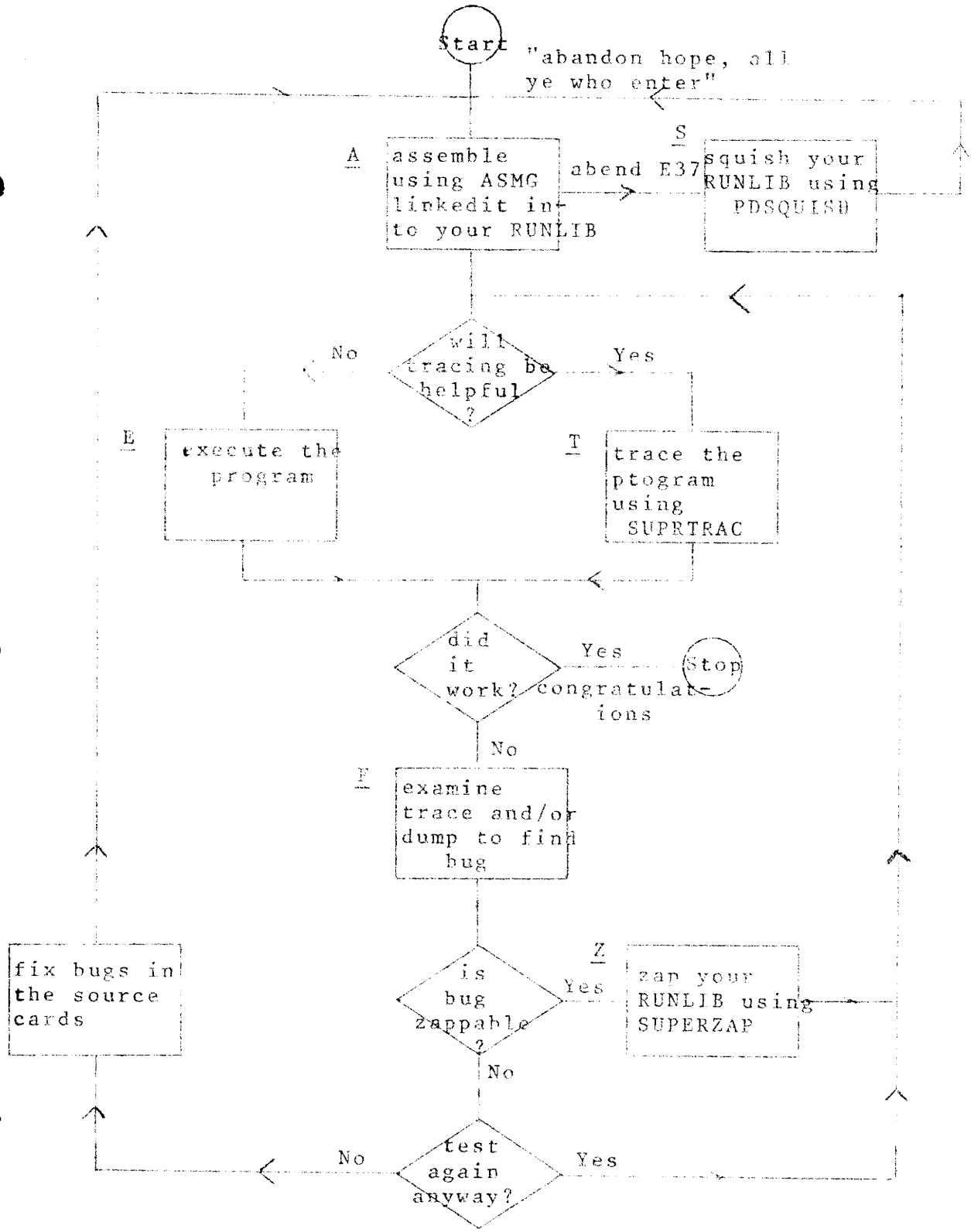
/360 Principles of Operation
OS Concepts and Facilities
OS Job Control Language
OS Assembler Language
OS Assembler (F) Programmer's Guide
OS Linkage Editer and Loader
OS Supervisor and Data Management Services
OS Supervisor and Data Management Instructions
OS Messages and Codes
OS Programmer's Guide to Debugging

NEUCC 360 User's Manual

Part I. How to do assemblies under OS

This section is meant to describe the overall activity of the assembly language programmer in assembling and testing a program under OS/360. It is simply a guide, based on experience, of the most productive methods of using the system to get your work done.

The overall process may be flowcharted as follows:



Summary of overall process.

A Assembly-Linkedit.

There are several advantages to having a RUNLIB dataset and doing an immediate linkedit into it following each assembly. These include:

1) you can test your program (with or without TRACE) as many times as you wish without re-assembling the source deck each time

2) if during testing you find a bug that is zappable, then you can SUPERZAP your RUNLIB and test the program again without re-assembling

3) if you have a large program, and it is written modularly, then you need only re-assemble the modules in error, and then linkedit them against your RUNLIB, keeping the correct modules.

S Squishing RUNLIB

This is the one disadvantage of having a permanent RUNLIB dataset: it overflows every so often. However, PDSQUISH will squish it for you in a few seconds.

E Executing the program

This is the fun part (if it works). If it doesn't work, OS is very good at printing core dumps.

T Tracing the program

By using the SUPRTRAC program you can trace your program without making any special modifications to it. You can also specify what parts are to be traced and what parts not traced in order to reduce the amount of time and volume of paper used.

F Finding the bug

This is the un-fun part (I hate core dumps)!

Z Zapping your program

By using SUPERZAP it may be possible to fix (or bypass to permit further testing) certain kinds of bugs. You cannot add instructions with SUPERZAP, but you can change instructions or data, and you can effectively remove instructions by changing them to NOP's.

U Updating your source

If your source is on cards, you use classical updating techniques. However, the WITS system will hopefully be running on the /75 soon, permitting you to keep your source decks on disk and do on-line updating, which is much faster, more convenient, and less error prone (I love WITS).

Control Cards to Create
your RUNLIB Dataset

```
//jobname   JOB    acctinfo,'name',MSGLEVEL=1
//          EXEC   PGM=IEHPROGM
//SYSPRINT  DD     SYSOUT=A
//DD        DD     DSN=your.runlib.dataset,VOL=SER=volume,
//                 UNIT=SYSDA,SPACE=(TRK,(20,2,10)),DISP=(NEW,KEEP)
//SYSIN     DD     *
          CATLG  DSNAME=your.runlib.dataset,VOL=2314=volume
/*
```

Notes - this job both creates and catalogs your RUNLIB dataset

- the user should check the current computing center policy on user datasets before creating his dataset.

Control Cards for
Assembly-Linkedit

```
//jobname          JOB      acctinfo,'name',MSGLEVEL=1
//                 EXEC     ASMGCL,[PARM.ASM='RENT',]  -include if testing for re-entroncy errors desired
//                 PARM.LKED='LIST,LET,NCAL',
//                 COND.LKED=(16,LT,ASM)   -to permit linkediting even if assembly errors occur
//ASM.SYSIN        DD       *
            source deck
/*
//LKED.SYSLMOD     DD       DSN=your.runlib.dataset,DISP=OLD
//LKED.RUNLIB      DD       DSN=your.runlib.dataset,DISP=SHR
//LKED.SYSIN       DD       *
                   ENTRY    entryname
                  [INCLUDE  RUNLIB(progname)]-include if re-
                   NAME     progname (R)      placing csects
                                              in your module
/*
```

Notes
- this job does both the assembly and the linkedit
- ASMG is operationally similar to Assembler F, but up to four times faster
- default options for ASMG are LOAD,NODECK, instead of DECK,NOLOAD
- ASMG uses SYSLIN instead of SYSGO for the object dataset
- if PARM.ASM='BATCH[,RENT]' is specified, then multiple source decks may appear in ASM.SYSIN.

Control Cards for
Squishing RUNLIB

```
//jobname   JOB    acctinfo,'name',MSGLEVEL=1
//          EXEC   PDSQUISH
//GO.DD     DD     VOL=SER=volume,UNIT=SYSDA,DISP=OLD
//GO.SYSIN  DD     *
        CONDENSE  DSNAME=your.runlib.dataset,VOL=2314=volume
/*
```

Notes - thats all there is to it

- the volume specified on the GO.DD card is the same as in the CONDENSE statement.

Control Cards for
Executing your Program

```
//jobname          JOB      acctinfo,'name',MSGLEVEL=1
//GO               EXEC     PGM=progname [,PARM= only if your
                                          program uses parms]
//GO.STEPLIB       DD       DSN=your.runlib.dataset,DISP=SHR
//GO.SYSUDUMP      DD       SYSOUT=A

if your program uses SYSPRINT include the following
//GO.SYSPRINT      DD       SYSOUT=A

if your program uses any other DD cards, include them here
//GO.ddname        DD       specifications

if your program uses SYSIN include the following
//GO.SYSIN         DD       *
                 data cards
/*
```

Notes - use SYSUDUMP instead of SYSABEND - save lots of paper and time.

Control Cards for
Tracing your Program

```
//jobname       JOB     acctinfo,'name',MSGLEVEL=1
//              EXEC    SUPRTRAC
//GO.STEPLIB    DD      DSN=your.runlib.dataset,DISP=SHR
```

optionally include SYSUDUMP, SYSPRINT, SYSIN, or any other dd cards used by your program

```
//GO.TRACEIN      DD      *
                PROGRAM     progname
              ( REGION      csectname+displacement, (PRINTON ))
              (             csectname+displacement, (PRINTOFF))
              ( REGION      csectname+displacement, (PRINTON ))
                            csectname+displacement, (PRINTOFF))
                EXEC        PGM=progname (,PARM='~' only if)
                                         (your program uses )
                                         (parms             )
```

Notes
- if no REGION statements are present, all tracing is printed
- to selectively print only certain portions of the program use REGION statements
- the REGION statements are processed sequentally until one is found that includes the instruction being traced. If none is found, then PRINTOFF is assumed
- the statement `REGION START, START+X'24E', PRINTON` causes printing of all instructions from CSECT START until CSECT START + displacement X'24E'
- the special symbols $MODSTRT, $MODEND, and $SECTEND may be used as operands of REGION

e.g.
```
PROGRAM   START
REGION    START+X'10C',START+X'24F',PRINTOFF
REGION    $MODSTRT,$MODEND,PRINTON
```

this prints tracing for the entire program except the instructions between START+X'10C' to START+X'24E'.

$MODSTRT = Modul start

$MODEND = Modul end

$SECTEND = CSECT ophören.

Control Cards for SUPERZAPing your program

```
//jobname   JOB      acctinfo,'name',MSGLEVEL=1
//          EXEC     SUPERZAP
//GO.SYSLIB DD       DSN=your.runlib.dataset,DISP=OLD
//GO.SYSIN  DD       *
            NAME     progname csectname
            VERIFY   displacement data
            REP      displacement data
          ( VERIFY   displacement data )  repeat as many
          ( REP      displacement data )  times as necessary
/*
```

Notes
- the displacement is a 4-digit hex number giving the displacement from the start of the CSECT (not necessarily the program)
- the data may be any even number of hex digits
- VERIFY checks old contents of module, REP replaces new data

e.g.

NAME	START	START
VERIFY	010C	5010604C
REP	010C	47000000

in this example a ST instruction at location 10C is replaced by a NOP.

The program name and the csect name were both START.

Part II. Assembly Language Programming Techniques

This section is meant to provide some general pointers on the best overall method of organizing programs written in assembler for OS/360. Again, the methods described here do not constitute rules, but simply explain the techniques which I have found, through hard experience, to be the best way to implement assembler programs, particularly large ones.

Three items are discussed in this section:

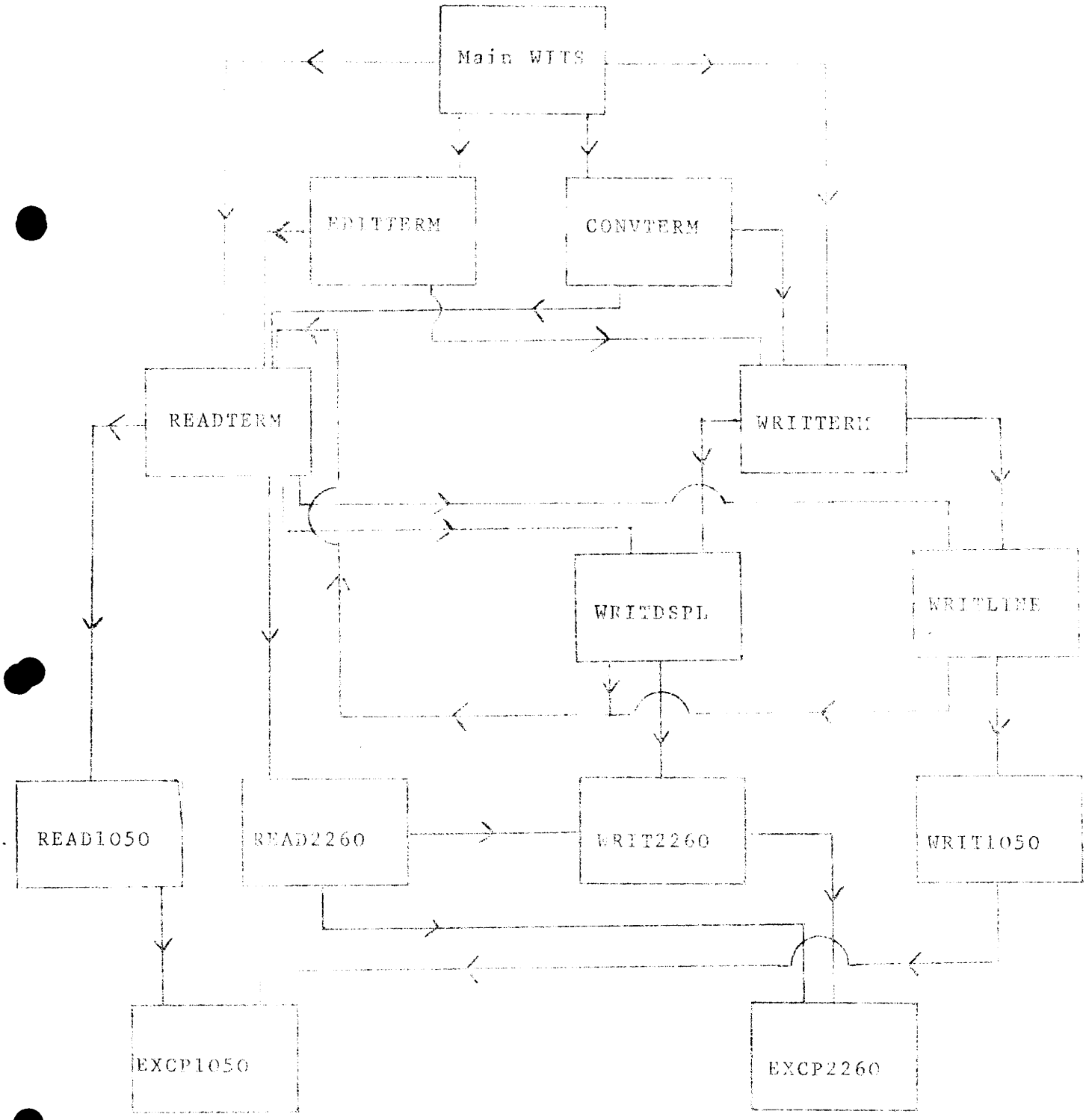
A) Programming modularly - that is, dividing the total program up into subroutines, each of which has a definite purpose or job to do and having a "clean" interface with the rest of the program.

B) Programming re-enterably - this is a concept relatively new to the third generation of computers. It is a very powerful and useful technique when applied to programs of general use, since it allows many users to use the same program simultaneously.

C) Macro augmentation - through the use of macros a programmer can reduce the amount of repetitive coding he does, thus increasing his productivity and reducing the possibility of errors.

A) Programming Modularly

By programming modularly, I mean the division of a large program up into a small main routine and multiple subroutines. As an example, consider the following structure of the WITS terminal support package:



This is perhaps more complicated than a typical case, but it does show that modular programming can still leave room for great flecibility in logical structure. The following pages will describe some of the requirements and advantages inherent in this method of programming.

A.1 Logical Independence of modules

Each module is written as an independent subroutine.

It saves and restores the callers registers, has a well-defined set of input parameters, a certain specific job to do, and a set of return or error codes that it passes back to the caller to describe whether it found any unusual or error conditions.

This approach has several advatages. It simplifies the planning and implementing of the total program because, once the division of modules has been decided upon, you can concentrate on each individual module, one at a time, without worrying too much about the total program or any strange interactions between diverse parts of the job. It eases the debugging problem since, to some extent, modules can be written and tested one at a time, and when there is a bug it is usually easy to isolate the particular module which is incorrect.

Finally, this technique permits a structuring of the modules which can provide increasing logical independence as you move up the structure. For example, the bottom routines in the WITS terminal support package are completely dependent on the type of terminal (1050 or 2260), while at the top, the main WITS routines do not need to know (and in fact, are unable to find out) what kind of terminal is being used when they call READTERM, WRITTERM, etc. This provides compatibilty between the various terminals, and greatly simplifies the coding of the main WITS routines.

A.2 Physical Independence of Modules

Each module is a seperate CSECT, and thus can be physically independent of the other modules in the program. The following are some of the advantages of this method.

The base register problem becomes almost non-existent since each module will load its own base register and it is extremely unlikely that any one module will be greater than 4 K bytes. This is a problem new with the third generation, but one which needs not worry you if you follow this coding method.

This technique permits you to assemble only those modules that you have changed. Since each module is a CSECT, the linkediter can be used to replace the re-assembled CSECTs in the load module, while keeping those CSECTs which are not altered. This results in faster assemblies when it is not necessary to re-assemble the entire program.

If this technique is used it becomes very easy to use the overlay facilities of the linkediter since it works on a CSECT basis. Also, it would simplify the process of going to some kind of system where the modules are normally non-resident but are loaded when required (an experimental terminal system called VPATS does this for instance).

Finally, it often happens that when writing a new program you find that you could use a subroutine that you once wrote for another program. Under this system it is very easy to "lift" a subroutine out of a program for insertion into another.

B) Programming Re-enterably

When a program is coded re-enterably, a single copy of it can be used simultaneously by many users. This is very important for any program of general applicability, since otherwise a seperate copy of the program must be in core for each user, a great waste of core in same cases. At Waterloo we discovered, too late, that several programs, including WATFIV and PDSQUISH, should have been re-enterable. In my opinion, any program of a general usage nature and which has even the remote possibility of being executed simultaneously by multiple users should be written re-enterably.

Writing a program so that it is re-enterable is not too difficult, especially, if the routine control macros described later are used. The basic rule for re-enterable programs is that they must not modify themselves, i.e. the program CSECT must never be altered during execution of the program. This is enforced in OS by store-protecting the program from itself, insuring that there is no possibility of it being changed by any method. The following paragraphs outline the main things to remember in re-enterable programming.

B.1 GETMAINing of Work Areas

Since the CSECT of a re-enterable program may contain only instructions and constants, it is necessary to do GETMAINs for any save areas, work areas, buffers, etc. which are stored into, modified, manipulated, read into, or otherwise changed at any time. The $ROUTINE/$ENTRY macros described later provide an easy method of getting save areas and work areas whose lengths do not vary from run to run. Other work areas or buffers whose lengths are not known until execution time must be dynamically allocated by using the R form of the GETMAIN system macro.

For example:

```
         LH      R0,UT1DBC+DBCBLKSI   GET BUF LEN
         GETMAIN R,LV=(0)             GET THE CORE
         LR      R7,R1                SAVE ADDR
```

In this example a buffer has been GETMAINed, and now this buffer can be used for READs or WRITEs on that data set. Note that this would not be necessary for QSAM since it gets the buffers for you.

B.2 EXecuting SS instructions

Often in a program it is necessary to do an MVC, TR, TRT, or CLC, etc. on character strings whose lengths are not known until execution time. One way to do this is to modify the length field of the instruction at execution time using the STC instruction. For re-enterable programs this is 'verboten' (it's not a good idea anyway since on models 65 and up a STC *+5 instruction gives the instruction pre-fetch circuits a nervous breakdown). The correct way of handling this situation is with an EX instruction which operates on the out-of-line subject SS instruction. Remember that the instruction length operand is the real length -1, and also that an EX cannot be done on R0.

For example:

```
              LR      R4,R2         STRING LENGTH IN R2
              BCTR    R4,0          (R4)-1 FOR EX
              EX      R4,VARMVC     MOVE THE STRING
VARMVC        MVC     0(*-*,R7),    BUF OUT-OF-LINE EX'ED MVC
```

In this example a string whose length is available in R2 is moved to the location pointed at by R7 by EXecuting an MVC instruction.

B.3 Re-enterably coding system macros

The rule that the program must not store into itself or modify itself includes the code generated by system macros. Unfortunately, many of the IBM system macros do not lend themselves to re-enterable programming. Those macros that expand into (generate) control blocks (DCB for example) must be expanded in the constant section of the program, a GETMAIN done for their core (see $ROUTINE/$ENTRY for an easier way), and the new core initialized by moving the constant DCB into the core required for the DCB. This new DCB in the GETMAINed core can now be OPENed and used without violating any re-enterability rules.

