**CSCI 360**

**Computer Programming in**

**Assembler Language**

**Notes and Examples**

**Edited by Geoffrey D. Decker**

**Table of Contents**

[Introduction 4](#_Toc504049635)

[Chapter 1 - Job Control Language (JCL) 5](#_Toc504049636)

[Assist JCL 5](#_Toc504049637)

[JCL for the Input Data Set 7](#_Toc504049638)

[Chapter 2 - Documentation Standards 8](#_Toc504049639)

[Program (or Subroutine) Documentation 8](#_Toc504049640)

[Source Code Documentation 10](#_Toc504049641)

[Macro Documentation Standards 11](#_Toc504049642)

[Chapter 3 - Structured Pseudocode 13](#_Toc504049643)

[Chapter 4 - Debugging Tips 14](#_Toc504049644)

[Chapter 2 of Overbeek & Singletary Text 14](#_Toc504049645)

[Chapter 3 of Overbeek & Singletary Text 15](#_Toc504049646)

[Chapter 4 of Overbeek & Singletary Text 16](#_Toc504049647)

[Chapter 5 of Overbeek & Singletary Text 16](#_Toc504049648)

[Chapter 6 of Overbeek & Singletary Text 18](#_Toc504049649)

[Chapter 8 of Overbeek & Singletary Text 18](#_Toc504049650)

[Chapter 9 of Overbeek & Singletary Text 18](#_Toc504049651)

[Chapter 5 - Common S0Cx Abends 19](#_Toc504049652)

[Chapter 6 - Decimal Conversion 21](#_Toc504049653)

[Input 21](#_Toc504049654)

[Output 22](#_Toc504049655)

[Chapter 7 - Load Address Instruction 23](#_Toc504049656)

[Chapter 8 - Memory Dumps 26](#_Toc504049657)

[Dump Contents 26](#_Toc504049658)

[Dump Assignment 1 28](#_Toc504049659)

[Dump Assignment 2 30](#_Toc504049660)

[Chapter 9 - External Subroutines 32](#_Toc504049661)

[Standard Linkage Conventions 33](#_Toc504049662)

[Register Conventions 34](#_Toc504049663)

[Entry Linkage 35](#_Toc504049664)

[Exit Linkage Method 1 36](#_Toc504049665)

[Calling External Subroutines 38](#_Toc504049666)

[Parameter Lists and Passing Parameters 40](#_Toc504049667)

[Chapter 10 - DSECTs 42](#_Toc504049668)

[Chapter 11 - Macro Instructions & Conditional Assembly 45](#_Toc504049669)

[The IBM Macro Language 45](#_Toc504049670)

[Macro Definition 46](#_Toc504049671)

[Format of a Macro Definition 46](#_Toc504049672)

[MNOTE and MEXIT 49](#_Toc504049673)

[Chapter 12 – Working on the Marist Mainframe 50](#_Toc504049674)

[12.1 Allocating Your Assignment PDSE 50](#_Toc504049675)

[12.2 Editing in ISPF 60](#_Toc504049676)

[12.3 Submitting and Viewing Results in ISPF 67](#_Toc504049677)

[12.4 Using mar\_ftp.exe to Download Marist Output 68](#_Toc504049678)

[12.5 Using FileZilla to Download Marist Output 69](#_Toc504049679)

[12.6 Transferring Files to Marist 73](#_Toc504049680)

[12.7 How to Fix Invalid Characters Found in ISPF 74](#_Toc504049681)

[12.8 Customizing TSO/ISPF 75](#_Toc504049682)

[Appendix A - Search Algorithms 78](#_Toc504049683)

[Linear Table Search 78](#_Toc504049684)

[Hashing 79](#_Toc504049685)

[Hash Search 79](#_Toc504049686)

[Appendix B - Sort Algorithms 81](#_Toc504049687)

[Bubble Sort 81](#_Toc504049688)

[Selection Sort 82](#_Toc504049689)

[Insertion Sort 83](#_Toc504049690)

[Insertion Sort With a Bucket 84](#_Toc504049691)

[Appendix C - Merge Algorithms 85](#_Toc504049692)

[Appendix D – The Marist Mainframe 87](#_Toc504049693)

# Introduction

The intent of this book is to supplement the class readings and lecture notes for CSCI 360: Computer Programming in Assembler Language. It should be used in conjunction with the text Assembler Language with **ASSIST** and **ASSIST/I**; it is by no means intended to replace the text. See the z/OS Principles of Operation manual for further help on the machine instructions, and the High Level Assembler Language Reference for more information on the assembly language.

A little information about your textbook. It was first published in 1976 by Science Research Associates. The authors, Ross A. Overbeek and Wilson E. Singletary, were both faculty members at Northern Illinois University when they wrote the book. In fact, the head administrative assistant, Marge Hargrave, typed the manuscript herself!

The book's first edition contains the following dedication:

This work is respectfully dedicated to the devoted Computer Science faculty at Northern Illinois University, without whose total commitment, encouragement, and support this work and the fine programs that have been developed here would have suffered greatly and perhaps perished.

In their acknowledgements, Drs. Overbeek and Singletary state:

We pledge our undying devotion to Marge Hargrave for the superb job she did typing final chapters of the manuscript, entering final editorial changes, and producing a finished manuscript.

Please know that your textbook, now in its fourth edition, is considered one of the finest on the subject of Assembly Language programming. It is listed in the bibliographies of almost all books and academic papers on the subject of Assembly Language programming published in the more than 30 years since its publication!

It is highly recommended that you buy a copy of the textbook and keep it close at hand throughout your professional programming career. Many who have completed NIU's program will attest that the book is an important part of their professional library.

# Chapter 1 - Job Control Language (JCL)

Once we have an assembler program coded, all we need to do is add JCL and we can run it. Our finished program appears below. See the JCL section for help on JCL. See the Documentation Standards section for more help on documentation standards. Use the Assist JCL unless otherwise told.

## Assist JCL

//**KC0nnnnA** JOB ,'**your name here**',MSGCLASS=H

//\*

//JSTEP01 EXEC PGM=ASSIST

//\*

//STEPLIB DD DSN=KC00NIU.ASSIST.LOADLIB,DISP=SHR

//\*

//SYSIN DD \*

MAIN CSECT

USING MAIN,15 Establish base register 15

\*

*your assembler source code goes here*

\*

BR 14 Exit from the program

END MAIN

/\*

//\*

//FT05F001 DD \*

*input data goes here*

/\*

//\* The following SYSOUT=\* displays your program's output

//\*

//FT06F001 DD SYSOUT=\*

//\*

//\* The following SYSOUT=\* displays the system's and ASSIST's output

//\*

//SYSPRINT DD SYSOUT=\*

//

**SUBSTITUTIONS:**

**jobname** = KC0nnnnA, where KC0nnnn is your assigned Marist logonid and A is required (you can use any

letter but stick with A for now).

**your name here** = will print on your job banner. 1-20 characters.

If your program requires input data from a file, then the JCL after the assembler source code should change as follows:

**//KC0nnnnA JOB ,'your name here',MSGCLASS=H**

**//\***

**//STEP1 EXEC PGM=ASSIST**

**//\***

**//STEPLIB DD DSN=KC00NIU.ASSIST.LOADLIB,DISP=SHR**

**//\***

**//SYSIN DD \***

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* \*

\* *Documentation goes here* ...... \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

MAIN CSECT

USING MAIN,15 Establish base register 15

\*

*your assembler source code goes here*

\*

BR 14 Exit from the program

END MAIN

**/\***

**//\***

**//FT05F001 DD DSN=name-of-file,DISP=SHR**

**//\***

**//FT06F001 DD SYSOUT=\***

**//\***

**//SYSPRINT DD SYSOUT=\***

**//**

Note that you may choose to use a NOTIFY keyword parm at the end of the first line of JCL, just following the MSGCLASS=H. Here is what it would look like:

//KC0nnnnA JOB ,'your name here',MSGCLASS=H,NOTIFY=&SYSUID

The NOTIFY= will automatically fill in YOUR KC-ID as the &SYSUID. After you submit a job, you will get a red message stating you have submitted the job. Then, with the NOTIFY=, you will be notified by a red pop-up stating your job is finished and what the COND CODE was when it finished.

**SUBSTITUTIONS:**

**name-of-file** = When a program requires data from an input disk file. The instructor will give you the name

of the file.

NOTE: When coding JCL, you must code everything exactly as shown, except for the substitution items; otherwise, you will receive JCL errors.

Also note that on the line of JCL that defines the file to be used, **FT05F001** contains only zeroes and not any capital letter Os.

As a matter of fact, neither does **FT06F001**!