The macros which only expand into instructions can be used freely. Examples of these macros are EXCP, WAIT, GET, PUT, and the R form of GETMAIN and FREEMAIN. However, the majority of the system macros expand into a combination of instructions and in-line parameter lists or control blocks. With these macros it is necessary to do the following in order to produce re-enterable code:

a) Expand an MF=L form of the macro in the constant area of the program. Only those macro parameters which are constant should be specified.

b) Get core for on image of the MF=L macro (see $ROUTINE/ $ENTRY for an easy way) and initialize it by moving the constant MF=L macro into the aquired core.

c) Execute an MF=(E,) form of the macro which points at the aquired copy of the MF=L form of the macro. All the macro parameters which are variable should be specified on the MF=(E,) macro.

For example:

(assume the core has been gotten for the MF=L open and is pointed at by R5, and that the DCB is pointed at by R6)

```
        MVC     0(OPLEN,R5),OPMFL   INIT. THE MF=L OPEN
        OPEN    ((R6)),MF=(E,(R5))  DO THE OPEN


OPMFL   OPEN    (*-*,INPUT),MF=L CONSTANT MF=L OPEN
OPLEN   EQU     *-OPMFL
```

In this example the MF=L part of the open includes the option INPUT since this is constant, while the MF=(E,) part specifies the DCB address since this is different for each run.

C) Macro Augmentation

It is often the case that you find yourself coding the same sequence of instructions over and over again, with little or no variation from one occurance to the next. In this situation it is often desirable to write a macro which will generate the sequence of instructions. This has several advantages. Once the macro is written and debugged, the probability of making a mistake in coding the instruction sequence decreases. Furthermore, since the macro call is noramlly shorter and simpler than the instructions it generates, a reduction in coding time and key-punching time results. Finally, a well-defined macro is easy to read, thus making the logic of the program easier to see, especially when the listing is produced with PRINT NOGEN.

Once a macro is written and debugged it is usually placed in a macro library dataset which is concatenated into SYSLIB during the assembly step. Until the macro is completely debugged it should be placed at the start of your program. This is due to the fact that the assembler diagnostics concerning macros from SYSLIB are impossible to relate back to the specific statement which was in error.

The following section describes a set of macros of general applicability which are available for use.

Part III. A set of macros to simplify assembler coding

This section describes a set of macros which are generally useful to the average assembly language programmer. These macros are in the data set SYS2.MACLIB and are available by default if you use any of the ASMG catalogued procedures (ASMGC,ASMGCL, etc.).

In order to prevent any possible conflict with user macros or IBM system macros, and also to make it obvious that these instructions are macro instructions, the mnemonic upcode of each of the macros begins with a $. Here at NEUCC, this character may appear as Â on some of the keypunches, terminals, and printers.

These macros can be divided into 5 categories as follows:

A)	Register Macros	$SETR, $SETX, $USE, $DROP
B)	String Macros	$DC, $LDV
C)	Routine Macros	$ROUTINE, $ENTRY, $CALL, $RETURN
D)	Switch Macros	$SWITCH, $BIT, $SET, $ON, $OFF, $IF, $IFON, $IFOFF
E)	Miscellaneous Macros	$ALIGN, $CCW, $REF, $NULL

A. Register Macros

Macros used to set up symbolic names for the registers, and to improve facilities for USING and DROPing base registers.

These macros are used for three purposes:

A.1 Symbolic register names

When programming the /360, it is common to use symbolic names for the registers (e.g. R5 instead of just 5). There are several reasons for this. The first is that it is just more natural to say R5 than simply 5, partly because R5 is obviously and uniquely associated with register 5, while the decimal number 5 has many possible associations. Another reason for using symbolic registers is that each reference to the register appears in the XREF, allowing you to locate all the instructions which use each register, or to help you to find a free register (i.e. one which is not used by the program). Finally, if it becomes necessary for some reason to change the logic of a program so that, for example, it uses R3 instead of R5, this can be done quite easily by replacing all occurances of 'R5' with 'R3', easily done with the WITS REPlace command. In contrast, replacing all occurances of '5' by '3' might after some decimal constants which happened to include the digit 5.

There are two common systems of symbolic register names in use. One system names the registers in decimal, R0 to R15, while the other system names them in hex, R0 to RF. My experience has been that the second system is far superior to the first, and therefore the hex system is the one implemented in these macros.

A.2 Symbolic Base Register references

In /360 programming it is often the case that one of the general registers contains the address of some data which is being referenced or manipulated. The following sequence of instructions in an example of this.

```
LOOP    L       R1,4(,R1)       POINT AT THE NEXT BLOCK
        TM      3(R1),X'80'     TEST END OF CHAIN BIT
        BZ      LOOP            LOOP IF NOT
        LA      R1,16(,R1)      POINT AT THE DATA PART
STORE   ST      R5,4(R6,R1)     DO AN INDEXED STORE
```

In this example, the general register R1 is being used as a data base register to move along a chain of control blocks. Note that the sequence '(R1)' and '(,R1)' appears quite often, and that this is not a particularly nice or obvious notation for the use of a changing base register. Using the method of symbolic Base Register references, the above code would be replaced by

```
LOOP    L       R1,XR1+4        POINT AT THE NEXT BLOCK
        TM      XR1+3,X'80'     TEST END OF CHAIN BIT
        BZ      LOOP            LOOP IF NOT
        LA      R1,XR1+16       POINT AT THE DATA PART
STORE   ST      R5,XR1+4(R6)    DO AN INDEXED STORE
```

This, in my opinion at least, is a far more natural notation for using a changing base register. Note that the sequence of characters '~~RXn~~ XRn' simply means 'base register n'.

The implementation of this scheme is not too difficulty and, in fact, needs not concern you at all since it is completely handled by the macros described in this section. Basically, what is done is to define 15 DSECTs whose names are XR1 to XRF, and to do USINGs on them such that the assembler thinks that register R1 contains the address of DSECT XR1, and similarly for the other 14 DSECTs. Thus, whenever the assembler finds a reference to XR1, it searches its USING table and determines that base register R1 plus 0 displacement provides the proper addressability for the symbol XR1. As mentioned, this needs not concern you since the macros which follow handle the declarations of the DECTs and the USINGs automatically.

A.3 Improved USING and DROPing facilities

Two of the macros which follow are $USE and $DROP. These macros are used by you instead of the assembler USING and DROP instructions. They have the following advantages:

The $USE macro reverses the order of the operands from USING. In my opinion, it is more natural to specify the register first and the address second when declaring a constant base register.

The $DROP macro has the extended facility for dropping all the registers currently in use. This is invoked by specifying $DROP without any operands.

Finally, $DROP implements the symbolic base register references described above. This is done by doing a USING XRn,Rn instead of a DROP Rn for each register that is $DROPed.

The following pages contain detailed descriptions of these four macros.

$DCL R
~~$SETR~~

$DCL
~~$SETR~~

This macro is used to declare the symbolic register names. Its format is simply

~~$SETR~~ $DCL R

This macro has no operands. It simply generates sixteen EQU statements to assign values to the names R0 to RF.

Example

~~$SETR~~ $DCR R

this would generate the following instructions

```
R0      EQU     0
R1      EQU     1
R2      EQU     2
R3      EQU     3
R4      EQU     4
R5      EQU     5
R6      EQU     6
R7      EQU     7
R8      EQU     8
R9      EQU     9
RA      EQU     10
RB      EQU     11
RC      EQU     12
RD      EQU     13
RE      EQU     14
RF      EQU     15
```

Sammesat

$DCL (R,XR)

$DCL
~~$SETX~~

$SETX

This macro is used to declare the symbolic base register reference names. Its format is simply

~~$SETX~~ $DCL XR

This macro has no operands. It simply generates fifteen DSECT statements to declare the names XR1 to XRF.

Example

~~$SETX~~ $DCL XR

this would generate the following instructions

```
XR1   DSECT
XR2   DSECT
XR3   DSECT
XR4   DSECT
XR5   DSECT
XR6   DSECT
XR7   DSECT
XR8   DSECT
XR9   DSECT
XRA   DSECT
XRB   DSECT
XRC   DSECT
XRD   DSECT
XRE   DSECT
XRF   DSECT
```

elh

$DCL (R,XR)

$USE

This macro is used to declare a constant base register. Its format is simply

```
$USE    Rn,address
```

This macro simply generates an assembler USING statement which permits the assembler to assign addressability (base + displacement) for any locations within 4K bytes of the specified address. It also sets a bit in on internal macro table which permits $DROP to know whether the register is in use or not. Note that the order of operands for $USE is the opposite of the assembler USING statement, and that only one register may be declared on each $USE instruction.

Example

```
$USE    R5,UT1DCB
```

this statement would generate the following instruction

```
USING   UT1DCB,R5
```

In this example R5 has been declared to contain the address of UT1DCB. The assembler will now be able to assign addressability to any reference within 4K of UT1DCB by generating a displacement + a base of 5.

$DROP

This macro is used to counter-act a former $USE statement, i.e. to specify that a register no longer is to be used as a constant base register. The format of the $DROP macro is:

```
$DROP   [Rn[,Rm,...]]
```

If no operand is specified, then all the registers currently in use are dropped. If an operand is present, then only those registers specified are dropped. An error message is issued if an attempt is made to explicity $DROP a register which is not in use.

In order to support the symbolic base register references, $DROP does not DROP the register but instead generates a USING instruction which permits the assembler to generate addressability for the XRn notation.

Example

```
$DROP   R5,R7
```

this would generate the following instructions

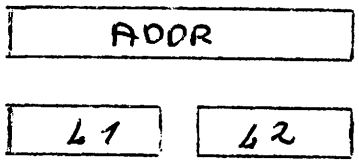
```
USING   XR5,R5
USING   XR7,R7
```

B. String Macros

Macros used to aid the passing of character strings between routines.

When programming the /360 it is often desirable to call a subroutine, giving it a character string as one of the parameters. The subroutine will read into, or write from, or otherwise use or manipulate the character string before returning to the caller. In many cases it is useful to write the subroutine such that it will work on character strings of different lengths, i.e., accept a length operand as one of its input parameters, and possibly return a different length operand to the caller if the length of the character string has been altered by the subroutine.

A method of handling this problem in a complete and general manner is to use Dope Vectors, a technique borrowed from PL/I. The format of a string dope vector is as follows:



one word: the address of the string

two half-words: the maximum and current length of the string

A string dope vector is eight bytes long. The first word contains the address of the string, or, if the string immediately follows the dope vector the first word may be zero. This word is followed by two half-words which describe the length of the string. The 1st half-word contains the maximum length of the string, i.e. the amount of core allocated to the string. The 2nd half-word contains the current length of the string, i.e. the number of data bytes currently valid in the string. Obviously, the current length must be less than or equal to the maximum length, and both are restricted to the range zero to 32K.

For example, the following instructions could be used to define a character string with a maximum length of 121 characters, a current length of 17, and an initial value of 'READER OPEN ERROR'

```
VECTOR1    DC     F'0',H'121,17' DOPE VECTOR
STRING     DC     CL121'READER OPEN ERROR'  STRING
```

If a subroutine was written to type a character string, and you wished to have the above message typed, you could simply call the subroutine (see $CALL macro), passing it the address VECTOR1 in a register. The subroutine could then determine that the string was located at VECTOR1+8 (since the first word of the dope vector is 0) and that its current length is 17 characters, and using this information it could properly type out the message.

To extend the example, another dope vector could be defined as follows

```
VECTOR2     DC      A(STRING),F'60,0'
```

This dope vector points at the same string, but describes it in a different way. VECTOR2 specifies that the maximum length of the string is 60 characters and that its length is 0, i.e. it is a null string. If you now called the string type-out subroutine, passing it the address of VECTOR2, it should either just return, or else return with an error code, depending on whether or not null strings are supposed to be used with that subroutine.

Now, suppose that we have another subroutine which reads a card whenever it is called, and passes the contents of the card back to the caller by placing the data in a string provided by the caller. If we called this subroutine, giving it the address of VECTOR1, then it should place the 80 characters from the card into the string and set the current length of the string to 80. However, if we called this read card subroutine, giving it the address of VECTOR2, then the following should happen: When the read card subroutine checks the maximum length of the string, it will find that it has been declared as only 60. Because of this the subroutine should only move the first 60 characters from the card into the string, and should set the current length to 60, and possibly return with an error code to indicate that some data has been lost because the output string was too short.

Two macros are provided to help support dope vectors. These are $DC, which allows you to declare a constant dope-vectored string, and $LDV, which is used to load the various parts of the dope vector into the general purpose registers. Detailed descriptions of these macros follow.

$DC

This macro is used to permit the declaration of a constant dope-vectored string. Its format is as follows

```
[label]      $DC    'character string'[,'string',...]
```

The operand format of this macro is similar that of the C-type DC instruction. By omitting the C specification you indicate that the character string is to be preceeded by a dope vector. A quote character within the string must be coded as two quotes. A null string may be declared by having nothing between the outer quotes. Replication factors and length modifiers are not allowed.

If any operand of $DC does not begin with a ', then an ordinary DC instruction is generated for that operand. Thus, a mixture of dope-vectored strings and other kinds of constants may be declared with one $DC.

Examples

```
DV1      $DC    'HI THERE'
DV2      $DC    '',X'88','A''B'
```

these would generate the following instructions

```
DV1       DC    A(*+8),2AL2(L'$DC1nnnn)
$DC1nnnn  DC    C'HI THERE'
DV2       DC    F'0',2H'0'
          DC    X'88'
          DC    A(*+8),2AL2(L'$DC2nnnn)
$DC2nnnn  DC    C'A''B'
```

Note: For a non-null string the following statements are generated

```
          DC    A(addr of string),2AL2(length of string)
label     DC    C'string'
```

where "addr of string" is *+8 and "length of string" is L'label.

For a null string the following statement is generated

```
          DC    F'0',2H'0'
```

i.e., a zero-length string at location zero.

$LDV (Load Dope Vector)

This macro is used as a quick method of determining the address, maximum length, and/or current length of a string, given the address of its dope vector. It is normally used at the start of a subroutine whose input parameter is a dope vector address. The format of this macro is

```
[label] $LDV    (Ra,Rm,Rc),dvaddr
```

The operand dvaddr is the address of the dope vector, usually in XRn notation. The operands Ra,Rm, and Rc are the registers which are loaded with the address, maximum length, and current length respectively from the dope vector. One or two of these three operands may be omitted if desired, in which case those parts of the dope vector are not loaded. Ra is loaded with the address of the dope vector plus 8 if the first word of the dope vector is 0. The three registers (Ra,Rm,Rc) must not be the same as the register specified in dvaddr if any.

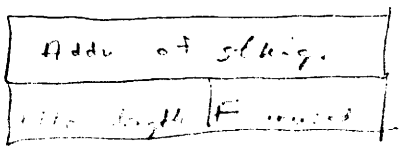
Example

```
PICKUP  $LDV    (R8,,R7),XR2 GET ADDR AND CUR LEN
```

this instruction would generate the following code

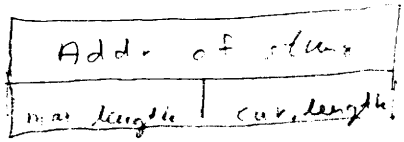
```
PICKUP  L       R8,XR2
        LTR     R8,R8
        BNZ     *+8
        LA      R8,XR2+8
        LH      R7,XR2+6
```

PL/I opt:

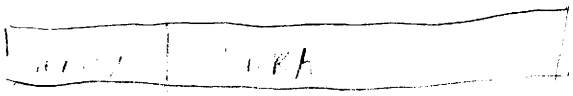


F = '0'B fixed string
= '1'B varying string

PL/I (F):



GS PUPH



C. Routine Macros

Macros used to facilitate modular, re-enterable programming.

These macros are used to make it easy to write programs modularly and re-enterably. When using this system, routines are coded in the following manner:

```
          TITLE   'routname  purpose of routine'
          Comments
routname  $ROUTINE
      Work Area Description (optional)      )
         $ENTRY                             )  DSECT
      instructions, including $CALLs        )
         $RETURN                            )
   *     EX'ED INSTRUCTIONS                 )
      executed instructions if any          )  CSECT
   *     CONSTANTS                          )
      constants if any                      )
         ~~LTORG~~                          )
```

The above sequence is repeated for each routine required to make up the complete program. Finally, the last (or only) routine is followed by the register definitions as follows:

```
          TITLE          'REGISTER DEFINITIONS'
          $SETR
          $SETX
          END
```

Each part of the above sequence of instructions will now be considered in more detail:

TITLE statement - this causes the assembler to skip to a new page and to print the specified title at the top of this page and each succeeding page until a new TITLE statement is encountered.

Comments - it is a standard practice to place comments of the start of a routine to describe the purpose of the routine, its method of operation, its limitations, etc. It is also useful to specify the input parameters, output values if any, and the error return codes if any.

$ROUTINE statement - this statement has several purposes.

a) it is used to declare the name of the whole routine.

b) it generates a DSECT instruction to "open" the definition of the routine's work area.

c) it defines an 18 word register save area in the work area.

d) a dope vector is optionally defined in the work area to describe the PARM field from the EXEC card.

Work Area Description - this section is used by you to define the contents and format of the routine's work area. Normally in this section you code labeled DS instructions and system macros which expand into control blocks (e.g. DCB) or parameter lists (e.g. MF=L OPEN). These statements are only used to reserve the necessary space in the work area and to determine the relative displacement of each label from the start of the work area. Remember that this section is within the DSECT generated by $ROUTINE, so it is not possible to declare any constants or instructions here.

$ENTRY statement - this statement performs the following services:

a) the first (or only) $ENTRY statement in a routine generates a symbol equated to the length of the work area (always 72 bytes or more due to the presence of the register save area), used in the GETMAIN. It then "closes" the definition of the work area and "opens" the definition of the body of the routine by generating a CSECT statement specifying the name of the routine.

b) Any $ENTRY statements generate the label specified on the $ENTRY and also generate an assembler ENTRY instruction to make the name known to the link-editor for inter-module resolution.

c) the $ENTRY macro then generates instructions to save the caller's registers in his save area, and to do a GETMAIN for the work area needed for this routine.

d) the register save area in this routine's work area is chained to and from the caller's save area according to OS conventions.

e) register RC is set as the base register for this routine and a $USE generated so that the assembler can assign addressability to the routine.

f) register RD is set to the address of the GETMAIN work area. This serves the dual function of pointing at this routine's register save area (as required by OS conventions), and as well acting as a base register for the routine's work area. A $USE is generated so that all the symbols declared in the Work Area Description section are addressable.

g) $ENTRY optionally generates code to initialize the PARM field dope vector in the work area. R1 is then set to point to this dope vector. The user can now access the PARM field as a dope-vectored string via R1.

h) finally, $ENTRY will optionally generate code to save the input parameter registers in core or in other registers.

Instructions - this is the main, executable part of the routine. This is the section where you actually do the computing required by the calling routine. The following conventions apply:

a) depending on how (and by whom) the routine was called, the input parameter registers may contain any of the following:

a1) R1-Rn contains the parameter values

a2) R1-Rn contains the addresses of the parameter values

a3) R1 contains the address of a list of addresses of the parameter values.

Cases a1) and a2) are usually chosen if the routine will only be called from other assembler routines (see $CALL below) or invoked by the system (via an EXEC statement). Case a3) is chosen if the routine is called from Fortran, Cobol, etc., and may also be chosen for calls from other assembler routines.

b) during execution of the routine, you must be aware of the usage of the general registers. Register RC is reserved for the program base and RD is reserved for a save area pointer and work area base. Registers RE, RF, R0, and R1 are used and/or destroyed by any system macros. Registers RE and RF are used by $CALL, in addition to R1 to Rn being possibly used for parameters and return values, depending on the subroutine called. Use of RA and RB should be avoided if possible since they may be used if the program is run under a special monitor routine. Thus, a possible register assignment might be as follows:

- long term constant registers (e.g. control block or data pointers used during all or almost all of the routine) - R9, R8, and R7
- medium term constant work registers (e.g. a register used for the same purpose across one or more $CALLs or system macros - R6, R5, and R4
- short term work registers (e.g. a register used in a computation involving about one to ten instructions with no $CALLs or system macros - R3, R2, R1, R0, RE, and RF
- save register across a $CALL - R0

This assignment is certainly not meant to be any kind of a rule, but merely a guide. Each routine will be different and require a slightly different assignment of register usage.

c) in order to call another subroutine you use the $CALL macro. To return control to the calling routine you use the $RETURN macro. See below for more information.

d) if you wish to program re-enterably you must follow the conventions outlined in Part II, section B) of these notes. Note that the use of $ROUTINE/$ENTRY has simplified this problem considerably since any control blocks, work areas, etc. which you defined in the Work Area Description section of the routine have been automatically aquired (GETMAINed) and made addressable for you. However, it is still your responsibility to initialize any part of this area before you use it.

$CALLs - the $CALL macro is used to call a subroutine from this routine. It generates code to perform the following actions:

a) it loads the address of the subroutine into RF

b) it optionally loads the parameters or the addresses of the parameters into R1 to Rn

c) it does a BALR RE,RF to the subroutine. *) specifically save the general registers since this service should be done by the subroutine. However, if this routine or any of its ascendants use the floating point registers, and if the subroutine may destroy them, then it is the calling routine's responsibility to save them before the call and to restore them afterwards. This consideration usually only applies to Fortran subroutines.

*) When CALLing a subroutine it is not necessary to ..

After the subroutine has returned it is usually proper to test the return code in RF to determine if any unusual or error conditions have been detected by the subroutine. See the next paragraph for more information on return codes.

$RETURN - this macro is used to return to the calling routine, or, in the case of the main routine, to return to who ever invoked it, usually OS. The $RETURN macro generates code to perform the following functions:

a) if a return code is specified it is placed in RF. Return codes are normally a multiple of 4 to permit their possible use in a jump table. Zero is used to signify normal completion and 4, 8, etc. to signify unusual and error conditions in increasing order of severity.

b) if return values are specified they are loaded into R1 to Rn.

c) the caller's registers are restored except for RF and possibly the return value registers.

d) the work area aquired for this routine by $ENTRY is freed by a FREEMAIN.

e) a BR RE is performed to return to the caller.

Two points should be made here. The first is that if this modular programming technique is to be used fully, then one of the rules is that every subroutine which is called must eventually return. In other words, once a routine is entered, whenever it issues a $CALL it can assume with confidence that control will eventually be returned to it, and it is the responsibility of each routine to eventually issue a $RETURN to return control to whoever called it. It is specifically forbidden for a routine to somehow branch outside itself or otherwise transfer control away from itself. This requirement imposes the necessity of having return codes and of checking them, but it pays off in providing a simpler, 'cleaner', easier-to-debug structure.

The second point is that if a subroutine has explicity aquired any resources, then it should free these resources before returning to the caller. Examples of resources include core storage and open datasets. An exception to this rule would be a subroutine whose purpose specifically is or includes the aquiring of a resource. In this case the program should be organized such that it is ensured that the resource is eventually freed. For example, if a subroutine is written whose purpose is to open a dataset, then another subroutine should be written to close the dataset and the main routine should

be organized so that if the open routine has been called, then the close routine will be called before the main routine returns to the invoker. It is not correct for a program to return to its invoker (including OS), when it still has GETMAINed core, OPENed datasets, or any other kind of resource.

Executed Instructions - since EX'ed instructions cannot be in-line, it is common to group them together at the end of the routine. A programming convention which I have found useful is to define the length field of an EX'ed instruction as *-*, thus indicating that the length comes from the register specified by the EX.

Constants - any constants used in the routine are defined here. If the program is to be re-enterable, remember that the constants must remain constant during execution of the routine.

LTORG - this statement specifies that any literals needed should be placed here. This statement should be placed after every routine, even if no literals are explicity used, since some of the coding produced by the macros use literals.

This completes the detailed over-view of the modular programming technique and how it is implemented through the four Routine Macros; $ROUTINE, $ENTRY, $CALL, and $RETURN. Note that while these macros are especially useful when writing a modular, re-enterable program, they can still be used to advantage even if writing a single non-modular, non-re-enterable program, since they provide equal or better services than the IBM SAVE and RETURN macros.

The following pages give the detailed specifications of the four Routine Macros.

$ROUTINE

This macro is used to declare the start of a routine and to "open" the definition of its work area. The format of the macro is:

```
routname   $ROUTINE    [OPTIONS=MAIN]
```

routname - This is the name of the routine. It is saved in an internal table for $ENTRY to use as the name of the CSECT.

OPTIONS=MAIN - This specifies that this routine will be invoked only by the OS initiator (via a // EXEC statement) or a program that uses a compatible calling sequence (the OS Loader and the Trace program meet this requirement), and also that the routine wishes to process the PARM field from the EXEC statement. If either of these conditions do not hold then OPTIONS=MAIN should not be specified. OPTIONS=MAIN should be specified only on the main routine. It should not be specified for subroutines. If OPTIONS=MAIN is specified then $ROUTINE allocates an extra 8 bytes in the work area for $ENTRY to use in building a dope vector to describe the PARM field.

This macro generates a DSECT to define the start of the work area and a DS 18F to define an OS-compatible register save area as the first item in the work area. Note that the DSECT name is not the same as the routname.

Example

```
START     $ROUTINE    OPTIONS=MAIN
```

this statement will generate the following instructions

```
$Wnnnn A   DSECT
           DS          18F
           DS          F,2H
```

The nnnn in the DSECT name is just an arbitrary 4-digit number which will be used by $ENTRY to determine the length of the DSECT and to declare addressability for the work area.

$ENTRY

This macro is used to specify an entry point into a routine. Its format is as follows:

```
entryname    $ENTRY      [(r1save[,r2save,...])]
```

entryname - for the first $ENTRY statement in a routine, the entryname is optional; if specified and different from the routine name, then an 'entryname EQU routname' and an 'ENTRY entryname' statement are generated following the 'routname CSECT' statement. For $ENTRY statements other than the first the entryname must be specified and must be different from the routine name and any other entrynames. In this case an 'entryname DC 0H'0' ' and an 'ENTRY entryname' are generated.

rnsave - these parameters may be coded as register names in brackets or as symbolic locations. They are used to specify that the input parameters located in R1-Rn are to be saved in less volatile registers (R9, R8, etc.) or in some location in the work area. If a register name in brackets is coded, then a 'LR rnsave,Rn' is generated, while if anything else is coded then a 'ST Rn,rnsave' is generated. The processing of the rnsave operands is the very last thing done by the $ENTRY macro.

Examples

a)

```
                $ENTRY      (PARMDOPE,(R9))
```

This entry statement causes the main part of the routine to begin. The entryname is the routine name by default. The operand specifies that, after the normal processing done by $ENTRY, the parameter in R1 is to be stored in PARMDOPE, and the parameter in R2 is to be loaded into R9. Note that R1 and R2 will still contain the parameters as well.

b)

```
ENTRY    $ENTRY
```

This statement declares ENTRY to be the name of this entrypoint. No optional saving of the parameters is requested. To invoke this routine at this entry point the $CALLstatement would be '$CALL ENTRY'.

A detailed description of the code generated by the $ENTRY macro follows:

1) If this is the 1st $ENTRY statement in the routine a '$WnnnnL EQU *-$WnnnnA' statement is generated to determine the length (symbolically) of the work area DSECT. A 'routname CSECT' statement is generated to end the work area DSECT and begin the main part of the routine.

2) The entryname parameter is processed as discussed above.

3) All in-use registers are dropped by generating a '$DROP' statement.

4) A Fortran compatible entry sequence is generated. This consists of ''B 14(RF)', and ' DC AL1(8),CL8'entryname' '. The entryname is only used by ABDUMP when printing the save area trace, and by Fortran when printing the subroutine trace-back.

5) The caller's registers are saved in his save area by doing a 'STM RE,RC,12(RD)'

6) Base register RC is set and a $USE RC,routname' statement generated to provide addressability for the routine. For the 1st $ENTRY, setting RC simply requires 'LR RC,RF'. For succeeding $ENTRY's the sequence is 'LR RC,RF','LA RF, entryname-routname' and 'SR RC,RF'. This ensures that the routine will always be running under the same base register value, regardless of which entry point was called.

7) A GETMAIN is issued to aquire the core necessary for the routine's work area.

8) Register RD is set to point at the save area/work area, and a '$USE RD,$WnnnnA' is generated to provide addressability for the work area.

9) This save area and the caller's save area are chained both ways according to OS standards. The backward chainning is required by $RETURN in order to find the calling routine. The foreward chainning is only a diagnostic aid. It permits ABDUMP to print the save area chain.

10) If OPTIONS=MAIN was specified on the $ROUTINE statement, then $ENTRY generates code which will build a dope vector in the work area to describe the PARM field, and will then set R1 to point at this dope vector. The 1st word in the dope vector will contain the address of the PARM field character string. The two half-words in the dope vector will each contain the length of the PARM field character string. These half-words will contain zero if there was no PARM field on the EXEC card. Note - if OPTIONS=MAIN was specified on the $ROUTINE statement, and the routine is invoked by a program which does not use a calling sequence compatible with the OS initiator, then a program check may occur when the $ENTRY code attempts to build the dope vector.

11) If any parameter register save operands have been coded, then they are processed as described above.

$CALL

This macro is used to call a subroutine, after optionally loading the specified parameter registers. The format of this macro is

```
[label]  $CALL     entryname[,(parm1 ,parm2,...])]
```

The entryname operand usually specifies the routname, except for routines with multiple entry points. It may optionally be coded as a register name in brackets - e.g. (R1), if the address of the routine or entry point has previously been placed in a register. Otherwise, the macro generates a

```
L      RF,=V(entryname)
```

instruction.

The parmn operands may be specified in any of three different forms.

a) Register name in brackets: In this case a LR Rn,parmn instruction is generated to set the parameter register.

b) A constant string, enclosed in quotes: In this case an in-line dope vector and character string are generated, and the parameter register is set by generating a BAL Rn,past string instruction. See example below.

c) Any other form: If the parmn operand does not begin with a quote or a left bracket, then a LA Rn,parmn instruction is generated. This can be used to load a constant as a parameter, or to set a parameter to the address of a work area or a literal.

Examples

a)

```
GO      $CALL     ENTRY,((R6),'HI THERE',25)
```

this statement would generate the following instructions:

```
GO        L        RF,=V(ENTRY)
          LR       R1,R6
          BAL      R2,$CAnnnn2
          DC       (A(*+8),2AL2(L'$DCnnnn1)
$DCnnnn1  DC       C'HI THERE'
$CAnnnn2  LA       R3,25
          BALR     RE,RF
```

b)

```
$CALL       (R5),(PARMDOPE,=X'FF',X'FF')
```

this statement would generate the following instructions:

```
LR          RF,R5
LA          R1,PARMDOPE
LA          R2,=X'FF'
LA          R3,X'FF'
BALR        RE,RF
```

In this example the subroutine whose address is in R5 is called with three parameters. The first is the address of something called PARMDOPE, the second is the address of a byte containing X'FF', and the third parameter is the constant value X'FF' itself.

$RETURN

This macro is used to set the return code, to optionally set some return registers, and then to return control to the calling routine. Its format is:

[label] $RETURN returncode[,(value1[,value2,...])]

returncode - may be specified as a decimal number or as a register name in brackets. If a register name in brackets is not specified, then a LA RF,returncode is generated. If a register name in brackets is specified, then a LR RF,returncode is generated.

valuen - these operands specify the values to be returned in registers R1,R2, etc. Each valuen may be specified as a register name in brackets or as some other quantity, for example a decimal number. If a register name in brackets is specified, then a LR Rn,valuen is generated, whereas if some other quantity is specified, then a LA Rn,valuen is generated.

After the return code and return values have been properly processed, the $PETURN macro generates code to restore the caller's registers (except for RF and the return value registers), to free the work area associated with this routine and to return to the caller.

Examples

a)

```
GETBACK   $RETURN  (RF)
```

This statement specifies that the return code has already been loaded into RF, and that there are no return calues.

b)

```
          $RETURN  12,((R9),256)
```

This statement causes RF to be set to the return code 12, R1 to be loaded with the contents of R9, and R2 to be set to the value 256, before returning control to the calling routine.

D. Switch Macros

Macros used to aid the setting and testing of bit switches.

When programming the /360 in assembler, it is often useful to set up groups of bit switches. This is facilitated through the use of the following macros:

$SWITCH	declare a byte full of bit switches
$BIT	declare a specific bit switch
$SET	set a byte full of bit switches
$ON	turn specific bit switches on
$OFF	" " " " off
$IF $IFON $IFOFF	test specific bit switches and branch according to their status

As an example, consider the following instructions:

```
            )
            )
FILE1       $SWITCH         STATUS OF SYSUT1
            $BIT        OP  OPEN
            $BIT        EOF EOF ENCOUNTERED
            $BIT        ER  SEQUENCE ERROR ENCOUNTERED
FILE2       $SWITCH         STATUS OF SYSUT2
            $BIT        OP  OPEN
            $BIT        EOF EOF ENCOUNTERED
            $BIT        ER  SEQUENCE ERROR ENCOUNTERED
            )
            )
            OPEN        (SYSUT1,INPUT,SYSUT2,INPUT)
            $SET        FILE1(OP)
            $SET        FILE2(OP)
            )
            )
            $IF         (FILE1(ER),ON),SEQER1
            )
            )
            $IF         (FILE2(OP+EOF),ON),READ1
            )
            )
EODAD2      $ON         FILE2(EOF)
```

In the above-mentioned example, two bytes are declared as switches, each containing three bit switches. This wastes five bits in each byte, but it is usually better to keep bit switches grouped logically than to try to cram eight switch bits in every byte. Next, the bytes are initialized with the OP bit on in each and all other bits off. The two $IF instructions test various bit switches and branch if they are on. Note that the second $IF requires that both OP and FOF be on in order to branch. To branch if either is on you would code switch (bit1|bit2) instead of switch (bit1+bit2). Finally, the $ON instruction is used to turn on one bit in the FILE2 switch byte.

Although each of these macros generates only one or two lines of code there are several advantages to their use. One is that they are less error-prone, especially the $IF macro, which greatly reduces the possibility of branching on the wrong condition when testing two or more switches. They also simplify the coding since you need not be concerned with the exact bit position of each switch and, in the case of $OFF, with the problem of working out the inverse bit mask. Finally, one of the useful side-effects of these macros is that each switch byte and each switch bit is referenced by name whenever it is tested or manipulated. Thus, if you have forgotten how part of the program works or are trying to track down a bug involving the switches, you can consult the XREF to determine the locations of all references to each specific switch byte or bit.

Note that these macros must be used together as a family; e.g. the use of $ON, $OFF, $IF, etc. requires the use $SWITCH and $BIT.

Detailed specifications for these macros follow.

$SWITCH

This macro is used to declare a byte full of bit switches. Its format is as follows:

```
switchname      $SWITCH     [INIT=(bit1[+bit2+...])]
```

If the INIT= operand is omitted, this instruction simply declares switchname to be one byte whose initial value is zero. If the INIT= parameter is coded, then those bits specified are on in the initial declaration of switchname.

Examples

a)

```
          FILE1     $SWITCH
```

this would generate the following instruction:

```
          FILE1     DC          X'00'
```

b)

```
          FILE1     $SWITCH     INIT=(OP+ER)
```

this would generate the following instruction:

```
          FILE1     DC          AL1(FILE1OP+FILE1ER+0+0+0+0+0+0)
```

which in turn will assemble into one byte which has two of its bits on (see $BIT).

$BIT

This macro is used to declare a bit switch. Its format is as follows:

```
        $BIT    bitname
```

The $BIT instruction must follow a $SWITCH instruction, and no more than eight $BIT instructions are allowed after each $SWITCH statement. Each $BIT statement assignes a value to a name formed by concatenating the switchname and the bitname. Thus, the number of characters in the switchname plus the number of characters in the bitname must not exceed eight. The first $BIT statement after a $SWITCH assignes the value 128 to that bitname (i.e. X'80'), the next $BIT uses 64, the next 32, etc.

Example

```
FILE1   $SWITCH
        $BIT    OP
        $BIT    EOF
```

this would generate the following instructions:

```
FILE1     DC      X'00'
FILE1OP   EQU     128
FILE1EOF  EQU     64
```

$SET

This macro is used to set all the bits in a switch byte at the same time. Its format is as follows:

```
[label]   $SET     switchname[(bit1[+bit2+...])]
```

The instruction sets all the bits that are named to one, all the others are set to zero.

Examples

a)

```
          INIT     $SET     FILE1
```

this would generate the following instruction:

```
          INIT     MVI      FILE1,X'00'
```

b)

```
                   $SET     FILE1(OP+ER)
```

this would generate the following instruction

```
                   MVI      FILE1,(FILE1OP+FILE1ER+0+0+0+0+0+0)
```

$ON

This macro is used to set one or more bits in a switch byte on. Its format is as follows:

```
[label]   $ON      switchname[(bit1[+bit2+...])]
```

If no bits are specified in the operand, all the bits in the switch byte are turned on. Otherwise, only those bits specified are turned on, all others remain unchanged.

Examples

a)

```
          EODAD1    $ON       FILE1(EOF)
```

this would generate the following instruction:

```
          EODAD1    OI        FILE1,(FILE1EOF+0+0+0+0+0+0+0)
```

b)

```
                    $ON       SWBYTE
```

this would generate the following instruction:

```
                    MVI       SWBYTE,X'FF'
```

$OFF

This macro is used to set one or more bits in a switch byte off. Its format is as follows:

```
[label]    $OFF      switchname[(bit1[+bit2+...])]
```

If no bits are specified in the operand, all the bits in the switch byte are turned off. Otherwise, only those bits specified are turned off, all others remain unchanged.

Examples

a)

```
           $OFF      SWBYTE
```

this would generate the following instruction:

```
           MVI       SWBYTE,X'00'
```

b)

```
CLOSE1     $OFF      FILE1(OP+ER)
```

this would generate the following instruction

```
CLOSE1     NI        FILE1,X'FF'-(FILE1OP+FILE1ER
                                  +0+0+0+0+0+0)
```

which turns off the OP and ER bits in FILE1, leaving the rest of the bits unchanged.

$IF

This macro is used to test a switch byte and to branch depending on whether certain switch bits are on or off. Its format is as follows:

[label] $IF (switchname[(bit1[{+|}bit2{+|}...])],{ON|OFF}),braddress

If no bits are specified in the operand, then all the bits in the switch byte are tested. If they are all on, and ON was coded, or, if they are all off, and OFF was coded, then a branch occurs to braddress.

If bits are specified in the operand, then only those bits are tested, other bits in that switch byte are ignored. If more than one bit is specified, they are seperated by a plus sign (+) which means "and", or by the vertical bar (|) which means "or". If the plus sign is used, then all the bits tested must meet the condition (ON or OFF) for the branch to occur. If the vertical bar is used, then if any of the tested bits meets the condition specified (ON or OFF) the branch will occur.

Examples

a)

```
        $IF    (SWBYTE,ON),JOEJOE
```

this will generate the following instructions:

```
        TM     SWBYTE,X'FF'    test all the bits in SWBYTE
        BO     JOEJOE          branch if all ones
```

b)

```
   TEST $IF    (FILE1(EOF),OFF),NOTEOF
```

this will generate the following instructions

```
   TEST TM     FILE1,(FILE1EOF+0+0+0+0+0+0+0)
        BZ     NOTEOF          branch if tested bit was 0
```

c)

```
$IF    (FILE1(EOF+OP),ON),FILE1OK
```

this will generate the following instructions:

```
TM     FILE1(FILE1EOF+FILE1OP+0+0+0+0+0+0)
BO     FILE1OK          branch if bits tested were 1
```

d)

```
$IF    (FILE1(EOF|ER),ON),NOFILE1
```

this will generate the following instructions:

```
TM     FILE1,(FILE1EOF+FILE1ER+0+0+0+0+0+0)
BNZ    NOFILE1          branch if either bit tested was 1
```

Notes

1) You cannot use a mixture of + and | in one $IF statement, the same operator must be used between all the bits specified.

2) If you code the braddress as a register name in brackets (i.e. (RF)), then BCR instead of a BC is generated as the second instruction.

$IFON and $IFOFF

These macros are variations of the $IF macro. They do the same thing, but have a slightly different syntax, and are a bit easier to code. Their format is as follows:

[label] {$IFON / $IFOFF} switchname[(bit1[{+ / |}bit2{+ / |}...])],braddress

Examples

These examples are the same as for $IF and will generate the same instructions:

```
a)          $IFON     SWBYTF,JOEJOF

b)    TEST  $IFOFF    FILE1(EOF),NOTEOF

c)          $IFON     FILE1(EOF+OP),FILE1OK

d)          $IFON     FILE1(EOF|ER),NOFILF1
```

E. Miscellaneous Macros

Documentation of the macros $ALIGN, $CCW, $REF, and $NULL.

This section documents four miscellaneous macros. They are:

$ALIGN - This macro is used to round the contents of a register up to a multiple of 2, 4, or 8. This operation is occasionally needed on the /360 due to the requirement that the operands of loads and stares must not be aligned on a multiple of their length (1,2,4 or 8 bytes). For example, a register might be being used as a pointer within a symbol table, in which each entry uses a variable length, but each entry had to start on a full-word boundry. In order to place the next entry, you would add the length of the previous entry to the pointer register, and then $ALIGN it up to the next full-word boundry.

$CCW - This macro is an extension of the assembler CCW instruction. Among its features is the ability to specify the CCW opcode and the flag bits by nmemonic, and the ability to specify the contents of the "option byte" - the sixth byte of the CCW.

$REF - This is a simple little macro used to create entries in the assembler's XREF. Often it occurs that two adjacent items in core can be referenced or set by one instruction which explicitly references only the first item. For example, two adjacent halfwords, labeled A and B, could be zeroed by the instruction

```
XC    A(4),A
```

The only problem with this is that at some later date, you may be changing the logic of the program, or trying to find some new bug, and because of this you may wish to find all references to the halfword B. The above instruction alters B, but there is nothing in the XREF to indicate this. Therefore, it is a good idea to place the statement

```
$REF  B
```

immediately following the above XC statement.

$NULL - This macro simply provides a convenient place to hang a label when you don't want to execute an instruction just yet.

Detailed specifications of these four macros follow.

$ALIGN

This macro is used to round the contents of a register up to a multiple of 2, 4, or 8. Its format is:

[label] $ALIGN Ra,{H/F/D}[,INC={X/Rn}][,OUT=Rm]

Ra — This is the name of the register containing the value which is to be rounded. It cannot be R0. If OUT= is not coded, then this is also the register where the result is placed.