## JCL for the Input Data Set

All assignments will be available via web browser, and will have a link to a web copy of any input data set. In case there is no such link, you can use the following JCL to submit a batch job to obtain the input data listing. The following example will print the listing of the input data for an assignment. For each subsequent assignment, just modify the **SYSUT1** statement to reflect the name of the input data set given to you on the assignment sheet.

//KC0nnnnA JOB ,'your name here',MSGCLASS=H

//\*

//JSTEP01 EXEC PGM=IEBPTPCH

//\*

//SYSUT1 DD DSN=**name-of-file**,DISP=SHR

//\*

//SYSUT2 DD SYSOUT=\*

//\*

//SYSIN DD \*

PRINT MAXFLDS=1

RECORD FIELD=(80)

/\*

//SYSPRINT DD SYSOUT=\*

//

**SUBSTITUTIONS:**

**name-of-file** = when a program requires data from an input file. The instructor will give you the name of

the file.

# Chapter 2 - Documentation Standards

## Program (or Subroutine) Documentation

Each program (or subroutine), whether internal or external, should contain a documentation box that looks like the one presented below.

Note: From this point on through this first part of the chapter, the word program will denote both a program and a subroutine.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* PROGRAMMER NAME: \*

\* \*

\* DATE: \*

\* \*

\* PROGRAM NAME: \*

\* \*

\* FUNCTION: \*

\* \*

\* INPUT: \*

\* \*

\* OUTPUT: \*

\* \*

\* ENTRY CONDS: \*

\* \*

\* REGISTER USAGE: \*

\* \*

\* EXIT CONDS: \*

\* \*

\* NOTES: \*

\* \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

PROGRAMMER NAME: This should be YOUR name. If you are working with another programmer and have been allowed to do so, put ALL of the programmer names as in the following example:

PROGRAMMER NAME: SALLY SMITH

TOM BROWN

DATE: The date that the program is due.

PROGRAM NAME: This should be the name of the program. This is usually the first label in the program, that the caller references (e.g., a CSECT label for programs or external subroutines).

FUNCTION: This should be a one or more line description of what the program is supposed to do.

INPUT: This is a description of the physical data that is read by this routine. A routine only has input if it contains an XREAD command. The content of a table is NOT input, it is storage. If there are no inputs, you must say 'NONE'.

OUTPUT: This is a description of any information sent to the printer, such as reports. A program only has output if it contains an XDUMP or XPRNT command. If there are no outputs, you must say 'NONE'.

ENTRY CONDS: This stands for Entry Conditions. This should describe the parameters passed to the program. If there no parameters passed, then you must type in 'NONE'. Examples of valid entry conditions follow later in this section.

EXIT CONDS: This stands for Exit Conditions. This should represent what registers the program will purposely change before returning back to the calling program (for external programs (or subroutines) only register 15 and 0 can be used) . If there are no register exit conditions, you must type in 'NONE'. Examples of valid entry conditions follow later on in this section.

**Examples of Entry and Exit Conditions for Internal Subroutines:**

ENTRY CONDS: R3 -- Contains the address of beginning of table

R4 -- Contains the address of the card

EXIT CONDS: R5 -- Contains the address of the end of table

The above may be abbreviated for space as follows.

ENTRY CONDS: R3 -- @(BEGINNING OF TABLE)

R4 -- @(CARD)

EXIT CONDS: R5 -- @(END OF TABLE)

Examples using a parameter list:

ENTRY CONDS: R1 -- @(PARMLIST)

0(R1) -- @(BEGINNING OF TABLE)

4(R1) -- @(CARD)

8(R1) -- @FULLWORD[@(LOGICAL END OF TABLE)]

EXIT CONDS: R15 -- CONTAINS RETURN CODE

0 -- MEANS SUCCESSFUL

4 -- MEANS UNSUCCESSFUL

REGISTER USAGE: Register usage should describe what registers are being used consistently in the

following following section of code. An example follows:

R6 -- USED FOR THE $TABLE DSECT USING

R7 -- CONTAINS THE ADDRESS OF THE END OF THE TABLE

R15 -- USED AS A BASE REGISTER

## Source Code Documentation

**NOTE:** You **MUST** follow all documentation standards beginning with the second programming

assignment!

**Line Documentation**

* You must provide Line Documentation on 80% or more of executable instructions or execution of your program will be suppressed.
* Documentation should usually start in column 36 and should be kept aligned.