H, F, or D - This specifies whether the contents of the register is to be half-word, full-word, or double-word aligned.

INC={X/Rn} — This is only specified if you wish to add a constant amount, or the contents of another register to the value in Ra before doing the alignment. To add a constant amount you specify INC=X, where X is any decimal or hex number. To add in the contents of another register you specify INC=(Rn).

OUT=Rm — If this operand is coded, then the result of the alignment calculations appears in Rm, and the contents of Ra are unchanged (even if INC= is coded). Rm may be R0 if desired.

The operation of the $ALIGN macro is as follows.
If the value in Ra is already correctly aligned, then it remains unaltered. Otherwise, it is rounded up to the next highest value which is properly aligned.

For example, if Ra contained 12, and a $ALIGN Ra,F was done, Ra would not be changed. However, if a $ALIGN Ra,D was done, then the value in Ra would be changed to 16.

Examples

a)

```
NEXTWORD  $ALIGN   R1,F
```

this statement would generate the following instructions:

```
NEXTWORD  LA       R1,XR1+3
          N        R1,=X'FFFFFFFC'
```

b)

```
$ALIGN   RF,F,INC=3,OUT=R0
```

this statement would generate the following instructions:

```
LA       R0,XRF+1+3
N        R0,=X'FFFFFFFE'
```

$CCW

This macro provides some extensions beyond the facilities of the assembler CCW instruction. Its format is:

[label] $CCW {(op,4) / op},address[,{(flags[,option byte]) / flag},length]

op - This is the op-code of the CCW. It may be specified by decimal or hex constant, by a symbol EQU'ed to a decimal or hex constant, ot by one of the following mnemonics: WRITE, READ, NOP, SENSE, TIC, or READBACK. In the case of TIC, if the 2nd word of the CCW is not needed the opcode may be specified as (TIC,4), in which case only four bytes are generated, and the last two operands of $CCW do not need to be coded.

address - This field may contain any value or expression which will be valid within a DC AL3.

flag - This item may be a hex or decimal number, or it may be one of the following mnemonics: CD, CC, SLI, SKIP, or PCI.

flags - This item may be a hex or decimal number, or it may be one of the following mnemonics: CD, CC, SLI, SKIP, or PCI; or, it may be several of these mnemonics seperated by commas.

option byte - This item may be a hex or decimal number, or a symbol EQU'ed to a decimal or hex constant. This value is assembled into the 6th byte of the CCW. If omitted, the value will be zero. This byte is ignored by the channel, but it is sometimes useful to be able to set it since your program may wish to use it to indicate the purpose of the CCW, or what sort of error recovery to initiate if a I/O error occurs on the CCW.

length - This field may contain any value or expression which will be valid within a DC AL2.

Examples

a)

```
        CHAIN1          $CCW        READ,BUF+00,CC,80
                        $CCW        READ,BUF+80,(CC,SLI),20
                        $CCW        (TIC,4),CHAIN2
```

b)

```
        OPEN2741        $CCW        DISABLE,0,(CC,SLI),1
                        $CCW        ENABLE,0,(SLI,5),1
```

This example assumes that DISABLE and ENABLE are symbols which have been EQU'ed to the necessary hex constants. The 2nd CCW in this example will have the constant 5 assembled into the 6th byte of the CCW.

$REF

This macro simply generates a statement which will result in a symbol being referenced in the assembler's Cross Reference table (XREF). The format of this macro is:

```
$REF        symbol[,symbol,...]
```

symbol - This may be any valid, defined symbol in the program. If desired, more than one may be coded on the same $REF statement.

This macro simply generates a 'DC 0AL4(symbol)' instruction for each symbol in the operand field of the $REF statement. The assembler processing of this instruction causes an entry to be generated for the XREF. This instruction does not cause any core to be used or any alignment to be performed.

Examples

a)

```
$REF        A
```

this statement would generate the following instructions:

```
DC          0AL4(A)
```

b)

```
$REF        A,B,C
```

this statement would generate the following instructions:

```
DC          0AL4(A)
DC          0AL4(B)
DC          0AL4(C)
```

$NULL

This macro is used to define a label at a location you wish to branch to, but you don't wish to code the first instruction at that point. Its format is:

```
[label]   $NULL
```

This macro has no operands.

The only use of this macro is at places where you wish to declare a label, but then you wish to code some $USEs, $DROPs, or comments before coding the instructions to be executed. The macro generates a DC 0H'0', which defines the label and forces half-word alignment, but does not generate any data output.

Example

```
HERE      $NULL
```

this instruction will generate the following code:

```
HERE      DC        0H'0'
```

Part IV. Reading OS MVT dumps

The information in this section is meant to be a supplement to the information found in the IBM manual "OS Programmer's Guide to Debugging", most of which is irrelevant and confusing. In fact, except for the section entitled "APEMD/SNAP Dump (Systems with MVT)" and the appendices. I would strongly advise everyone to stay away from the above-mentioned manual.

This part is divided into four sections:

A) Analyzing the Completion Code.

B) Analyzing the Execution Flow.

C) Analyzing Memory Contents.

D) Analyzing I/O Errors.

Analyzing an abend always starts with section A: Analyzing the Completion Code. In some cases this is sufficient to determine the cause of the error. In other cases it is necessary to get into sections B, C, and/or D. The type of additional analysis necessary is usually indicated by the Completion Code analysis, and in most cases is section B: Analyzing the Execution Flow.

A. Analyzing the Completion Code

The completion code is the first thing you always check when your job abnormally terminates. There are two kinds of completion codes, the User Completion Codes, denoted by Unnnn, and the System Completion Codes, denoted by Snnn.

If the completion code is a User Completion Code, then you must check the documentation on the particular processor which issued the ABEND. For example, ASMG ABENDS with a U0020 if DD cards are missing or there is not enough core, while PL/I run-time routines ABENDs with a U4000 if you manage to clobber some of the PL/I control blocks.

If the completion code is a System Completion Code, then in some cases it is all the information you need. For example, the B37,D37 or E37 abend during a linkedit means that the SYSLMOD dataset has overflowed, and it is time for you to either squish the dataset or else to allocate a larger one.

Full explanations of the System Completion Codes can be found in the manual "OS Messages and Codes". Brief explanations of the more common System Completion Codes can be found in Appendix B of the "OS Programmer's Guide to Debugging". Very brief explanations of the most common System Completion Codes follow:

S001 — Bad channel program. Go to section D.

S0Cx — Program check. x is the hex digit specifying which type of program interruption occurred. Check the POP (Principles Of Operation) manual or the green card. Go to section E.

S0F2 — Program check in SVC. You passed the SVC routine bad pointers or control blocks. Go to sections B and possibly C.

S122 — Cancelled by operator with a dump. Possibly cancelled by HASP for excessive printed or punched output.

S222 — Cancelled by operator without a dump.

S322 — Ran out of time. If not looping, re-submit with a larger time estimation.

S604,605,60A — Bad FQE. Your program clobbered a Free Queue Element (storage management control block) at some point in the past. Go to section C. This can be a hard one to find because ABDUMP sometimes refuses to dump the adjacent core. Look for buffers too short or MVC's too long.

S706 Not-executable load module. Check the linkedit output to find out why the module was marked not-executable. Either fix that problem or re-linkedit with PARM=LET.

S804,80A Ran out of core. Resubmit the job with a larger REGION=parameter.

S806 Non-existent load module. If the module is not in SYS1.LINKLIB, check that your JOBLIB or STEPLIB card is present and correct. Check that you haven't mispelled the module name.

SB37,D37,E37 Ran out of space on disk dataset. Either squish the dataset or allocate a larger dataset.

If, after having looked up the completion code, you are still unable to determine exactly what is wrong, then it is necessary to examine the ABDUMP. (If you didn't get an ABDUMP, re-run the job with a //SYSUDUMP DD SYSOUT=A card in the abending step). When analyzing the dump you are usually looking for one of three things:

B. Execution Flow - you wish to find where the last instruction executed in your program is, and what the flow of execution was that led to that point.

C. Memory Contents - you wish to look at your data areas to determine their present contents.

D. I/O Errors - you wish to examine the results of an I/O operation to determine why it ended with an I/O error.

B. Analyzing the Execution Flow

To find the last instruction executed in your program, proceed as follows. On the first page of the dump, under the heading "ACTIVE PRBS" (Request Blocks), find the last (current) or only PRB (Program Request Block). In this PRB, look at the APSW (Abend Program Status Word). If this entry is non-zero, then that is the address at which the interrupt occurred. If the APSW is zero, then the second half of the PSW in this PRB contains the address at which the interrupt occurred.

Several things must immediately be noted here. The first is that the interrupt address is the address at the end of the last instruction, not the start. To find the start of the last instruction you usually: a) find the interrupt address in the core dump, and by looking at the preceeding couple bytes you can usually see what kind of instruction was being executed; or b) un-relocate the interruption address (see later) and look at your assembly listing to determine the preceeding instruction. Note that the correct method of finding the start of the last instruction executed is to multiply the ILC (Instruction Length Code - the top two bits in the APSW or 2nd half of the PSW) by 2 and subtract this from the interruption address. However, the above two methods are usually faster.

The next thing to note is that the interrupt address may not be in your program, for example it may be 000002 (down in the low core) or C1C1C1 (up in non-existant core). In these cases the usual cause is that you have branched through a register containing garbage instead of an address. Look at the "REGISTERS AT ENTRY TO ABEND" (immediately preceeding the body of the dump) for a register that is 2, 4, or 6 less than the interrupt address. If RF looks likely, then RE probably contains the return address which will give you the last instruction executed (i.e. a L RF, garbage; BALR RE,RF was executed). If RF is not the culprit, then you may have problems. Check that your base register is correct, and un-relocate the contents for any register that looks like it is a return register (a return register is one set by a BAL or BALR instruction; they have a non-zero top byte and a valid address (address in your program) in the low 3 bytes) in order to try to localize the branch instruction that went wild.

Another thing to note is that, although IBM doesn't advertise the fact, the /75 is slightly imprecise on a few of the interrupts. Specifically, if on a S0C4 or S0C5 abend, you have found the last instruction executed, and you cannot see anything wrong with it or the contents of the registers it references, then check that the ILC is not zero. If the ILC is zero, this means that it was not the last, but the 2nd last, or possibly even the 3rd last instruction which caused the interrupt, and you must look at these instructions too in order to find the one in error. Under these circumstances (S0C4 or S0C5 and ILC=0), if the interrupt address corresponds to a labeled instruction in the source, then you may have to check the instructions which preceed each branch instruction to that label, since the /75 can sometimes execute one branch before the interrupt occurs. Fortunately, these occurrances are quite rare.

Finally, it is time to discuss how to un-relocate addresses. The quickest way usually is to find the module in the core dump (each module is dumped separately, preceeded by its name) and to note the address at which the dump starts (this will be the address printed beside the first line, possibly plus 8, 16, or 24 if the dump starts part way across the first line). Once you have this you simply subtract the address of the start of the module from the address you wish to un-relocate. If you have multiple CSECTs/module , you may have to further un-relocate this address by the offset of the active CSECT within the module, which you determine from the linkedit map. If you are using the $ROUTINE/$ENTRY macros previously described, then the entry names will appear in the right half of the dump in EBCDIC, and this may be used to help to find your un-relocated location.

The correct way to un-relocate addresses is to look at the entry marked FL-CDE in the current PRB, and then to find the CDE (contents directory element) which is at that address (the CDEs are printed following the PRBs). In this CDE you will find three things: the name (NM) of the module (in case you were not sure which module you were executing), the entry point address (EPA) of the module, and an item called the XL/MJ. The EPA of the module can usually be used to un-relocate addresses, or, if the EPA is not at the start of the module , look under the XLs for one at the address specified by the CDE's XL/MJ. The XL (extent list) contains the address (ADR) and the length (LN) of the module. Usually, the methods for un-relocating addresses described in the previous paragraph are faster than these "correct" methods.

Once you have located and un-relocated the interruption point in the present module, it is sometimes useful to try to determine how the program got to that point. This can be done at two levels, the module level and the routine level. If your program consists of multiple modules which pass control via LINK and XCTL, then the last (current) PRB may not be the only one in the list of "ACTIVE RBS". These PRBs above the last one are modules which have passed control by issuing a LINK to the next module down. Their names may be determined by finding their CDEs as described above.

When attempting to determine the flow of execution at the routine level, your main tools are the current register contents ("REGISTERS AT ENTRY TO ABEND") and the "SAVE AREA TRACE". As mentioned previously, the registers set by a BAL or BALR instruction can usually be spotted because of the non-zero top byte followed by a valid address. Unless RD (R13) has been clobbered and/or the save areas were not chained according to OS standards (typically, OS access methods are one of the things which do not conform to the OS save area standards), then you should be able to work your way back through the save area chain. RD points at the current save area, and the save area previous to that (the save area pointed at by the HSA (High Save Area) field of the current one) contains the registers belonging to the calling routine at the time it called the current routine. The fields labeled RET (Return) and EPA (entry point address) were its RE and RF respectively, while the fields labeled HSA and LSA are the save area chain pointers (high save area and low save area). If the $ROUTINE/$ENTRY macros previously described have been used, then each save area is preceeded by the name of the entry point, making it easy to see at a glance the flow of control through the routines.

The only remaining thing to mention is that there is one possible exception to some of these procedures. If the last (current) PRB is not followed by two, and only two, SVRBs, then this indicates that your program was not in control when it abended. The last (or only) two SVRBs are always ABEND and ABDUMP respectively.

If there are any SVRBs or IRBs between the last PRB and the last two SVRBs, then either an SVC and/or an asynchronous interrupt routine was in control when the abend occurred. If this is the case, then the program's current registers are not the "REGISTERS AT ENTRY TO ABEND", but are the registers saved in the RB (either SVRB or IRB) which follows the last PRB. The above procedures are correct, if this problem does not occur.

In the case where an IRB (interruption request block) was in control at the time of the abend, then you may wish to find the interrupt address in order to determine the cause of the abend. The method used is the same as for a PRB, i.e. check the APSW, and if it is zero then check the right half of the PSW to find the ILC and interrupt address.

C. Analyzing Memory Contents

Fortunately, finding your data areas is not as tricky as finding the last instruction executed. In fact, it is quite straight-foreword. Any data areas that have been defined within your module with DS or DC instructions may be found by relocating the address and finding it within the dump of the module. To relocate the address you reverse the un-relocating process, i.e. you simply add the address of the start of the module onto the address of the data area, or, if you have multiple CSECTs, then add the address of the start of the CSECT to the data offset within the CSECT.

Finding dynamically acquired core is slightly more difficult. This core is printed at the end of the dump under the heading SP 000 (subpool 0). Note that if you have multiple blocks of SP 0(the usual case) the system doesn't necessarily print them in any order (it least, not any order apparent to mere humans), so that when trying to find a certain address you may have to look at all the SP 0 blocks printed.

Determing the address of what you wish to find is another matter. Often your registers point at your control blocks and data areas. If the $ROUTINE/$ENTRY macros previously described have been used, then RD points at the current save area/work area, and the save areas of previous routines each preceed the work area belonging to that routine. The buffers associated with a file can be found from the DCBBUFCB pointer in the DCB (DCB plus 21 bytes, a 3 byte pointer). The buffers are preceeded by an eight-byte buffer control block. Often, it is easier to find buffers and work areas just by looking at the EBCDIC portion (right half) of the dump until you spot something which looks familiar.

D. Analyzing I/O Errors

When OS abends you or enters your SYNAD exit because of an "I/O error", what this actually means in 99 cases out of 100 is a programming error involving I/O. Real I/O errors (an error in the I/O device or recording media) are extremely rare with the /360s and associated equipment. While the following discussion will be of most help to persons who are programming I/O using EXCP (Execute Channel Program), it may be of some use to persons programming I/O using access methods.

The first thing to do is to find the IOB (Input Output Block) associated with the channel program which failed. If you are using EXCP you must know where the IOB is. If you are using access methods you can usually find the IOB from the DCB (Data Control Block). The DCBIOBA field (DCB+68, 4 byte pointer) should point to the IOB-8. In some cases, for example when programming in a high-level language, you may not even know where the DCB is or even which dataset was the one in error. DCBs can be found by looking at the DEBDCBAD fields in the DEBs (data extent blocks) which are printed out preceeding the core dump. The address of the DEB is found by looking at the DEB pointer in the TCB for the first DEB, then that location + 5 is a 3-byte pointer to the next DEB and so on. You have to find the address of the DEB in this manner because ABDUMP prints the DEB prefix with the DEB, and the length of the prefix is not constant. Anyway, given the address of the main portion of the DEB, the pointer to the DCB is at DEB+25 and is a 3-byte pointer. The DCB which is in error can be spotted because it willhave the 1st 2 bits (hex C0) of the DCBIFLGS byte on (DCB+44), indicating permanent error.
See "OS Programmer's Guide to Debugging", Appendix G.

Once you've found the IOB associated with the failing channel program, there are several useful fields you can look at. The first is the IOBECBCC (IOB+4, 1 byte). This byte contains the completion code for the I/O event. See "OS Programmer's Guide to Debugging", appendix E. The usual completion code for an I/O error is 41 hex. In this case the IOBCSW will be useful. The IOBCSW (IOB+8, the last 7 bytes of the channel status word) contains the address +8 of the last CCW (channel command word), the channel status bits (see POP manual or green card for interpretation), and the residual count from the last CCW. If the ECBCC was 41 and the CSW status includes unit check (hex 02 in the first status byte), then the IOBSENS bytes will be useful. These bytes (IOB+2; 1 or 2 bytes depending on device) can be interpreted from the "OS Programmer's Guide to Debugging", appendix F, or from the manual for the appropriate device.

In any case, the IOBSTART field (IOB+17, 3 byte pointer) will point to the start of the CCW chain which was executed. These CCWs can be interpreted via the green card or the manual for the appropriate device. If the chain was partially executed, then the data areas pointed at by the address fields in the CCWs may be interesting.

Part V. Programming I/O in assembler

The following notes are meant to be a supplement to Section II of the "OS Supervisor and Data Management Services" manual. Specifically, I would recommend that you read the following topics in that manual:

Section II, Part 1, "Data Set Characteristics, and "Interface With the Operation System".

Section II, Part 2, "Data Processing Techniques", and "Processing a Sequential Data Set".

Appendix B: Control Characters.

These notes are divided into four sections as follows:

A) Data set characteristics.

B) General processing techniques.

C) QSAM.

D) BSAM/BPAM.

A. Data Set Characteristics

The only additional notes necessary to the discussion in the "OS Supervisor and Data Management Services" manual are some hints on what the most common types of data sets are.

Most datasets contain card images, and therefore have a LRECL (logical record length) of 80. Datasets containing printed output usually have a LRECL of 121 or 133 and a RECFM (record format) which includes the letter A or M, indicating that each record begins with a control character. When possible, you should try to write your programs to print only 120 data characters/line. Most printers do have 132 print positions, but some have only 120 and terminals have only 130, and it is nicer to have a program that will work independent of minor details such as this. Note that there is no problem printing a LRECL=121 dataset on a printer with 132 print positions, while printing a LRECL=133 dataset on a printer with 120 print positions causes 12 data characters to be lost. The A type of control characters are far preferable to the M type, unless you happen to be a computer.

Almost all datasets are RECFM=FB, with possibly an A or M added. All datasets on disk or tape should be blocked. Blocking on tape results in increased volume capacity and processing speed. Blocking on disk results in increased capacity and greatly increased processing speed. Remember that on the avarage the disk must rotate once for each block read or written, so why not block your dataset and let your program run 40 to 80 times faster? The most common block size for LRECL=80 is BLKSIZE=3360. This gives you 42 records/block, 1 block/2311 track, or 2 blocks/2314 track.

RECFM=FBS means that the blocks are "standard", in other words, all the blocks in the dataset must be of length BLKSIZE, except possibly the last. This should not be specified for a PDS (partitioned dataset) with many small members since it tends to waste a lot of disk space. Also, if it is specified, it may be necessary to over-ride the specification if that dataset is concatenated with non-standard datasets.

This is because it is permissable to read on FBS dataset that is OPENed for FB, while reading an FB dataset which was opened for FBS results in I/O errors.

Except for the above two limitations, RECFM=FBS is recommended since it is processed somewhat faster than RECFM=FB.

B. General Processing Techniques

Most of the points mentioned in this section are discussed in the "OS Supervisor and Data Management Services" manual, but some of them are not properly emphasised. The purpose of this section is to present again the more important optional services that your program can be made to include. To the person writing a specialized program that will only have limited use, the inclusion of these extra services are optional, depending on whether you feel they are necessary or not. To the person who is writing a general program which will receive wide use, the inclusion of these services should be mandatory since it results in a more flexible and usable program.

This section is divided into four sub-sections under the following headings:

1. Supporting Device Independence
2. Supporting Optional Blocking
3. Supporting Concatenation of Unlike Devices
4. Providing Error Diagnostics.