**Example**

LA 3,TABLE Load R3 with @ of table <- Bad!

LA 3,TABLE Get table beginning @ <- Good

AR 3,4 Add R4 to R3 - Bad doc!

AR 3,4 Add amount to total <- Good

XDECI 6,CARD Put converted number in <- Good

ST 6,TABLE The table

MVI PLINE,C' ' Clear the <- Good

MVC PLINE+1(132),PLINE print line

* Documenting storage:

CARD DS CL8O Input area

PLINE DC C12'THE SUM IS:' (Don't have to document).

SUM DS CL12 Storage area for the total.

NAME DS CL20 Storage area for name.

## Macro Documentation Standards

The following minimal documentation should be placed in all macro definitions.

1. Describe the function of the macro; what it does for the user. Do NOT describe how the macro works. To the user, the macro is just an instruction and the user only wants to know how to use it.
2. Describe the symbolic parameters including coding specifications.

**Documentation Example**:

.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

.\* NAME: CHANGE \*

.\* \*

.\* FUNCTION: TO CONVERT THE VALUE IN &REG INTO ITS \*

.\* DECIMAL EQUIVALENT AND TO EDIT THE VALUE \*

.\* INTO THE 12 BYTE OUTPUT AREA DESIGNATED BY \*

.\* THE &PRNTFLD ADDRESS. \*

.\* \*

.\* PROTOTYPE STATEMENT: \*

.\* &LABEL ZDECO &REG,&PRNTFL \*

.\* \*

.\* SYMBOLIC PARAMETERS: \*

.\* &LABEL AN OPTIONAL LABEL FOR THE FIRST STATEMENT \*

.\* GENERATED \*

.\* &REG REQUIRED OPERAND IN THE FORM OF A REGISTER \*

.\* NUMBER CONTAINING THE BINARY VALUE TO BE \*

.\* CONVERTED TO A PRINTABLE FORMAT. \*

.\* &PRNTFLD A REQUIRED OPERAND IN THE FORM OF ANY VALID \*

.\* RX-TYPE ADDRESS INTO WHICH THE DECIMAL \*

.\* EQUIVALENT OF THE BINARY VALUE IS TO BE \*

.\* EDITED. \*

.\* \*

.\* ERROR CONDITIONS: \*

.\* IF EITHER &REG OR &PRNTFLD IS MISSING, AN MNOTE AND \*

.\* MEXIT ARE ISSUED. \*

.\* \*

.\* NOTES: \*

.\* THE CONTENTS OF CALLERS REGISTERS 1 AND 2, AND THE \*

.\* CONDITION CODE, ARE SAVED AND RESTORED UPON EXIT \*

.\* FROM THE MACRO. \*

.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

NAME: This should be the name of the macro.

FUNCTION: This should be a one or more line description of what the macro is supposed

to be used for.

PROTOTYPE: This should show what parameters this macro accepts and the order the

parameters should be specified to call the macro.

SYMBOLIC PARMS: This is a description of all symbolic parameters, what they mean, any

defaults they may have, and if they are optional. If there are no parameters,

then 'NONE' must be specified.

ERROR CONDS: This should be a listing of any MNOTE messages that may be generated by

this macro, as well as under what conditions they will be generated. If there

are no error conditions, you must say 'NONE'.

NOTES: This should specify any special programming techniques that this macro uses

that someone looking at your program should know about.

**Internal Macro Documentation:**

There are two kinds of documentation which should appear within a macro definition: documentation which will appear as generated assembler source, and documentation which describes the operation of the macro instruction itself, particularly conditional assembly.

The first, assembler source documentation, should be written exactly as is done in a standard assembler language program.

The second, macro language documentation, should appear as both comment blocks (using the .\* form of macro comment) and as line doc on conditional assembly statements.

Remember, *there is never “too much” documentation*.

# Chapter 3 - Structured Pseudocode

Structured programming provides the programmer logical constructs from which to build efficient programs. It facilitates simplified logic, easier debugging, and better maintenance. Structured programming is best demonstrated through the use of pseudocode. Pseudocode is an English like language that allows the description of logic to be written, free from the syntax of a formal computer language. Therefore, a particular set of pseudocode becomes portable. That is, it may be shared with different programmers, each of which may work with a different language, and it will be readily understood. Actual computer programming simply becomes the translation of pseudocode into a computer language. The following are four basic control constructs that we will use in pseudocode to give our programs good structure:

\* sequence (straight-line)

\* if-else

\* if-elseif-else

\* do-while

In addition, these four constructs may be layered within themselves. For example, a do-while construct may contain an if-else construct, which definitely will contain a sequence construct. To better demonstrate the use of structured pseudocode, both computer code examples and examples relating to something from everyday life will be given. Remember, pseudocode is a description of logic, therefore it can define any process.