1. Supporting Device Independence

If you write your program so that it is device independent, then the device on which the dataset resides can be specified on the // DD (data definition) card at run time, without any need for modifying or re-assembling the program processing the dataset. As an example, suppose you write a simple program which reads BCD (7094) cards on SYSIN, translates them to EBCDIC (/360), and writes them out on SYSPUNCH. If your program is not written device independent, it might require that SYSIN must be a card reader and SYSPUNCH must be a card punch. However, by supporting device independence, you could make the program work whether SYSIN was a card reader, a magnetic tape dataset, or a disk dataset, and also let SYSPUNCH be a card punch, a tape dataset or a disk dataset, in any combination. The program is now much more useful since it can be used to copy + translate card decks onto tape or disk, or to copy + translate tape onto cards or tape or disk, etc., as well as the original purpose of duplicating + translating card decks.

There are only two rules to remember in order to support device independence on sequential datasets that are only read or only written. The first is that you must not use the DEVD parameter in the DCB (data control block) macro for the dataset; and the second is that you cannot use the CNTRL macro, and should not specify its use in the MACRF parameter of the DCB.

Since CNTRL is not usable, we must now consider what is the correct method of producing page control for datasets destined for the printer. The simplest solution is to not provide any page control at all, but this is generally not accaptable because nobody likes printed output that goes on, page after page, without any spacing over the perforations between pages. Also, it is much nicer to start printing output at the top of the next page than to have the first line of output immediately follow the JCL (Job Control Language) or OS allocation messages.

The best (and usual) method is to use A or M control characters. This is done by setting the LRECL to the number of data character plus 1, specifying A or M in the RECFM parameter, and prefixing each line written with an A or M control character. Assuming the use of A characters, a simple page control method might be to prefix the first line and every 60th succeeding line with a "1" (causes skip to a new page), and to prefix all other lines with a blank (causes normal printing).

A simple example of the coding that could be used to do this follows (it is assumed the output line has already been placed in LINE):

```
*           THIS IS THE PRINT ROUTINE
PRINT       LH    R0,NOLINES          FETCH NO LINES LEFT
            BCT   R0,NOTNEW           DECREMENT AND TEST
            MVI   ASA,C'1'            SET TO SKIP
            LA    R0,60               SET 60 LINES LEFT
NOTNEW      STH   R0,NOLINES          SAVE NO LINES LEFT
            PUT   OUTPUT,ASA          PRINT THE LINE
            MVI   ASA,C' '            SET FOR NORMAL PRINT
            (
            (
OUTPUT      DCB   DSORG=PS,DDNAME=SYSPRINT,      X
                  MACRF=PM,RECFM=FA,             X
                  LRECL=121,BLKSIZE=121
NOLINES     DC    H'1'                NO LINES LEFT ON PAGE
ASA         DS    C                   ASA CONTROL CHARACTER
LINE        DS    CL120               PRINT LINE
            (
            (
```

2. Supporting Optional Blocking

One thing to be noted about the immediately preceeding example is that, though it does support device independence (SYSPRINT could be allocated to a printer, tape, or disk dataset), it does not allow blocking. Because the BLKSIZE is specified in the DCB, it is not possible for either the DD card or the data set label to alter it. As mentioned previously, all disk and tape datasets should be blocked, so it is therefore desirable to permit the user to specify the BLKSIZE on the SYSPRINT DD card or in the dataset label, but it is also desirable not to require that he must specify the blocksize if he doesn't want to, for example when the output device is a printer (it's much easier to say //SYSPRINT DD SYSOUT=A than //SYSPRINT DD SYSOUT=A,DCB=BLKSIZE=121).

The way to do this is with a DCB exit list that specifies a data control block exit routine. This data control block exit routine will be entered by the OS OPEN routines after they have finished merging the DD card and dataset label parameters into the DCB (but before they have used any of this information). This gives you a last-minute chance, in the data control block exit routine, of supplying or modifying any of the fields in the DCB before the dataset is completely OPENed.

The following code is an example of using a data control block exit routine to determine if the BLKSIZE field has been specified by the user on his DD card or dataset label, and if not, then the BLKSIZE is set equal to the LRECL (i.e. the dataset is assumed to be unblocked; blocking factor=1).

```
            (
            (\
            OPEN        (OUTPUT,OUTPUT)
            (
            (
*     THE FOLLOWING CODE IS THE DATA CONTROL BLOCK
*     EXIT ROUTINE
EXOUT       DC          0F'0',X'85',AL3(OUTEXIT)
OUTEXIT     $NULL
*           ON ENTRY TO THIS ROUTINE, RF IS THE BASE AND
*              R1 IS THE DCB POINTER
            $USE        RF,OUTEXIT          BASE REG IS SET BY OPEN
            CLC         XR1+DCBBLKSI(2),=H'0'     TEST IF BLKSIZE
                                                    STILL 0
            BNER        RE                  RETURN IF SPECIFIED
            MVC         XR1+DCBBLKSI(2),XR1+DCBRECL   SET BLKSIZE
                                                    EQUAL LRECL
            BR          RE                  RETURN
```

```
            $DROP     RF
*           DCB       FIELD OFFSETS
DCBBLKSI    EQU       62                  BLOCK SIZE HALF-WORD
DCBLRECL    EQU       82                  RECORD LENGTH HALF-WORD
            (
            (
OUTPUT      DCB       DSORG=PS,DDNAME=SYSPRINT,      X
                      MACRF=PM,RECFM=FBA,            X
                      LRECL=121,EXLST=EXOUT
            (
            (
```

3. Supporting Concatenation of Unlike Devices

OS permits input datasets to be "concatenated" that is, several sequential datasets and/or members of PDSs to be processed as if it were one long sequential dataset. This can be a very handy feature sometimes. Unfortunately, OS makes a distinction between what it calls concatenating of "like" devices and concatenating of "unlike" devices. Concatenating of like devices is supported by OS without any extra work on your part at all, in fact, your program is totally unaware that it has happened. Unfortunately, concatenation of unlike devices requires that you place some extra coding in your program in order to make it work. Concatenating of unlike devices is discussed in a cryptic location in the "OS Supervisor and Data Management Services" manual; consult the index to find it.

As an example of the usefulness of this feature, consider the fact that ASMG supports concatenation of unlike devices on SYSIN. This permits you to give ASMG an input dataset which partly comes from the card reader (DD *) and partly from some disk dataset. Another example is the linkeditor program which supports concatenation of unlike devices on SYSLIN, allowing you to give it a mixture of object datasets and card image control statements. Note that if you attempt to concatenate unlike devices with a program that doesn't support it, then an abend results.

In order to support concatenation of unlike devices on your input datasets, it is necessary to have a data control block exit routine (same as in previous section) which, in addition to supporting optional blocking if desired, must also set a re-read switch whenever it is entered. In addition, an extra bit must be set in the DCB to indicate that concatenation of unlike devices is being supported, and each GET (or READ/CHECK) must be followed by a test of the re-read switch; if it is on the GET must be re-issued.

The following coding shows how this might be done for an input dataset called SYSIN:

```
            (
            (
            OPEN        (INPUT,INPUT)
            (
            (
*     THIS IS THE READ ROUTINE
READ        $OFF        INSW(CAT)          TURN OFF RE-READ SW
            GET         INPUT,CARD         GET NEXT CARD
            $IFON       INSW(CAT),READ     RE-READ IF NECESSARY
            (
```

```
                (
*     THE FOLLOWING CODE IS THE DATA CONTROL BLOCK
*           EXIT ROUTINE
EXIN        DC          0F'0',X'85',AL3(INEXIT)
INEXIT      $NULL
*     ON ENTRY TO THIS ROUTINE, RF IS THE BASE
*           AND R1 IS THE DCB POINTER
            $USE        RF,INEXIT
            OI          XR1+DCBOFLGS,X'08'    SET UNLIKE CONCATE-
                                                  NATION BIT
            $ON         INSW(CAT)             SET RE-READ SW
      processing for support of optional blocking, same
            as in previous example
            BR          RE                    RETURN
            $DROP       RF
*           DCB         FIELD OFFSETS
DCBOFLGS    EQU         48                    OPEN FLAGS BYTE
            (
            (
INPUT       DCB         DSORG=PS,DDNAME=SYSIN,
                        MACRF=GM,RECFM=FB,
                        LRECL=80,EXLST=EXIN
INSW        $SWITCH                           INPUT SWITCHES
            $BIT        CAT                   RE-READ SWITCH
CARD        DS          CL80                  INPUT CARD
```

4. Providing Error Diagnostics

One of the most frustrating things that can happen to you is to try to run some program, and to have it abend without giving any indication whatsoever concerning what was wrong. In order that you do your part in reducing the number of programs of this nature in existence, you should do the following two things whenever you are coding I/O routines in assembler:
1) check each OPEN to ensure that it did in fact open, and
2) code a SYNAD exit to get control of I/O errors, and write out the SYNADAF information.

It is necessary to check that OPENs worked because if the user forgets or misspells the DD card that is to be opened, then OPEN returns to the user without issuing a diagnostic or giving any indication that it failed. The only way you can find out if the OPEN worked or not is to test a bit in the DCB. The usual procedure followed when this bit is off is to print or type a message informing the user that his DD card is missing. If this check is not performed, an S0C6 abend occurs on the first GET or PUT done on the DCB.

Coding a SYNAD exit causes OS to give control to you whenever a permanent I/O error occurs. By issuing the SYNADAF macro, you can get some information about the nature of the error. This information can then be printed out for the user. The usual procedure then is to CLOSE the dataset and return to the calling program with an error code.

The following coding includes examples of both of the above error checking procedures:

```
                (
                (
                OPEN        (DCB,INPUT)
                TM          DCB+DCBOFLGS,X'10'    TEST IF OPEN OK
                BZ          OPENIKKE              BR IF NOT
                (
                (
*               ERROR ROUTINES
*
*               FAILURE TO OPEN
OPENIKKE        WTO         'UNABLE TO OPEN SYSIN. CHECK CONTROL CARD'.
                LA          RF,8                  RETURN CODE
                B           RETURN                RETURN
*
*               I/O ERROR
```

```
IOERROR     SYNADAF   ACSMETH=QSAM          DO ANALYSIS
            MVC       IOWTO+19+8(60),XR1+68     MOVE MSG INTO WTO
IOWTO       WTO       'I/O ERROR. SYNADAF=******** X
                      ************************** X
                      **************************'
            CLOSE     DCB                   CLOSE THE DATASET
            FREEPOOL  DCB                   FREE THE BUFFER POOL
            SYNADRLS                        RELEASE SYNADAF CORE
            LA        RF,12                 RETURN CODE
RETURN      $RETURN   (RF)                  RETURN WITH ERROR CODE
            (
            (
            (
DCB         DCB       DSORG=PS,DDNAME=SYSIN,      X
                      MACRF=GM,RECFM=FB,          X
                      LRECL=80,EXLST=EXIN,        X
                      SYNAD=IOERROR
            (
            (
```

C. QSAM

This section simply emphasizes some of the major points concerning QSAM, and gives some hints on its use.

QSAM (Queued Sequential Access Method) is a very good access method, and should be used whenever possible. It provides all the facilities needed for simple reading or writing of a sequential dataset. Subject to the limitations mentioned in section B.1. it provides complete device independence. For blocked datasets it automatically handles the blocking/deblocking for you; you do GETs or PUTs of logical records and never worry about the blocking. Multi-buffering is also supported and completely taken care of by QSAM. If you don't specify BUFNO (buffer number) in the DCB, then QSAM assumes 2. If you wish more buffers you need only specify it in BUFNO parameter. The buffer pool is automatically gotten at OPEN time.

About the only special thing you need to remember about using QSAM is that after you CLOSE the dataset you should issue a FREEPOOL macro to free the buffer pool which was automatically gotten by the OPEN. For some unknown reason, IBM chose to not automatically free this buffer pool. An example of doing this was included in the previous section.

D. BSAM/BPAM

You should only use BSAM or BPAM when you have to do things not supported by QSAM, for instance process multiple members of a PDS using FIND, BLDL, and/or STOW, or jumping around within a dataset using NOTE/POINT.

When you use BSAM/BPAM you no longer have any of the nice QSAM features at your disposal. If you wish to process blocked datasets, then you must write your own blocking or deblocking routines. You have to manage the buffers yourself, and for that reason it's usually better to not specify BUFNO in the DCB; simply do a GETMAIN for the buffer(s) and later FREEMAIN them. Note that if you do specify a BUFNO, then OPEN will get a buffer pool for you, but, like QSAM, CLOSE will not free it, so you must explicity issue a FREEPOOL on the DCB.

Multi-buffering is possible, but remember that you must use multiple DECBs (the MF=L part of a READ or WRITE), and handle the buffer switching yourself.

Part VI. Examples

Two sample programs are shown on the following pages. There are comments in them which explain what they do and how they do it. The two programs are similar, but the second one is more flexible, includes all of the optional dataset facilities discussed in the previous section, and is re-enterant.

```
//A9900034 JOB (0001,NEU,50,1),'EXAMPLE NO 1',MSGLEVEL=1,REGION=150K
//         EXEC ASMCLG,PARM.CO='TEST'
//ASM.SYSIN DD *
         TITLE    'RESEQ          RESEQUENCE A CARD DECK   VERSION 1'   EX100001
*                                                                       EX100002
*        THIS PROGRAM READS CARDS ON SYSIN. IT PLACES A NEW SEQUENCE    EX100003
*          NUMBER IN COLUMNS 73-80. IF A DECK ID HAS BEEN SUPPLIED VIA  EX100004
*          THE EXEC CARD PARM FIELD, THEN IT IS PLACED IN COLUMNS       EX100005
*          73-NN, WHERE NN DEPENDS ON THE LENGTH OF THE PARM FIELD.     EX100006
*        FINALLY, EACH CARD IS PRINTED AND PUNCHED.                     EX100007
*                                                                       EX100008
*        THE LENGTH OF THE PARM FIELD DETERMINES HOW MANY COLUMNS WILL  EX100009
*          CONTAIN THE DECK ID AND HOW MANY COLUMNS WILL CONTAIN        EX100010
*          SEQUENCE NUMBER DIGITS.                                      EX100011
*        FOR EXAMPLE                                                    EX100012
*                  IF NO PARM FIELD          73-80 SEQ                  EX100013
*                  IF 4-CHAR PARM FIELD      73-76 ID,  77-80 SEQ       EX100014
*                  IF 8-CHAR PARM FIELD      73-80 ID                   EX100015
*                                                                       EX100016
*        INPUT     R1  ADDR OF PARM FIELD POPE VECTOR (SUPPLIED BY      EX100017
*                        &ENTRY BECAUSE OPTIONS=MAIN WAS SPECIFIED)     EX100018
*                                                                       EX100019
*        OUTPUT    NONE                                                 EX100020
*                                                                       EX100021
*        RETURN    0  OK                                                EX100022
*                  4  PARM LENGTH GT 8                                  EX100023
*                  8  UNABLE TO OPEN DATASETS                           EX100024
*                                                                       EX100025
*        LIMITATIONS  ALL THREE DATASETS MUST BE NOT BLOCKED            EX100026
*                     SYSIN MUST NOT BE CONCATENATED ON UNLIKE DEVICES  EX100027
*                     THE BUFFERS GOTTEN FOR THE DATASETS ARE NOT FREED EX100028
*                                                                       EX100029
*        ATTRIBUTES NON-REENTRANT,NON-SERIALLYREUSABLE                  EX100030
*                                                                       EX100031
RESEQ    &ROUTINE  OPTIONS=MAIN                                         EX100032
         &ENTRY    ((R1))                                               EX100033
*        LOAD THE ADDRESS AND LENGTH OF THE PARM FIELD                  EX100034
         ALOV      (R0,,R0),VR1          ADDR TO R0, CURRENT LEN TO R0  EX100035
         CH        R0,=H'8'              TEST IF LENGTH GT 8            EX100036
         BH        RETURN4               BR IF SO                       EX100037
*        OPEN THE DATASETS                                              EX100038
         OPEN      (SYSIN,INPUT,SYSPRINT,OUTPUT,SYSPUNCH,OUTPUT)        EX100039
*        CHECK THAT THEY OPENED OK                                      EX100040
         TM        SYSIN+DCBOFLGS,X'10' CHECK SYSIN IS OPEN             EX100041
         BZ        RETURN8               BR IF NOT                      EX100042
         TM        SYSPRINT+DCBOFLGS,X'10' CHECK SYSPRINT IS OPEN       EX100043
         BZ        RETURN8               BR IF NOT                      EX100044
         TM        SYSPUNCH+DCBOFLGS,X'10' CHECK SYSPUNCH IS OPEN       EX100045
         BZ        RETURN8                                              EX100046
*        INITIALIZE OUTER LOOP FOR 60 LINES/PAGE                        EX100047
NEWPAGE  MVI       ASA,C'1'              SET TO SKIP TO NEW PAGE        EX100048
         LA        R7,60                 SET LOOP COUNTER TO 60         EX100049
*        READ A CARD AND SET THE SEQUENCE FIELD                         EX100050
NEXTCARD GET       SYSIN,BUFFER          READ A CARD                    EX100051
         AP        NUMBER(6),=P'1'       ADD ONE TO PREVIOUS SEQ NUMBER EX100052
```

```
         UNPK      SEQNO(8),NUMBER(5)   PLACE IT ON THE CARD                EX100082
         OI        SEQNO+7,X'F0'        CORRECT ZONE ON LAST DIGIT         EX100083
         LTR       R1,R8                PARM LENGTH TO R1 AND TEST         EX100084
         BZ        PRINT                BR IF PARM LENGTH ZERO             EX100085
         BCTR      R1,0                 (R1)-1 FOR EX INSTRUCTION          EX100086
         EX        R1,IDMOVE            MOVE THE PARM FIELD TO THE CARD    EX100087
*        PRINT AND PUNCH THE CARD                                          EX100088
PRINT    PUT       SYSPRINT,ASA         PRINT, USING ASA CONTROL CHAR      EX100089
         PUT       SYSPUNCH,BUFFER      PUNCH WITHOUT CONTROL CHARACT      EX100090
*        TEST FOR END OF PAGE                                              EX100091
         MVI       ASA,C' '             SET FOR NORMAL PRINT ON NEXT       EX100092
         BCT       R7,NEXTCARD          GET NEXT CARD IF NOT END OF PAGE   EX100093
         B         NEWPAGE              START A NEW PAGE                   EX100094
*        END OF FILE ON SYSIN ENCOUNTERED                                  EX100095
SYSINEOF SR        R4,R4                SET RETURN CODE TO 0               EX100096
         B         CLOSE                GO CLOSE DATASETS                  EX100097
*        INCORRECT LENGTH OF PARM FIELD                                    EX100098
RETURN4  WTO       'RESEQ - PARM LENGTH GT 8'                              EX100099
         LA        R4,4                 SET RETURN CODE                    EX100100
         B         RETURN               .                                  EX100101
*        ONE OR MORE DATASETS NOT OPENABLE                                 EX100102
RETURN8  WTO       'RESEQ - UNABLE TO OPEN DATASETS'                       EX100103
         LA        R4,8                 SET RETURN CODE                    EX100104
*        CLOSE ALL OPEN DATASETS                                           EX100105
CLOSE    CLOSE     (SYSIN,,SYSPRINT,,SYSPUNCH,)                            EX100106
*        RETURN TO CALLER                                                  EX100107
RETURN   &RETURN   (R6)                                                    EX100108
DCBOFLGS EQU       48                   OPEN FLAGS BYTE OFFSET IN DCB      EX100109
*        EXTED INSTRUCTIONS                                                EX100110
IDMOVE   MVC       SEQNO(*-*),X89       MOVE DECK ID                       EX100111
*        NON-RE-ENTRANT WORK AREA                                          EX100112
ASA      DS        C                    PRINTER CONTROL CHARACTER          EX100113
BUFFER   DS        CL72                 CARD COLS 1-72                     EX100114
SEQNO    DS        CL8                  CARD COLUMNS 73-80                 EX100115
NUMBER   DC        PL5'0'               CURRENT SEQUENCE NUMBER            EX100116
*        DCB'S                                                             EX100117
SYSIN    DCB       DDNAME=SYSIN,DSORG=PS,MACRF=GM,LRECL=80,BLKSIZE=80,     XEX100118
               RECFM=F,EODAD=SYSINEOF                                      EX100119
SYSPRINT DCB       DDNAME=SYSPRINT,DSORG=PS,MACRF=PM,LRECL=81,             XEX100120
               BLKSIZE=81,RECFM=FA                                         EX100121
SYSPUNCH DCB       DDNAME=SYSPUNCH,DSORG=PS,MACRF=PM,LRECL=80,             XEX100122
               BLKSIZE=80,RECFM=F                                          EX100123
*        CONSTANTS                                                         EX100124
         LTORG                                                             EX100125
         TITLE     'REGISTER DEFINITIONS'                                  EX100126
         &SETR                                                             EX100127
         &SETX                                                             EX100128
         END       RESEQ                                                   EX100129
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSPUNCH DD SYSOUT=B
//GO.SYSIN DD   *
         PLACE CARDS TO BE RESEQUENCED HERE
/*
```

```
//A999903A JOB (0031,'HEU,50,2),'EXAMPLE NO 2',MSGLEVEL=1,REGION=150K
//         EXEC ASMGCLG,PARM.ASM='RENT',PARM.GO='TEST'
//ASM.SYSIN DD  *
         TITLE    'RESEQ          RESEQUENCE A CARD DECK   VERSION 2'  EX200001
*                                                                       EX200002
*        THIS PROGRAM READS CARDS ON SYSIN. IT PLACES A NEW SEQUENCE    EX200003
*          NUMBER IN COLUMNS 73-80. IF A DECK ID HAS BEEN SUPPLIED VIA  EX200004
*          THE EXEC CARD PARM FIELD, THEN IT IS PLACED IN COLUMNS       EX200005
*          73-NN, WHERE NN DEPENDS ON THE LENGTH OF THE PARM FIELD.     EX200006
*        FINALLY, EACH CARD IS PRINTED AND/OR PUNCHED.                  EX200007
*                                                                       EX200008
*        THE LENGTH OF THE PARM FIELD DETERMINES HOW MANY COLUMNS WILL  EX200009
*          CONTAIN THE DECK ID AND HOW MANY COLUMNS WILL CONTAIN        EX200010
*          SEQUENCE NUMBER DIGITS.                                      EX200011
*        FOR EXAMPLE                                                    EX200012
*                  IF NO PARM FIELD          73-80 SEQ                  EX200013
*                  IF 4-CHAR PARM FIELD      73-76 ID,  77-80 SEQ       EX200014
*                  IF 8-CHAR PARM FIELD      73-80 ID                   EX200015
*                                                                       EX200016
*        EITHER SYSPRINT OR SYSPUNCH (BUT NOT BOTH) MAY BE OMITTED IF   EX200017
*          DESIRED, HOWEVER A WARNING MESSAGE WILL BE TYPED.            EX200018
*        IF SYSIN IS MISSING OR IF BOTH SYSPRINT AND SYSPUNCH ARE       EX200019
*          MISSING THEN THE RUN WILL BE ABORTED WITH A RETURN CODE 8.   EX200020
*                                                                       EX200021
*        SYSIN MAY BE CONCATENATED ON UNLIKE DEVICES IF DESIRED.        EX200022
*                                                                       EX200023
*        ANY OR ALL OF THE THREE DATASETS MAY BE BLOCKED IF DESIRED.    EX200024
*                                                                       EX200025
*        INPUT     R1 - ADDR OF PARM FIELD DOPE VECTOR (SUPPLIED BY     EX200026
*                         &ENTRY BECAUSE OPTIONS=MAIN WAS SPECIFIED)    EX200027
*                                                                       EX200028
*        OUTPUT    NONE                                                 EX200029
*                                                                       EX200030
*        RETURN    0  OK                                                EX200031
*                  4  PARM LENGTH GT 8                                  EX200032
*                  8  UNABLE TO OPEN DATASETS                           EX200033
*                  12 I/O ERROR ON ONE OF THE DATASETS                  EX200034
*                                                                       EX200035
*        ATTRIBUTES RE-ENTRANT                                          EX200036
*                                                                       EX200037
RESEQ    &ROUTINE  OPTIONS=MAIN                                         EX200038
*                                                                       EX200039
*        WORK AREA (WITHIN DSECT)                                       EX200040
*                                                                       EX200041
ASA      DS        C                 PRINTER CONTROL CHARACTER          EX200042
BUFFER   DS        CL72              CARD COLS 1-72                     EX200043
SEQNO    DS        CL8               CARD COLUMNS 73-80                 EX200044
NUMBER   DC        PL5'0'            CURRENT SEQUENCE NUMBER            EX200045
SWITCH   &SWITCH                     SWITCH BITS                        EX200046
         &BIT      PP                PRINTER OPEN                       EX200047
         &BIT      PU                PUNCH OPEN                         EX200048
         &BIT      RR                RE-READ INPUT FOR CONCATENATION    EX200049
ERRDOPE  &DS       0F                DOPE VECTOR FOR SYNAD MESSAGE      EX200050
*                                                                       EX200051
         &ENTRY    (1,R1)                                               EX200052
*        LOAD THE ADDRESS AND LENGTH OF THE PARM FIELD                  EX200053
         &LOV      (RR,,RR),XPL      ADDR TO RR, CURRENT LTH TO RR      EX200054
         CH        RR,=H'8'          TEST IF LENGTH GT 8                EX200055
         BNH       INIT              BR IF NOT                          EX200056
         &CALL     WTO,('RESEQ - PARM LENGTH GT 8',0) TYPE ERROR        EX200057
```

```
         LA        R3,4                    SET RETURN CODE                  EX200058
         B         RETURN                  RETURN TO CALLER                 EX200059
*        INITIALIZE THE WORK AREA                                           EX200060
INIT     ZAP       NUMBER,=P'0'            ZERO THE CURRENT NUMBER          EX200061
         XSET      SWITCH                  ZERO THE SWITCH BITS             EX200062
*        OPEN SYSIN                                                         EX200063
         XCALL     OPEN,(INPUT,INLEN,0)    OPEN THE SYSIN DCB               EX200064
         LR        R6,R1                   SAVE ITS ADDRESS IN R6           EX200065
         LTR       RF,RF                   TEST RETURN CODE                 EX200066
         LA        R3,8                    RETURN CODE FOR NOT OPEN         EX200067
         BNZ       RETURN                  BR IF NOT OPEN                   EX200068
*        OPEN SYSPRINT                                                      EX200069
         XCALL     OPEN,(PRINT,PRLEN,1)    OPEN THE SYSPRINT DCB            EX200070
         LR        R5,R1                   SAVE ADDRESS OF DCB IN R5        EX200071
         LTR       RF,RF                   TEST RET CODE FOR OPEN OK        EX200072
         BNZ       *+8                     BR IF NOT OPEN                   EX200073
         XON       SWITCH(PR)              TURN ON PRINTER SWITCH           EX200074
*        OPEN SYSPUNCH                                                      EX200075
         XCALL     OPEN,(PUNCH,PULEN,1)    OPEN THE SYSPUNCH DCB            EX200076
         LR        R4,R1                   SAVE ADDRESS OF DCB IN R4        EX200077
         LTR       RF,RF                   TEST IF OPENED OK                EX200078
         BNZ       *+8                     BR IF OPEN FAILED                EX200079
         XON       SWITCH(PU)              TURN ON PUNCH SW                 EX200080
*        TEST IF BOTH PRINTER AND PUNCH FAILED TO OPEN                      EX200081
         LA        R3,8                    RETURN CODE IN CASE TRUE         EX200082
         XIFOFF    SWITCH(PR+PU),CLOSERD GO CLOSE READER AND RET IF SO      EX200083
*        INITIALIZE OUTER LOOP FOR 60 LINES/PAGE                            EX200084
NEWPAGE  MVI       ASA,C'1'                SET TO SKIP TO NEW PAGE          EX200085
         LA        R7,60                   SET LOOP COUNTER TO 60           EX200086
*        READ A CARD AND SET THE SEQUENCE FIELD                             EX200087
NEXTCARD XOFF      SWITCH(RR)              RESET THE RE-READ SWITCH         EX200088
         GET       (R6),BUFFER             READ A CARD FROM SYSIN           EX200089
         XIFON     SWITCH(RR),NEXTCARD DO THE READ AGAIN IF REREAD SW        EX200090
         AP        NUMBER,=P'1'            ADD ONE TO PREVIOUS SEQ NUMBER   EX200091
         UNPK      SEQNO,NUMBER            PLACE IT ON THE CARD             EX200092
         OI        SEQNO+7,X'F0'           CORRECT ZONE ON LAST DIGIT       EX200093
         LTR       R1,R8                   PARM LENGTH TO R1 AND TEST       EX200094
         BZ        PUTPR                   BR IF PARM LENGTH ZERO           EX200095
         BCTR      R1,0                    (R1)-1 FOR EX INSTRUCTION        EX200096
         EX        R1,IOMOVE               MOVE THE PARM FIELD TO THE CARD  EX200097
*        POINT AND/OR PUNCH THE CARD                                        EX200098
PUTPR    XIFOFF    SWITCH(PR),PUTPU        SKIP PRINTING IF SYSPRINT MISSED EX200099
         PUT       (R5),ASA                PRINT, USING ASA CONTROL CHAR    EX200100
PUTPU    XIFOFF    SWITCH(PU),TEST         SKIP PUNCHING IF NO SYSPUNCH     EX200101
         PUT       (R4),BUFFER             PUNCH WITHOUT CONTROL CHARACT    EX200102
*        TEST FOR END OF PAGE                                               EX200103
TEST     MVI       ASA,C' '                SET FOR NORMAL PRINT OF NEXT     EX200104
         BCT       R7,NEXTCARD             GET NEXT CARD IF NOT END OF PAGE EX200105
         B         NEWPAGE                 START A NEW PAGE                 EX200106
*        END OF FILE ON SYSIN ENCOUNTERED                                   EX200107
INPUTEOF SR        R3,R3                   SET RETURN CODE TO 0             EX200108
CLOSEALL XIFOFF    SWITCH(PU),CLOSEPR      DON'T CLOSE PUNCH IF NOT OPEN    EX200109
         XCALL     CLOSE,((R4),PULEN)      CALL CLOSE SUBROUTINE TO CLOSE   EX200110
CLOSEPR  XIFOFF    SWITCH(PR),CLOSERD      SKIP CLOSING SYSPRINT IF NOT OP  EX200111
         XCALL     CLOSE,((R5),PRLEN)      CALL CLOSE SUB TO CLOSE IT       EX200112
CLOSERD  XCALL     CLOSE,((R6),INLEN)      CLOSE THE SYSIN DCB              EX200113
*        RETURN TO CALLER                                                   EX200114
RETURN   XRETURN   (R3)                    WITH RETURN CODE IN R3           EX200115
*                                                                           EX200116
*        THIS ROUTINE IS ENTERED DURING OPENS ON EACH DCB                   EX200117
```

```
*                                                                       EX200118
EXLIST   DC          0F'0',X'85',AL3(EXITROUT) DCB EXIT LIST            EX200119
*                                                                       EX200120
EXITROUT &NULL                                 DATA CONTROL BLOCK EXIT ROUTINE  EX200121
         &USE        R1,IHADCB                 R1 IS DCB POINTER  USE IBM DSECT EX200122
         &USE        RE,*                      RE IS BASE REGISTER              EX200123
*        THE FOLLOWING FOUR OPERATIONS SUPPORT CONCATENATION ON                  EX200124
*          UNLIKE DEVICES FOR SYSIN.  THEY ARE EFFECTIVELY IGNORED               EX200125
*          FOR THE OUTPUT DATASETS (SYSPRINT AND SYSPUNCH).                      EX200126
         TM          DCBOFLGS,X'08'            TEST FOR FIRST ENTRY             EX200127
         BZ          *+8                       SKIP NEXT INST IF SO             EX200128
         &ON         SWITCH(RR)                SET RE-READ SW                   EX200129
         OI          DCBOFLGS,X'08'            TURN ON UNLIKE CONCAT            EX200130
*        THE REST OF THIS EXIT ROUTINE IS USED TO SUPPORT OPTIONAL               EX200131
*          DATASET BLOCKING ON ALL THREE DATASETS.                               EX200132
         CLC         DCBBLKSI(2),=H'0'         TEST IF BLKSIZE NOT SPECIFIED    EX200133
         BNER        RE                        RETURN IF SPECIFIED              EX200134
         MVC         DCBBLKSI(2),DCBLRECL      SET BLKSI TO LRECL               EX200135
         BR          RE                        RETURN                           EX200136
         &DROP       R1,RE                                                      EX200137
*                                                                       EX200138
*        THIS ROUTINE IS ENTERED IF AN I/O ERROR OCCURS                 EX200139
*                                                                       EX200140
IOERROR  SYNADAF     ACSMETH=QSAM              DO SYNADAF ANALYSIS              EX200141
         LA          R1,X'1+48                 BUILD A DOPE VECTOR TO DESCRIBE  EX200142
         ST          R1,ERRDOPE                  THE LAST 60 BYTES OF THE       EX200143
         MVC         ERRDOPE+4(4),=H'60,60'    SYNADAF OUTPUT                   EX200144
*        TYPE THE INFO PRECEEDED BY A COMMENT USING WTO SUBROUTINE              EX200145
         &CALL       WTO,('RESEQ - I/O ERROR. SYNADAF INFO=',ERRDOPE)           EX200146
         SYNADRLS                              RELEASE SYNAD CORE               EX200147
         LA          R2,12                     SET RETURN CODE                  EX200148
         B           CLOSEALL                  CLOSE DATASETS AND RETURN        EX200149
*                                                                       EX200150
*        EXTED INSTRUCTIONS                                             EX200151
IDMOVE   MVC         SEQNO(*-*),X89            MOVE DECK ID                     EX200152
*                                                                       EX200153
*        CONSTANTS                                                      EX200154
*                                                                       EX200155
*        DCB'S       (NOTE THAT THESE DCB'S ARE NOT OPENED HERE.  THE   EX200156
*                      OPEN SUBROUTINE DYNAMICALLY GETS CORE FOR A      EX200157
*                      COPY OF EACH, AND IT IS THIS COPY WHICH IS       EX200158
*                      OPENED AND USED)                                 EX200159
*                                                                       EX200160
INPUT    DCB         DDNAME=SYSIN,DSORG=PS,MACRF=GM,LRECL=80,                XEX200161
               RECFM=FB,SYNAD=IOERROR,EXLST=EXLIST,EODAD=INPUTEOF         EX200162
INLEN    EQU         *-INPUT                                               EX200163
*                                                                       EX200164
PRINT    DCB         DDNAME=SYSPRINT,DSORG=PS,MACRF=PM,LRECL=81,             XEX200165
               RECFM=FBA,SYNAD=IOERROR,EXLST=EXLIST                       EX200166
PRLEN    EQU         *-PRINT                                               EX200167
*                                                                       EX200168
PUNCH    DCB         DDNAME=SYSPUNCH,DSORG=PS,MACRF=PM,LRECL=80,             XEX200169
               RECFM=FB,SYNAD=IOERROR,EXLST=EXLIST                        EX200170
PULEN    EQU         *-PUNCH                                               EX200171
*                                                                       EX200172
         LTORG                                                          EX200173
         TITLE       'OPEN     OPEN A DATASET'                          EX200174
*                                                                       EX200175
*        THIS ROUTINE IS USED TO OPEN A DCB.                            EX200176
*                                                                       EX200177
```

```
*        CORE FOR THE DCB IS OBTAINED VIA GETMAIN, AND THE DCB COPIED   EX200178
*          INTO IT.                                                     EX200179
*        THE DCB IS THEN OPENED.                                        EX200180
*        IF THE OPEN FAILS, THEN THE DCB IS FREED AND A MESSAGE TYPED.  EX200181
*                                                                       EX200182
*        INPUT     R1  ADDR OF CONSTANT DCB                             EX200183
*                  R2  LENGTH OF DCB                                    EX200184
*                  R3  0 IF INPUT,  1 IF OUTPUT                         EX200185
*                                                                       EX200186
*        OUTPUT    R1  ADDR OF OPENED DCB IN GOTTEN CORE                EX200187
*                                                                       EX200188
*        RETURN    0  OK                                                EX200189
*                  4  UNABLE TO OPEN                                    EX200190
*                                                                       EX200191
OPEN     &ROUTINE                                                       EX200192
*                                                                       EX200193
*        WORK AREA                                                      EX200194
*                                                                       EX200195
*        AN ME=L OPEN MACRO                                             EX200196
OPENMEL  OPEN      *-*,MF=L            RESERVE CORE FOR ME=L OPEN       EX200197
OPENLEN  EQU       *-OPENMEL           LENGTH OF CORE RESERVED          EX200198
*                                                                       EX200199
NAMEDOPE DS        2F                  RESERVE CORE FOR DOPE VECTOR TO XEX200200
                                         DESCRIBE DDNAME FIELD IN DCB  EX200201
*                                                                       EX200202
*                                                                       EX200203
         &ENTRY    (R0),(R8),(R3)                                       EX200204
*        GET CORE FOR THE DCB AND PUT THE DCB IN IT                     EX200205
         GETMAIN   R,LV=(R8)           GET THE CORE                     EX200206
         LR        R7,R1               SAVE THE ADDR OF GOTTEN CORE     EX200207
         BCTR      R2,0                LENGTH OF DCB -1 FOR MVC         EX200208
         EX        R2,MOVEDCB          EX OF MVC OF DCB                 EX200209
*        INITIALIZE THE ME=L OPEN AND EXECUTE THE ME=E OPEN ON IT       EX200210
         LTR       R3,R3               TEST IF INPUT OR OUTPUT          EX200211
         BZ        ININIT                                               EX200212
         MVC       OPENMEL(OPENLEN),OPENOUT INIT ME=L IN WORK AREA      EX200213
         B         OPENIT                                               EX200214
ININIT   MVC       OPENMEL(OPENLEN),OPENIN  INIT ME=L IN WORK AREA      EX200215
OPENIT   OPEN      ((R7)),MF=(E,OPENMEL) DO THE OPEN                    EX200216
*        CHECK THE OPEN WORKED                                          EX200217
         USING     IHADCB,R7           ADDRESSABILITY VIA IBM DSECT     EX200218
         TM        DCBOFLGS,X'10'      TEST OPEN BIT                    EX200219
         BO        OK                                                   EX200220
*        OPEN FAILED - TYPE A MESSAGE                                   EX200221
         LA        R1,DCBDDNAM         BUILD DOPE VECTOR FOR DDNAME     EX200222
         ST        R1,NAMEDOPE         ADDRESS                          EX200223
         MVC       NAMEDOPE+4(4),=H'8,8' MAX AND CURRENT LENGTH         EX200224
         CALL      WTO,('RESEQ - UNABLE TO OPEN ',NAMEDOPE) TYPE THEM   EX200225
         FREEMAIN  R,LV=(R8),A=(R7)    FREE DCB'S CORE                  EX200226
         LA        RF,4                SET RETURN CODE                  EX200227
         B         GETBACK             GET BACK                         EX200228
         DROP      R7                                                   EX200229
*        RETURN TO CALLER WITH ADDR OF GOTTEN DCB                       EX200230
OK       LA        RF,0                SET RETURN CODE                  EX200231
GETBACK  &RETURN   (RF),((R7))         RETURN, PASSING BACK DCB ADDR    EX200232
*        EXECUTED INSTRUCTIONS                                          EX200233
MOVEDCB  MVC       XR7(*-*),XR0        MOVE THE DCB INTO GOTTEN CORE    EX200234
*        CONSTANTS                                                      EX200235
OPENOUT  OPEN      (*-*,OUTPUT),MF=L   ME=L CONSTANT OPEN FOR OUTPUT    EX200236
OPENIN   OPEN      (*-*,INPUT),MF=L    ME=L CONSTANT OPEN FOR INPUT     EX200237
         LTORG                                                          EX200238
```

```
         LE      'CLOSE          CLOSE A DATASET'                    EX200238
                                                                     EX200239
*        THIS SUBROUTINE IS USED TO CLOSE A DATASET                  EX200240
                                                                     EX200241
*        AFTER THE DCB IS CLOSED THE BUFFER POOL IS FREED AND THEN   EX200242
*          THE CORE OCCUPIED BY THE DCB ITSELF IS FREED.             EX200243
*                                                                    EX200244
*        INPUT     R1  ADDR OF DCB TO BE CLOSED                      EX200245
*                  R2  LENGTH OF THIS DCB                            EX200246
*                                                                    EX200247
*        RETURN    0  OK                                             EX200248
*                                                                    EX200249
CLOSE    IROUTINE                                                    EX200250
*                                                                    EX200251
*        WORK AREA                                                   EX200252
*                                                                    EX200253
*        AN NF=L CLOSE MACRO (TO RESERVE SPACE IN THE WORK AREA DSECT) EX200254
CLOSEMEL CLOSE     *-*,MF=L              MF=L CLOSE CODE             EX200255
CLOSELEN EQU       *-CLOSEMEL            LENGTH OF CORE RESERVED     EX200256
*                                                                    EX200257
         &ENTRY    ((R9),(R8))           MOVE ENTRY REGS TO R9 AND R8 EX200258
*        INIT THE MF=L CLOSE AND EXECUTE IT                          EX200259
         MVC       CLOSEMEL(CLOSELEN),CLOSECON INIT THE MF=L CLOSE   EX200260
         CLOSE     ((R9)),MF=(E,CLOSEMEL) EXECUTE A CLOSE ON THE DCB EX200261
         DO A FREEPOOL ON THE DCB                                    EX200262
         FREEPOOL  (R9)                  FREE THE BUFFER POOL        EX200263
*        FREE THE DCB'S CORE                                         EX200264
         FREEMAIN  R,LV=(R8),A=(R9)      FREEMAIN THE DCB            EX200265
*        RETURN TO CALLER                                            EX200266
         &RETURN   0                     RETURN WITH 0 RET CODE      EX200267
*        CONSTANTS                                                   EX200268
CLOSECON CLOSE     *-*,MF=L              CONSTANT MF=L CLOSE MACRO   EX200269
         LTORG                                                       EX200270
         TITLE     'WTO          WRITE TO OPERATOR'                  EX200271
*                                                                    EX200272
*        THIS SUBROUTINE IS USED TO TYPE MESSAGES TO THE OPERATOR    EX200273
*                                                                    EX200274
*        IT ACCEPTS TWO DOPE-VECTORED STRINGS AS INPUT.              EX200275
*        THESE TWO STRINGS ARE CONCATENATED TOGETHER AND TYPED OUT   EX200276
*          ON ONE LINE USEING THE WTO MACRO.                         EX200277
*        EITHER OF THE STRINGS MAY BE NULL IF DESIRED (BUT NOT BOTH) EX200278
*                                                                    EX200279
*        INPUT     R1  ADDR OF DOPE VECTOR OF 1ST STRING             EX200280
*                  R2  ADDR OF DOPE VECTOR OF 2ND STRING             EX200281
*                                                                    EX200282
*        RETURN    0  OK                                             EX200283
*                                                                    EX200284
*        LIMITATIONS - COMBINED LENGTH OF THE 2 STRINGS MUST BE 122 OR EX200285
*                      LESS. THERE IS NO CHECKING OF LENGTHS BY THIS EX200286
*                      ROUTINE.                                      EX200287
*                                                                    EX200288
WTO      &ROUTINE                                                    EX200289
*                                                                    EX200290
*        WORK AREA                                                   EX200291
*                                                                    EX200292
*        THE FOLLOWING TWO STATEMENTS ARE A HAND-GENERATED MF=L WTO  EX200293
WTOMEL   DS        0F'0',AL2(L'WTOMES+4),H'0' LENGTH+4 AND ZEROS     EX200294
WTOMES   DS        CL122                 MESSAGE  (MAX LENGTH OF 122) EX200295
*                                                                    EX200296
         &ENTRY    ((R1),(R9))           ADDR OF 2ND DOPE VECTOR TO R9 EX200297
```

```
         LA        R4,WTOMES          R4 POINTS AT MESSAGE AREA          EX200298
         XLDV      (R2,,R5),XR1       GET ADDR AND LENGTH OF 1ST STRIN   EX200299
         LTR       R1,R5              LEN TO R1 AND TEST IF NULL         EX200300
         BNH       STRING2            IGNORE STRING IF NULL              EX200301
         BCTR      R1,0               LEN-1 FOR MVC                      EX200302
         EX        R1,MVCMES          MOVE 1ST STRING TO MESSAGE AREA    EX200303
         AR        R4,R5              BUMP MESSAGE AREA POINTER          EX200304
STRING2  XLDV      (R2,,R6),XR9       GET ADDR AND LEN OF 2ND STRING     EX200305
         LTR       R1,R6              LENGTH TO R1 AND TEST IF NULL      EX200306
         BNH       DOIT               BR IF NULL                         EX200307
         BCTR      R1,0               LENGTH-1 FOR MVC                   EX200308
         EX        R1,MVCMES          ADD 2ND STRING ONTO MESSAGE AREA   EX200309
DOIT     LA        R0,4(R5,R6)        WTO LENGTH = LEN1 + LEN2 + 4       EX200310
         SLL       R0,16              MOVE TO TOP HALF OF REGISTER       EX200311
         ST        R0,WTOMEL          STORE WTO LEN AND ZEROS IN MF=L    EX200312
         WTO       MF=(E,WTOMEL)      DO THE WTO                         EX200313
         XRETURN   0                  RETURN TO CALLER                   EX200314
*                                                                         EX200315
*        EXECUTED INSTRUCTIONS                                            EX200316
MVCMES   MVC       XR4(*-*),XR2       MOVE A STRING TO MESSAGE AREA      EX200317
*        CONSTANTS                                                        EX200318
         LTORG                                                           EX200319
         TITLE     'REGISTER DEFINITIONS'                                EX200320
         XSETD                                                           EX200321
         XSETX                                                           EX200322
         TITLE     'DCB DSECT USING LRN DCBD MACRO'                      EX200323
         DCBD      DSORG=PS                                              EX200324
         END       RESEQ
/*
//GO.SYSPRINT DD SYSOUT=A
//GO.SYSPUNCH DD SYSOUT=B
//GO.SYSIN DD   *
         PLACE CARDS TO BE RESEQUENCED HERE
/*
```