# Chapter 4 - Debugging Tips

## Chapter 2 of Overbeek & Singletary Text

1. Spelling mistakes.

**NUM** for **NUMB**.

1. Omit **USING** statement. Will cause a lot of addressability errors.
   1. Failure to assign and load enough base registers for a long program.
   2. Using the base register for another purpose in the program thereby destroying the base reference.
2. Violating rules for declarative.
   1. DS with a nominal value, which is viewed as documentation.
   2. DC without a nominal value.
   3. Wrong length on a DC --- **DC CL3'DATE'**.
   4. Wrong nominal value coded --- **DC CL5'0'** gives '0bbbb' rather than '00000'.
3. Select the correct instruction.

**LR 7,4** LOAD CONTENTS OF R4 INTO R7.

**L 7,4** LOAD CONTENTS OF BYTES 4, 5, 6, 7 INTO R7.

**LA 7,4**  LOADS THE 4 INTO R7.

Errors may not occur if you use the wrong instruction, but your output will be incorrect.

1. Binary division requires that the even register be initialized with the sign bit of the odd register.
2. Binary calculations may generate unexpectedly large values that exceed the capacity of one register.
3. Use **XDUMP**s to verify register contents and storage.
4. When using **EQU** be sure you code in what you want.

For example:  **R12 EQU 13**

Whenever you say **R12** the compiler will interpret it as a 13.

1. Remember the rules of using **literals** and the **LTORG** statement.
2. When using apostrophes, ampersands, or quotes you are required to use two of them next to each other in a literal in order to represent just one of them.

**C'T&&D''S' ---> E350C47DE2**

## Chapter 3 of Overbeek & Singletary Text

1. Incorrect use of the mnemonic branching, reversing operand 1 and 2, using **BNH** rather than **BNL**.
2. The increment value for stepping through the table may be wrong.
3. The beginning or end of table address may be wrong.
4. If beginning or end of table addresses are wrong, you may never find the end of table and get into an infinite loop. You will probably end up with a protection exception as you go through memory.
5. Work in steps. Build table, dump and verify. Sort table, Dump and verify. Print table, dump and verify.

## Chapter 4 of Overbeek & Singletary Text

**NOTE:** If an instruction is not completely understood and may be causing an error, check the assembler manual to insure for it's correct use.

1. Coding a length on **MVI** or **CLI** or on the second operand of **MVC** or **CLC**.
2. Be sure that input definition agrees with actual input record. Any differences may cause garbage results.
3. Omitting explicit length on an **MVC** statement with relative addressing:

**MVC PRINT+95,=C'DATE'**

The computer will move 133 bytes starting from the first byte of

DATE.

1. Using Literal ("=") with a **CLI** or **MVI**.

## Chapter 5 of Overbeek & Singletary Text

Assembly errors:

1. Coding packed **DC**s (type **P**) with values other than digits 0-9.

1. Coding hex data - pads and truncates on the left.

**DC XL3'FF'** generates '0000FF' rather than 'FFFFFF'

Logic errors:

1. Packing a field that contains a blank will give an invalid sign (X'04') and an attempt to do arithmetic with this field will cause a data exception. Most likely cause is an invalid input record.
2. Another cause of a data exception is using a field not initialized by DC or defined as Character or Hex.

1. Watch out for using **MVC** and **CLC** on packed data (incorrect execution) and **ZAP** or **CP** on character data (Data exception).
2. Watch for improper relative addressing.

1. Missing explicit lengths can cause unexpected results.
2. Double check **MVC**/**ED** operations.
   1. There must be an odd number of digit selectors.
   2. Print area field must be the same as the edit field.
3. Watch for too short a field in **MP**. Data exception.
4. Make sure field is big enough on **DP**. Decimal divide.
5. Make sure you don't divide by zero. Decimal divide.
6. As well as technical misuse of instructions,
   1. Wrong compares
   2. Wrong branches
   3. Wrong calculations
7. Operand two of **CVD** or **CVB** must be a double word.
8. Analyze the programming problem well before you begin to code.