SIDSTE NYT OM RENNIE'S MACROER

De er alle indlagt i SYS2.MACLIB

New Macros

These notes document some macros which have been written or altered since the release of my manual "Introduction to Assembly Language Programming With OS/360".

Rennie Petersen

Oct 70
Nov 70

NEUCC

The #DC and #DS macros are used to declare strings and/or dope vectors. The basic form of the macro is

[label] {#DC / #DS} opnd1, opnd2, ...

Each operand can take one of the following forms

'string' this is the implied string constant. A dope vector and a character string are generated. The string may be null if desired.

#[L modifier]'string' this is the explicit string constant. The only difference is that it permits a length modifier to be specified. A dope vector is generated with an address of *+8, a max length as specified in the modifier or, if no modifier, the strings length, and the current length is the strings length. The string may be null, or, in the case of #DS, may be omitted.

@(addr, maxlen, curlen) a dope vector is generated with the specified address, maxlength, and current length fields. For #DS the specification may be omitted.

other any other type of operand is assumed to be an ordinary DC, and is generated as such

examples of &DC or &DS

a) implied null string

```
[label]   &DC or &DS      ''

+[label]  DC or DS        A(0), AL2(0,0)
```

b) implied string

```
[label]   &DC or &DS      'string'

+[label]  DC or DS        A(*+8), AL2(length, length), C'string'
```

c) null string

```
[label]   &DC or &DS      &''

+[label]  DC or DS        A(0), AL2(0,0)
```

d) string

```
[label]   &DC or &DS      &'string'

+[label]  DC or DS        A(*+8), AL2(length, length), C'string
```

e) null string with length modifier

```
[label]   &DC or &DS      &Lmodifier''

+[label]  &DC or &DS      A(*+8), AL2(modifier, 0)

+         ORG             *+length
```

f) string with length modifier

```
[label]   &DC or &DS      &Lmodifier'string'

+[label]  &DC or &DS      A(*+8), AL2(modifier, length), C'string'

          ORG             *-length+modifier
```

g) dope vector

```
[label]   &DC or &DS      @(addr, maxlen, curlen)

+[label]  DC or DS        A(addr), AL2(maxlen, curlen)
```

new $ROUTINE,
$ENTRY
and $RETURN

new Routine Macros

One problem with the original version of my Routine macros was that one had to pay the penalty of a GETMAIN and FREEMAIN being done everytime a routine was called, even if it was not necessary. This could consume a large amount of CPU time. There are two cases where the GETMAIN/FREEMAINs are not necessary.

1. If the routine is not re-entrant or recursive.
2. If the routine is a lowest-level routine (does not call any other routines or OS subroutines) and does not need a work area (i.e. it does all its manipulations in the registers or in areas supplied by the caller).

To overcome this problem, several changes have been made to the $ROUTINE, $ENTRY, and $RETURN macros. The changes to $ENTRY and $RETURN are completely internal, i.e. there is no change to the way they are used. The $ROUTINE macro is changed to permit the user to specify, if the routine is Reentrant or Recursive, and whether or not a new save area is desired. Note that if the routine is re-entrant, this must be specified, which is different from the old version of the macros, which always assumed this.

To start with, if the user is coding a lowest-level routine which does not need a save or work area, then on the $ROUTINE statement, he codes NEWSAVE=NO. When this is done, the $ROUTINE macro does not generate a DSECT or a save area, $ENTRY saves the callers registers, but does not acquire a new save area, and $RETURN simply

restores the caller's registers, without freeing any save area.

In the other case, if the user does not specify OPTIONS=RENT or OPTIONS=(MAIN,RENT), (and if he has not specified NEWSAVE=NO), then the following will happen. &ROUTINE will generate the save/work area as a CSECT instead of a DSECT (it will still have the same arbitrary name), &ENTRY will set RD to point at the save/work area by loading a V-type address constant which references it, and will then chain the save areas in the normal fashion, and finally, &RETURN will point RD back to the caller's save area and restore his registers without freeing any save area. This permits the program to be written in a re-entrant fashion if desired for easy later conversion, but, until OPTIONS=RENT is specified, it eliminates the GETMAIN/FREEMA overhead.

One final feature of the new Routine macros is that, in the case of non-reentrant programming, they permit the user to specify the address of the save area. In this case, &ROUTINE does not generate any save/work area, but &ENTRY will set RD to point at the user's save area by doing a LA on it, and &RETURN will again point RD back to the caller's save area and restore his registers. In order to specify the address of a save area, the user should code NEWSAVE=(YES,saveareaname) on his &ROUTINE statement. Note one restriction: if OPTIONS=MAIN is also specified, then the save area supplied by the user must be 20 words long, rather than just 18.

$END

This macro is used to end a routine which was begun with a $ROUTINE macro. Its format is as follows

```
$END    routinename
```

The routine name is required, and must match the name on the previous $ROUTINE statement. This macro generates a $LTORG to generate all literals and executed instructions (see $LTORG and $EX), and then generates a $DROP to drop all the registers.

example

```
$END    MAIN
```

$EX

This macro permits the executed instruction to be specified as the 2nd operand of the $EX instruction. Its form is as follows

[Label] $EX register, (opcode, operand1, operand2)[, USE=(reg, addr

where opcode, operand1, and operand2 fully define the instruction to be executed. The executed instruction will be generated at the location of the next $END instruction (or $LTORG instruction). See $END and $LTORG for more details. If the USE= operand is coded, then a $USE reg, addr is generated before the executed instruction, and a $DROP reg is generated following it.

```
ex.   PROPAGAT   $EX     R5, (MVC, XR1+1(*-*), XR1)
                 $EX     R7, (TR, XR1(*-*), WTCTTAB1), USE=(R9,
                                                    WTCT)
                  ⋮
                 $END    ROUTINE
```

these instructions will generate the following

```
PROPAGAT   EX     R5, $LIT0925
           EX     R7, $LIT0934
            ⋮
$LIT0925   MVC    XR1+1(*-*), XR1
           $USE   R9, WTCT
$LIT0934   TR     XR1(*-*), WTCTTAB1
           $DROP  R9
```

#LTORG

This macro generates any literals that have been saved by other macros (#CLS, #MVS, etc), and also generates any instructions saved by the #EX macro. It then generates an ordinary assembler LTORG. The format of the instruction is

#LTORG

The instruction has no operands, and can not have a label.

example

#LTORG

String Manipulation Macros

The following macros are used to manipulate strings.

$CLS - Compare Logical Strings

$CLS

format:

[label] $CLS string1, string2

The two strings are compared (if they are of unequal length, only their lengths are compared) and the condition code is set. Registers R1, R2, and RF are clobbered. Either operand may be a string literal if desired.

examples:

```
COMPARE  $CLS   BUFFER,='/*'
         $CLS   XR4,DOPE1
```

$MVS - Move String

$MVS

format:

[label] $MVS string1, string2

The second string is moved into the first strings location (and truncated if necessary) and the length of the first string is set. Registers R1, R2, and RF are clobbered. The second operand may be a string literal if desired. Note that the operand string1 must refer to a valid dope vector, whose address and max length is used by this instruction.

examples:

```
MOVE     $MVS   BUFFER,='HI THERE'
         $MVS   DOPE1, XR4
```

#CVSC - Convert to String from Character — $CVSC

format:

[label] #CVSC string, opnd [,LEN=length]

The characters at location opnd are moved into the string (and truncated if necessary), and the current length of the string set. If LEN is not specified, then the length attribute of opnd is used. Registers R1 and RF are clobbered. Opnd may be a character literal if desired.

examples:

```
MSG   #CVSC   BUFFER,=C'HI THERE'
      #CVSC   DOPE1,XR5,LEN=20
```

#CVCS - Convert to Character from String — $CVCS

format:

[label] #CVCS opnd, string [LEN=length]

The contents of the string are moved to the location opnd (and truncated or padded with blanks if necessary). If LEN is not specified, then the length attribute of opnd is used to determine the length of the resultant data. Registers R1, R2, and RF are clobbered. String may be a string literal if desired.

examples:

```
PAD   #CVCS   XR5,='/*',LEN=80
      #CVCS   LINE,BUFFER,LEN=(R7)
```

$CVBS - Convert to Binary from String — $CVBS

format:

[label] $CVBS reg, string

The string (which should consist of numeric data and optionally preceeding blanks) is converted to binary in general register reg. Registers R1, R2, and RF are clobbered. The condition code is set to equal or high, depending on whether the result is zero or not. The string operand may be a string literal if desired.

examples:

```
        $CVBS   R0, BUFFER
        $CVBS   R3, ='b'-10'5'
```

$MVDV - Move Dope Vector — $MVDV

format:

[label] $MVDV opnd1, opnd2

This macro simply generates a move of 8 bytes. The second operand may be a dopevector literal if desired.

examples:

```
MOVEDV  $MVDV   DOPE, =@(LINE, 80, 0)
        $MVDV   BUFFER, XR4
```

Many More Miscellaneous Macros

. Rennie 1 Feb 71

Register Macros

#USING - this macro maintains the internal tables in a manar compatible to the #USE and #DROP macros. However, unlike #USE, it conforms to the syntax of the assembler USING statement.

example -

```
#USING  *, R5, R6
```

this statement would generate the following instructions

```
USING  *+0, R5
USING  *+4096, R6
```

Routine Macros

Support of error conditions.

When a routine returns to its caller, its
code is usually in one of the following catag
normal - return code Ø
unusual condition found - return codes 4 or
example end-of-file encountered
error condition - return codes 12 and high
example permanent I/O error.

Often, in a highly structured program with nested routines, one finds that it is annoyi to check for error conditions on return from call. Usually, when an error condition is en ou want to "filter back up to the top" of the subroutine calls, and there at the top level, o of the highest levels, take the appropriate actions neccessiated by the error.

This sort of scheme is implimented as First of all, if the global symbol &&MON is a non-null value, then the code produced by and &RETURN is modified. And secondly, a new macro, &ERROR, has been added to use in signalling the detection of an error condition.

&RETURN - when &&MON is non-null, the &RETURN ro generates a branch to four bytes past RE stea of a direct branch through RE

$CALL - when &MON is non-null, this macro generates an extra instruction following the BALR to the subroutine. This extra instruction will be skipped if the subroutine does a normal $RETURN. $RETURN will branch to 4 bytes past RE. if instead a $ERROR is done by the subroutine then this extra instruction will be executed as the subject of an EX instruction (see $ERROR below). The $CALL macro has an ERROR= operand to allow the setting of this extra instruction. ERROR= operand may have the following:

operand	instruction generated
ABEND	DC ZH'0'
IGNORE	B *+4
(BUMP, n)	LA RF, XRF+n
(BRANCH, a)	B a
(JUMP, a)	B a(RF)

example

$CALL READFILE, (PARM, (R5)), ERROR=(BUMP,...

would generate

```
L     RF,=V(READFILE)
LA    R1,PARM
LR    R2,R5
BALR  RE,RF
LA    RF,XRF+8      - only executed if subr does a $ERROR
```

$ERROR - this macro is somewhat similar to the $RETURN macro, in that it sets the return code, frees the work area if any, and restores the callers registers. However, instead of branching to the contents of RE+4, as $RETURN does, $ERROR generates an EX of the instruction at RE. The instruction at this location depends on the ERROR= option coded on the $CALL macro. If ERROR=ABEND was coded, a $C1 abend will occur. If ERROR=IGNORE, (BRANCH,a), or (JUMP,a) was coded, then a branch takes place and the calling program is now in control again. However, if ERROR=(BUMP,n) was coded, then the instruction which modifies RF is executed, control returns to the remaining instructions generated by the $ERROR macro. These instructions effectively result in looping through the $ERROR macro again (except for the return code setting). Then, as if the calling program had issued a $ERROR and its work area (if any) is freed and the registers of its calling program are restored, and an EX done on the instruction at RE again. So it continues until a $CALL is found that specifies something other than ERROR=(BUMP,n).

general form of $ERROR

[label] $ERROR returncode

for example, consider the following structure

```
MAIN    &ROUTINE    OPTIONS=MAIN

        &CALL       A, ERROR=(BRANCH, NUTS)
NUTS    LR          R1,RF
        ABEND       (1)
```

```
A       &ROUTINE
        &CALL       B, ERROR=(BUMP, 8)
```

```
B       &ROUTINE
        &CALL       C, ERROR=(BUMP, 8)

        &ERROR      8
```

```
C       &ROUTINE


        &ERROR      8
```

In this example, MAIN calls A, which calls B, which calls C. If C executes the &ERROR 8 statement then C returns with an error code of 8, B returns with an error code of 16 (8+8), and A returns with an error code of 24 (8+16), and a branch to location NUTS is executed. If instead, the &ERROR statement in B was executed, RF would contain 16 the error had filtered up to MAIN.

Routine Macros

Data Block Macros

These macros are used instead of #ROUTINE and #ENTRY for blocks of data which contain no executable code, and thus no need for saving and restoring registers, etc. An example of a possible use might be a CSECT full of translate tables or lookup tables.

#BLOCK - this macro simply sets up internal variables for the following #DATA macro(s). It takes the place of #ROUTINE when no executable code is involved.

syntax

label #BLOCK

#DATA - this macro is used to generate CSECT or ENTRY statements. The first #DATA in a block generates a CSECT statement of the name of the block. If it has a label, and that label is different from the name of the block, an entryname EQU blockname and a ENTRY entryname are generated. For all subsequent #DATA statements, an entryname DC 0D'0' and an ENTRY entryname are generated.

syntax

entryname #DATA

String Macros

#CLSC - compare logical string to characters

This macro can be used to compare a string to some characters in memory.

syntax

[label] #CLSC string, opnd[,LEN=length]

The code generated by the macro first compares length of the string to the length attribute of opnd, if LEN= is specified, to the length specified by it. If these lengths are equal, then a CLC is done the contents of the string and opnd. In either case the condition code is set, reflecting either the comparison of the lengths or the comparison of the data

The first operand may be a string literal
The second operand may be any other kind of [illegible] if desired.

examples:

```
        #CLSC   BUFFER,=C'/*'
        #CLSC   XR4,LINE1
TEST    #CLSC   =C'$',BYTE1
```

Assembler op extension macros

These macros simply provide minor extensions beyond using the same opcode without the #.

#CSECT - as well as generating a CSECT statement, this macro sets the global variable &#SECNAM to the name of the CSECT, and sets the global variable &#SECTYP to 'CSECT'.

#DSECT - as well as generating a DSECT statement this macro sets the global variable &#SECNAM to the name of the DSECT, and sets the global variable &#SECTYP to 'DSECT'. Also, if the label field is blank, an arbitrary name is made up for it

#EQU - if the label field is non-blank, an EQU is generated. If the label field is blank, nothing is generated.

#PRINT - as well as generating a PRINT statement, this macro saves the ON/OFF, GEN/NOGEN, and DATA/NODATA values in the global variables &#PRON, &#PRGEN, &#PRDATA.

Control Block Generation Macros

The following macros are used to aid in the writting of macros to generate control blocks. Using these macros provides the following services:

a) allows the same macro to be used to generate either a DSECT or ordinary copy of the control block.

b) permits the optional generation of no internal labels, some internal labels, or all internal labels on the fields within the control block.

The following is an example of a control block macro written using these services.

```
         MACRO
&L       &DDB    &ADDR=0,&LNTH=0,                              X
                   &DSECT=NO,&GEN=
.*       DATA DESTROYER BLOCK
&L       &E      DDB,&GEN,&DSECT,F
DDBADDR  &DC     A(&ADDR)      ADDR OF DATA TO BE DESTROYED
DDBLNTH  &DC     H'&LNTH'      LENGTH OF DATA TO BE DESTROYED
         &X
         MEND
```

This macro can now be used either to generate a DDB block in your program

for example

```
DESTBLOK  &DDB   ADDR=DATA5,LNTH=10
```

would generate

```
DESTBLOK  DC     0F'0'
          DC     A(DATA5)
          DC     H'10'
```

Or else you can use this macro to generate a DDB DSECT, for purposes of symbolic addressing of the fields within the block. In this case, only those labels which you request be generated will be generated.

for example

```
        #DDB    DSECT=YES, GEN=(DDBLNTH)
```

would generate the following

```
DDB     DSECT
        DC      A(0)
DDBLNTH DC      H'0'
```

i.e. the name of the block, plus all labels included in the GEN= operand are generated, but none of the other labels. (If you wish to generate all the labels, you can code GEN=9).

This scheme is implimented via the #E and #X macros, plus the fact that #DC tests internal tables set up by #E before generating the label. Description of the #E and #X macros follow.

#E - (for Entry) - this macro is used to set up internal tables and switches for #DC and #X to support optional generation of labels. It also generates either a DSECT or a DC statement to begin the block.

syntax -

```
&label  #E  blockname, &GEN, &DSECT, blockalignment
```

explanation -

&label - this should be the same variable symbol as used on the control block's macro prote... It is used in determining the label to pla... on the DSECT or DC statement which generated by &E to begin the block.

blockname - this is the name of the block. ... generated as the label on the DSECT or ... produced by &E under the following condit...
a) &label was blank
b) &GEN was non-blank.

&GEN - this should just be the &GEN= parameter from the control block macro proto... It should have been given a null defau... value. It is used to set internal table.

&DSECT - this should just be the &DSECT= parame... from the control block macro prototype. It should have been given a default value of YES or NO. It is used to d...ine whether to generate a DSECT or a ... to begin the control block.

blockalignment - this specifies the alignment nece...ary for the block, usually F or D. It is used in the DC to begin the block if DSE...=NO was specified or assumed. The resulting DC statement will be DC 0x'0', where x is F or D, as specified by this parameter.

#X - this macro simply cleans up internal tables to restore #DC to its normal mode of always generating the label. It should always be the last statement in a control block generating macro.

syntax

#X

this macro has no label and no operands.

Another macro involved in this scheme is the #G macro. This macro is used to determine if a label is to be generated or not. It is used by #DC, #CCW and #DS, among others of the data generating macros. If it is desired to use it explicitly, the following documentation specifies how.

#G - this macro determines if a label should be generated or not.

syntax -

&label #G

where label is the name to be determined if it should be generated or not.

The answer is returned in the global symbol &#L. If the label should be generated, then it will be returned in &#L. If the label should not be generated, &#L will be set to null. The following example illustrates the use of this macro

```
DDBADDR  #G                    TEST IF DDBADDR TO BE GEN
&#L      DC    A(&ADDR)        ADDR OF DATA TO BE DESTROYE
```

Internal Macros

The following macros are used internally by the other macros. Their exact syntax and operation need not concern the average user.

General

#P - this macro is used to load or store a parameter into or from a register. It is used by many of the other macros.

#SAVELIT - this macro is used to save a literal away for generation at the next #LTORG. It is used by #EX, #P, and all the string manipulation macros.

String Manipulation Macros

#S1 and #S2 - are used to process the first second operands respectively

Switch Macros

#SWSCAN - is used to scan off the bit operands for #IF, #ON, #OFF, etc

Register Macros

#RNUM and #RSET are internal macros of #USE and #DR

Routine Macros

#GETCORE and #PUTCORE are internal macros of #ENTRY, #RETURN, and #ERROR

#DC and #DS

#S@L, #S@D, #S@P, and #S@S are internal macros used to scan off various types of operands. #S@P is also used by #SAVELIT.

#E and #G

#GENSAVE and #GENSYM are internal macros used to manipulate the table of labels to be generated, and to determine if a label is to be generated.

Miscellaneous

#DCDS - internal macro of #DC and #DS.

#CCW1 - internal macro of #CCW

#SI - i cann't figure out what this is for.

. Rennie

2 Feb 71

[illegible]

This option should only be specified for the main routine, not for subroutines. Specifying this option has the following effect:

[illegible] Each ENTRY within the routine which [illegible] has operands will contain extra code to build a dope vector describing the PARM field on the EXEC card. This dope vector occupies two words which are automatically allocated immediatly following the save area in the save/work area. The address of this dope vector is then placed in R1. The user must be careful of the following: i) This service is not provided if NEWSAVE=NO is specified. For [illegible] the building of the dope vector to work, the routine must be called with an OS-initiator-compatible parameter list which describes the PARM field. See the OS Supervisor Services manual for more information. If this requirement is not met a program check may occur. Therefore, the user should not specify OPTIONS=MAIN or should not code operands on the ENTRYs unless this requirement is met. [illegible] This requirement is met by the TRACE [illegible] and the LOADER, as well as by the OS initiator. ii) Since the dope vector requires two words following the save area, [illegible] the user [illegible] must provide [illegible] words [illegible] [illegible] be [illegible] coded! NEWSAVE=(YES,[illegible]

a routine with OPTIONS=MAIN and operands on any SAVE statement. These two words are automatically checked for other forms of the recursive operand.

b) If OPTIONS=REENT or RECUR is also specified and SAVE=(YES,number) is specified (implying save area stacking), then the test to see if this save/work area will fit in the previous stack is omitted. I.e., a routine with OPTIONS=(MAIN,REENT or RECUR) with SAVE=(YES,number) will always obtain a new stack when it is entered.

c) If OPTIONS=ERRET is also specified, then any SAVE/RETURN statements in this routine will branch to the address in RE, rather than to R14. Also, USERERR is not permitted in a routine with OPTIONS=(MAIN,ERRET). The OPTIONS=ERRET still causes all RECALL statements to include the extra error handling instructions.

OPTIONS = REENT or RECUR

This option specifies that the routine will be used in a reentrant or recursive manner, implying that the save/work area, if any, must be obtained in a dynamic manner, i.e. via GETMAIN or from a stack of save/work areas. This option works in conjunction with the RENTCODE= options, and its effect is described more completely in a chart in the documentation of LIBRARY.

OPTIONS = BASET

This option specifies that this routine is one of a family of routines making use of the ERROR facility. The ERROR facility is described more completely in the ERROR macro previously in the section 'Automatic handling of error conditions'.

NEWSAVE= - This operand specifies the type of save/work area to be aquired for this routine. This operand can take four major forms:

NEWSAVE=YES (default) - This specifies that a save/work area is to be aquired for this routine.

NEWSAVE=(YES, number) - This specifies that save area stacking is desired. When necessary a save/work area stack will be obtained, with 'number' specifying the size of the stack.

NEWSAVE=(YES, symbol) - This specifies that the user wishes to use a save/work area which is in his program at the location 'symbol'. No work area may be declared following the FRONTLINE.

NEWSAVE=NO - This specifies that no new save area is required at all. No work area may be declared following the FRONTLINE, and the routine must not call any other routines nor use any OS macros which require a save area.

The NEWSAVE= operand interacts with the presence or absence of OPTIONS=RENT or REUSE in specifying exactly how the save/work area is acquired. The following chart included in the documentation of HERCKY indicates the possible combinations and the resultant method used. If the NEWSAVE= operand is omitted, then NEWSAVE=(YES, H=0) is assumed.

In addition, when the NEWSAVE= operand is used with HIARCHY support, the following forms are also permitted:

NEWSAVE=.(YES, H=0 or 1) or
NEWSAVE=(YES, number, H=0 or 1)

- These forms can be used to specify which hierarchy the GETMAIN is to be done in if a GETMAIN is necessary.

DCL=

This operand is simply used to get &ROUTINE to declare the symbolic register names, instead of having to use a &DCL macro. The usual procedure is to specify DCL=(R,XR) on the first &ROUTINE, and to not specify it at all on the other &ROUTINE statements in the program. See the &DCL macro for more information on the symbolic register names.

C. Routine Macros

Macros used to facilitate modular programming under OS, with special support for reentrant programming, save area stacking, and automatic handling of error conditions.

C.1 Modular programming under OS

When using these macros to write modular programs to run under OS, one proceeds as follows: ~~First,~~ the entire program is divided up into ~~several~~ one or more independent routines, which call each other in a structured manner. Each routine is written to the following conventions:

```
         TITLE    'routname   purpose of routine'
routname BROUTINE
         comments
         ~~BSAVE~~ work area (optional)
         BENTRY
         executable instructions,
           including BCALLs

         BRETURN
         ~~BERROR~~ ~~(optional)~~
*        CONSTANTS
         constants, if any
         BEND     routname
```

The above sequence is repeated for each routine required to make up the complete program. Finally, the last routine is followed by an assembler END statement.

Each part of the above sequence of instructions will now be considered in more detail:

TITLE statement - This causes the assembler to skip to a new page and to print the specified title at the top of this page and each succeeding page until a new TITLE statement is encountered.

$ROUTINE statement - This statement has several purposes.

a) It is used to declare the name of the whole routine and to specify the optional facilities desired. For example, if the first routine is invoked directly from OS, then OPTIONS=MAIN should be specified. If the routine is to be reentrant, then OPTIONS=REENT should be specified.

b) It can be used to optionally declare the symbolic register names, via the DCL= operand. This is usually done on the first $ROUTINE in a program.

c) Unless NEWSAVE=NO or NEWSAVE=(YES,symbol) is specified on the &ROUTINE, a CSECT or DSECT will be generated to "open" the definition of the routines save/work area. An 18 word save area will be placed in the save/work area, optionally followed by a 2 word dope vector to describe the PARM field on the EXEC card (only if OPTIONS=MAIN is specified). ~~Also, if save area stacking has been requested (via a NEWSAVE=(YES,number) operand), then the save/work area will be extended~~

comments - It is standard practice to place comments at the start of a routine to describe the purpose of the routine, its method of operation, its limitations, etc. It is also useful to specify the input parameters output values if any, and return codes if any.

work area - This section is used for the DC's, DS's, etc to describe the routines work area. The work area has special significance in reentrant routines, which will be discussed later. No work area exists if NEWSAVE=NO or NEWSAVE=(YES,symbol) was specified on the &ROUTINE.

BENTRY statement - This statement defines the entry point of the routine when called at execution time. As such, the first BENTRY statement in a routine may be regarded as 'closing' the definition of the save/work area (if any) and 'opening' the definition of the executable part of the routine, that is the routine CSECT itself. In addition, each BENTRY statement in a routine performs the following major services:

a) Saves the callers registers in his save area.

b) Sets RC as the base register for the routine.

c) Aquires a new save/work area, if necessary, and chaining it to the caller's save area according to OS conventions. This gives the routine a valid OS save area and makes the users work area addressable.

d) If requested via operands on the BENTRY, code will be generated to place the input parameter registers in core or in other registers.

executable instructions - This is the main, executable part of the routine. This is the section where you actually do the computing required by the calling routine. //Input to the routine consists of the parameters passed in the registers R1, R2, ... For assembly-language-only applications, it is usual to use several registers as necessary to contain input values, addresses of values, or addresses of control blocks. For routines called by OS or Fortran, Cobol, etc the standard calling sequence uses R1 as a pointer to a list of addresses of the input parameters.

To call another routine you can use the $CALL macro.

$CALLs - The $CALL macro is used to call another subroutine from this subroutine. It generates code to perform the following actions:

a) it loads the address of the subroutine into RF.

b) it optionally loads the parameters or the addresses of the parameters into R1 to Rn.

c) it does a BALR RE,RF to the subroutine.

When calling a subroutine it is not necessary to specifically save the general registers since this service should be done by the subroutine. However, if this routine or any of its ascendants use the floating point registers, and if the called subroutine may destroy them, then it is the calling routine's responsibility to save them before the call and to restore them afterwards. This consideration usually only applies to Fortran subroutines.

After the called subroutine has returned it is usually appropriate to test the return code in register RF to determine if any unusual or error conditions have been detected by the subroutine.

$RETURN - This macro is used to return to the calling routine, or in the case of the main routine to return to whoever invoked it, usually OS. The $RETURN macro generates code to perform the following functions:

a) the specified return code is placed in RF. Return codes are normally a multiple of 4 to permit their possible use in a jump table. Zero is used to signify normal completion and 4, 8, etc. to signify unusual or error conditions in increasing order of severity.

b) if return values are specified they are loaded into R1 to Rn.

c) if a save/work area was aquired for this routine by BENTRY it is de-allocated by BRETURN.

d) the callers registers are restored except for RF and the return value registers if any.

e) a LTR RF,RF is done to set the condition code to reflect the zero or non-zero status of the return code.

f) a BR RE is performed to return to the caller.

constants - any constants used in the routine can be defined here.

%END - this statement declares the end of the routine, begun by the %ROUTINE statement. It also causes the generation of any literals or %EX subject instructions to take place.

This completes the detailed over-view of the modular programming technique as implimented through the %ROUTINE, %ENTRY, %CALL, %RETURN, and %END macros. Note that while these macros are especially useful when writing a large modular program, they can still be used to advantage even if writing a small non-modular program or subroutine, since they provide equal or better services than the IBM SAVE, CALL, and RETURN macros.

C.2 Reentrant programming

When this set of macros is used to aid in writing reentrant routines, the following considerations apply as well as those in the preceeding section.

BROUTINE statement - The operand OPTIONS=REENT or RECUR should be specified to cause BENTRY to aquire the save/work area from dynamic core.

work area - This work area must contain all data areas which will be altered or modified by the routine. This area will contain such things as DS's to allocate buffers, pointers, temporary data, etc, as well as system macros which expand into control blocks (e.g. DCB) or parameter lists (e.g. MF=L OPEN). These statements are used to reserve the necessary space in the work area and to define the relative displacement of each label from the start of the work area.

Remember that this work area is declared in the BROUTINE as a DSECT. This means that none

of the data declared here is ever loaded into ~~storage~~ core. The BENTRY sequence will aquire the necessary core for the work area, but ~~it~~ it will not be initialized in any way - it will just contain garbage. Thus it is useless to code DC's in this area, and any of the DS's or system macros (DCB's, MF=L OPENs, etc) that need to be initialized must be explicitely initialized by your routine each time it is entered.

BENTRY - The only difference in BENTRY for a RENT or RECUR program is that the save/work area (if any) is aquired from dynamic core, either via GETMAIN or from a save area stack. Otherwise it functions identically with the ordinary BENTRY.

executable instructions - The only thing to note here is to emphasize once again that ~~it~~ a reentrant program must not modify itself in any way. Thus, the executable part of the routine must not store into itself nor modify the constants part at any time. To use an SS instruction with a computed length the

EX instruction must be used - see the DEX macro for a convenient method.

All ~~work areas~~ items of ~~core~~ storage which will be modified must be allocated in the work area. If necessary, these ~~areas~~ items must be initialized by moving constants from the constant section of the routine to the modifiable ~~area~~ items in the work area. This can be done with a series of MVC instructions at the start of the routine.

in dynamic core. The easiest way to do this is to let ~~the~~ DENTRY aquire the items automatically be declaring them

BRETURN - ~~Like DENTRY,~~ The only difference between the RENT or RECUR BRETURN and the ordinary one is that the reentrant BRETURN frees the save/work area (if any) from the dynamic core that DENTRY obtained it from.

constants - Once again the warning: in a reentrant program the constant area must remain constant. No modification of any kind is permitted to the items that appear in this section.

This completes the discussion of how reentrant programs are implemented with this set of routine macros.

C.3 Save Area Stacking

When writing a large reentrant program using these macros, several problems become apparent. The first is that a GETMAIN and FREEMAIN for every subroutine callo results in a very high system overhead. Secondly, the reentrant option causes ~~versions of~~ ENTRY and RETURN to expand into a not inconsiderable amount of code, so that your program is larger than expected, especially if you have a lot of small reentrant subroutines. Finally, in a reentrant multi-tasking program, the continuous aquiring and freeing of save/work areas can result in storage fragmentation of subpool zero.

All of these problems are relieved to a greater or lesser extent by using the save area stacking option. The principal of save area stacking is as follows. The ~~top~~ main routine aquires a large amount of core, refered to as a stack. This main routine then sub-allocates its own save/work area from the beginning of this stack and sets pointers indicating that the remainder of the stack is free. When this routine calls a subroutine, the subroutine tests if the free part of the stack is large enough for

its save/work area. If so, this save/work area is also sub-allocated in the stack. If not, the subroutine will aquire a new stack. In either case, the stack pointers are updated to indicate the amount of free space remaining in the last stack. This space is available for use by subroutines called by this subroutine. for save area stacking to be effective, it is obvious that ~~the top routine~~ stacks should be ~~obtained~~ aquired that are large enough ~~for~~ to satisfy the majority of requests of all the subroutines called. The ideal situation (from the execution time viewpoint) is if the ~~top~~ main routine aquires a large enough stack to satisfy all possible save/work requests made by the subroutines. On the other hand, if a recursive subroutine is called, it may be undesirable to aquire a stack large enough to satisfy the demands that will be made by the maximum possible amount of recursions if this event only occurs rarely. In this case it might be better to have the ~~main~~ ~~top~~ main routine aquire a stack large enough to satisfy the usual amount of recursion by the subroutine, and then let the subroutine aquire additional small stacks when the first large one runs out.

To request the save area stacking facility the user specifies NEWSAVE=(YES, number) on his IROUTINE statement, where 'number' is the size of the stack which will be obtained if it is necessary for this routine to aquire a new stack when it is entered. If it is not necessary for this routine to aquire a new stack due to sufficient space in the previous one, then 'number' will be ignored (however, it must be specified in order to invoke the save area stacking option).

Although save area stacking only works in conjunction with OPTIONS=RENT or RECUR, the NEWSAVE=(YES, number) operand may be specified on a IROUTINE that does not specify OPTIONS=RENT or RECUR. In this case a static stack is generated at assembly time and loaded as part of the program. This could be useful in the case of a non-reentra main routine that calls reentrant and/or recursive subroutines. See the chart following the documentatio of IENTRY for an overview of the different methods of aquiring stacks and save/work areas.

// Also, there is one exception to the above described operation of save area stacking for reentrant and recursive routines. If OPTIONS=MAIN is specifie as well as RENT or RECUR, then the check

to see if this routines save/work area will fit in the previous stack is bypassed, and a new stack is always GETMAINed. This is because for OPTIONS=MAIN it is assumed that the previous save area can not possibly be in a suitable stack.

Example:

```
MAIN    BROUTINE   OPTIONS=MAIN, NEWSAVE=(YES,512)
          .
          .
          .
        BCALL      FACTOR,((R1))
          .
          .
          .
        BEND       MAIN

FACTOR  BROUTINE   OPTIONS=(RECUR),NEWSAVE=(YES,144
          .
          .
          .
        BCALL      FACTOR,((R1))
          .
          .
          .
        BEND       FACTOR
```

For this example, assume that neither MAIN nor FACTOR has any work area. The 512 byte stack aquired by MAIN (as a pre-assembled CSECT since MAIN is not reentrant) will hold MAIN's save area (80 bytes) plus up to 6 save areas for FACTOR (72 bytes each, $80 + 72*6 = 512$). Thus FACTOR can call itself recursively 5 times without doing a GETMAIN. Beyond this, a GETMAIN will be done every second recursive call, since

two 72 byte save areas will fit in each 144 byte stack.

(X)

The &ENTRY and &RETURN macros generate calls to out-of-line routines when save area stacking is used. The same out-of-line routines will be used for each use of &ENTRY and &RETURN, thus making the overall program smaller.

	RENT or RECUR	not RENT or RECUR
4. NEWSAVE = NO	~~callers regs saved in his save area~~ no new save area acquired, no work area permitted ~~No calls to other routines are permitted; save areas not chained~~	~~callers regs saved in his save area~~ no new save area acquired, no work area permitted ~~No calls to other routines are permitted; save areas not chained~~
3. NEWSAVE = (YES, symbol)	invalid combination of options	~~callers regs saved in his save area~~ user ~~specified~~ defined save/work area pointed at by RD. ~~No work area may be declared following the DROUTINE. Users save area should be 18 words unless OPTIONS=MVEN and DENTRY has an operand, in which case must be 20 words.~~ ~~Save areas are chained.~~
1. NEWSAVE = YES or NEWSAVE= omitted (default)	~~callers regs saved in his save area~~ the save/work area is developed as a DSECT between the DROUTINE and the first DENTRY. The storage necessary is GETMAINed when the routine is entered and RD is set to point at the GETMAINed core. ~~Save areas are chained.~~	~~callers regs saved in his save area~~ the save/work area is developed as a uniquely named CSECT between the DROUTINE and the first DENTRY. RD is set to point at this CSECT. ~~Save areas are chained.~~

No work area is permitted.
Users save area should be 18 or 20 words; see OPTIONS=MAIN.

(cont)

	not RENT or RECUR	RENT or RECUR
2. NEWSAVE = (YES, number)	~~callers regs saved in this save area~~ the save/work area is developed as a uniquely named CSECT between the BROUTINE and the first BENTRY. It is then extended to the length "number" and the first word is set as a stack prefix word so the remainder of the CSECT can be used as a stack. RD is set to point at this CSECT. ~~The save areas are chained~~	~~callers regs saved in this save area~~ The save/work area is developed as a DSECT between the BROUTINE and the first BENTRY. When the routine is entered, a check is made to see if this save/work area will fit in the previous stack. If so, it is allocated there and RD set to point at it. If not, a GETMAIN is done for "number" bytes, and this core is set up as a new stack with this routines save/work area allocated in it ~~RD~~ and RD pointing at it. ~~The save areas are chained.~~ For this combination of options BENTRY and BRETURN are generated as out-of-line routines

IBM

IBM SYSTEM/360 ASSEMBLER CODING FORM.

PROGRAM		PUNCHING INSTRUCTIONS	GRAPHIC								PAGE OF
PROGRAMMER	DATE		PUNCH								CARD ELEC. NO.

STATEMENT

NAME 1–8	OPERATION 10–14	16–20	OPERAND 25–30	35–40	45–50	COMMENTS 55–60	65–71	IDENTIFICATION–SEQUENCE 73–80
								01
								02
								03
								04
								05
								06
								07
								08
								09
								10
								11
								12
								13
								14
								15
								16
								17
								18
								19
								20

26 Oct 76

Changes to $ROUTINE, $ENTRY, $END

These macros are changed so that the work area of non-reentrant routines is first generated as a DSECT between $ROUTINE and $ENTRY (same as for reentrant routines), and is then allocated via a DS in the $END macro. The $ENTRY macro points at the work area using a LA instruction.

These changes make it necessary to always use the $END macro at the end of a routine. The $END macro also generates a LTORG and a $LTORG (for use with $EX).

also, option added to $ROUTINE, OPTIONS = (MAIN, NOPARM) implies that a dope vector not be allocated to describe the PARM= field from the EXEC card.

8 Feb 77

Updates to my macro library

The #EX macro is extended to accept the operand LOCAL=YES.

The #LTORG macro is also extended to accept the same operand.

When LOCAL=YES is specified on #EX, then the executed instruction is kept seperate from the instructions for which LOCAL=YES was not specified, and can be generated by #LTORG LOCAL=YES without generating the other instructions

example

```
        #EX   R5, (MVC, ABC(*-*), XYZ), LOCAL=YES
        #EX   R2, (MVC, XR1(*-*), XRF)

        #LTORG  LOCAL=YES
+#Litnnn  MVC    ABC(*-*), XYZ

        #LTORG
+#Litnnn  MVC    XR1(*-*), XRF
```