## Chapter 6 of Overbeek & Singletary Text

1. Preventive programming is your best defense.
2. Be careful about register use (very common mistake although simple, it could take a while before it is noticed). Write down what you are using the registers for in each routine.
3. If you have problems with external linkage, double check that the standard linkage was entered correctly. It is easy to reverse registers.
4. As soon as you get the parameters in a subroutine, dump the registers so you can verify that you have the right parameters.
5. Make sure registers used with **DSECT**s contain the right values.
6. Remember to **DROP** all base registers when you're done with the **DSECT**s.

## Chapter 8 of Overbeek & Singletary Text

1. Registers 1 and 2 are implicitly altered by the **TRT** instruction.
2. When using a **LH** instruction the two leftmost bytes will be changed to the sign bit.

## Chapter 9 of Overbeek & Singletary Text

1. A macro loop is different from an assembler loop.

1. **REMEMBER:** do not mix assembler statements with macro processing statements. For example

**AIF ('' NE '').FOUND**

**.FOUND DS 0H <---- WRONG**

**.FOUND ANOP <---- OK**

1. Do not use **R** with the register number or use literals in a macro, because you do not always know if the programmer who is going to use this macro is using **EQUREGS** or **LTORG**s in his program.

# Chapter 5 - Common S0Cx Abends

* **S0C1 - Operation code.**
  + You get this error when an attempt is made to execute an invalid operation code.
  + Probable causes:
    - a missing branch instruction at the end of the routine.
    - code overwritten; compare the typed instruction at the abending address with the contents of storage in the dump at the abending address.
* **S0C4** - **Protection Exception.**
  + Attempt is made to refer to a storage not allocated to your program.  
    - Probable causes: Invalid value in the base or index register due to any prior instruction.
* **S0C5 – Addressing Exception.**
  + Attempt is made to refer to a storage address that does not exist anywhere.  
    - Probable causes: Invalid value in the base or index register due to any prior instruction.
* **S0C6** - **Specification Exception.**
  + With RX instructions: Address of the storage referred to is not on a full word boundary;

Invalid value in the base or index register.

* + With **MP/DP:** Invalid length specified for the receiving field.
* **S0C7** - **Data Exception.**
  + Attempting decimal arithmetic with a field containing an invalid packed number.
* **S0C9** - **Fixed point divide exception**.  
  + The quotient does not fit in the odd register of the even/odd pair.
  + Attempt to divide by zero.
  + This error occurs mostly because the even register does not contain the extension of the sign bit.
* **S0CB** - **Decimal Divide Exception.**
  + Attempt to divide by zero.
  + The quotient after DP does not fit in the specified area.

# Chapter 6 - Decimal Conversion

To facilitate numeric input/output, ASSIST accepts the commands XDECI (eXtended DECimal Input), and XDECO (eXtended DECimal Output). XDECI can be used to scan input cards for signed or unsigned decimal numbers and convert them to binary form in a general purpose register, also providing a scan pointer in register 1 to the end of the decimal number. XDECO converts the contents of a given register to an edited, printable, decimal character string.

Both instructions follow the RX instruction format, as shown:

**XDECI R1,D2(X2,B2)** **XDECO R1,D2(X2,B2)**

where **R1** is any general purpose register, and **D2(X2,B2)** is an RX-type address, such as **LABEL 0(4,5) LABEL+3(2)**

## Input

XDECI is generally used to scan a data card read by XREAD. The sequence of actions performed by XDECI is as follows:

1. Beginning at the location given by **D2(X2,B2)**, memory is scanned for the first character which is not a blank.
2. If the first character found is anything but a decimal digit or plus or minus sign, register 1 is set to the address of that character, and the condition code is set to 3 (overflow) to show that no decimal number could be converted. The contents of **R1** are not changed, and nothing more is done.
3. From one to nine decimal digits are scanned, and the number converted to binary and placed in **R1**, with the appropriate sign. The condition code is set to 0 (0), 1 (-), or 2 (+), depending on the value just placed in **R1**.
4. Register 1 is set to the address of the first non-digit after the string of decimal digits. Thus **R1** should not usually be 1. This permits the user to scan across a card image for any number of decimal values. The values should be separated by blanks, since otherwise the scanner could hang up on a string like -123\*, unless the user checks for this himself. I.e., XDECI will skip leading blanks and signs but will not itself skip over any other characters.
5. During step 3, if ten or more decimal digits are found, register 1 is set to the address of the first character found which is not a decimal digit, the condition code is set to 3, and **R1** is left unchanged. A plus or minus sign alone causes a similar action, with R1 set to the address of the character following the sign character.
6. In summary, the condition code is set to

0 - if converted number was 0

1 - if converted number was < 0

2 - if converted number was > 0

3 - if an invalid character is found, or the number was too   
 long (or short)

And R1 will point to the blank after the last digit of the   
 number.

**NOTE**: **XDECI** is not at all linked to **XREAD**. You can convert any number from character to binary by providing proper address. It is mostly used with **XREAD** because the data is generally specified in character format and needs to be converted (numbers only) to binary format for calculations.

**NOTE**: Never use **R1** as the receiving field on **XDECI** instruction. To be extra careful, expecially at the beginning of the semester, DO NOT use **R1** for anything else if **XDECI** is being used in the program.

## Output

XDECO converts the value in **R1** to printable decimal, with leading zeroes removed, and a minus sign prefixed if needed. The resulting character string is placed right-justified in a 12-byte field beginning at **address**. It can then easily be printed using an XPRNT instruction. The XDECO instruction modifies NO registers.

**D2(X2,B2)** - must be a 12 byte field where you want to store the converted number (Normally on the print line).

# Chapter 7 - Load Address Instruction

The Load Address (LA) instruction is in the RX format:

**label LA R1,D2(X2,B2)**

* Execution of this instruction causes the ***address*** given by **D2(X2,B2)** to be calculated and placed in **R1**.
* An address is only three bytes (24 bits) long, so the leftmost byte of **R1** is set to 00.
* You must also remember all the other rules governing the address calculation:
  + R0 is not considered in address calculation
  + The address is only three bytes (fourth byte, the leftmost byte, is X'00').

For example, if R5 contains 12345678

and R6 contains 34567890

then the second operand of

**LA 3,30(6,5)**

will be calculated as an address in the following way:

12345678

+34567890

+ 1E

---------

008ACF26 (Note that the resulting high-order byte is zero!)

So, after execution **R3** will contain 008ACF26. Likewise,

**LA 3,0(0,5)**

will put 00345678 + 0 + 0 or 00345678 in **R3**.

**The Difference Between LA and L**

As mentioned above **LA** puts the address itself in the receiving register whereas **L (load)** picks up the four bytes at the specified memory address and put those bytes in the receiving register.

LOC OBJECT CODE

0000E0 00000014 WORD DC F'20'

Thus, **LA 3,WORD** will put the address of location **WORD** in **R3**, so **R3** will contain 000000E0 after this instruction has been executed.

On the other hand, **L 3,WORD** will get the contents of the four bytes of memory beginning at location **WORD** and put them in **R3**, so **R3** will contain 00000014 after this instruction has been executed.

What would be in **R2** after the following two instructions have been executed???

**LA 2,WORD**

**L 2,0(,2)**  **R2 = \_\_\_\_\_\_\_\_\_\_\_\_**

All RX instructions are encoded as **OORXBDDD**, so we have three hex digits (**DDD**) available for displacement. This means that the maximum displacement is **X’FFF’**, which is 4095 decimal (the maximum possible displacement which can be coded on any instruction).

Another common mistake with **LA** is the idea that the address of a register can be loaded into another register. Thus,

**LA 3,5** <--- The misconception is that the address of R5 will be loaded into R3. But, there is no address for a register. Furthermore, just because we have "R" in front of "5" it does not become a register. In this case **LA R3,5** will produce the same result as **LA R3,R5**.

**Common Uses of Load Address**

1. To get the memory address of a field.
2. To set a register to any integer value between 0 and 4095

**LA 5,10** - will put 0000000A in R5 because there are no base and/or index registers; thus the displacement will be 10.

The same thing could also be done by **L 5,=F'10'** but this takes 8 bytes (4 for literal and 4 for instruction) and thus is less efficient (even though clearer).

1. To increment a register by any integer value between 0 and 4095

**LA 5,20(,5)** will increment R5 by 20. The same thing could be done by using the instruction **A 5,=F'20'** which would be clearer but not as efficient.

**WARNING**: Remember that address calculation drops the leftmost byte, so if the value in the register is negative or a big positive number then you must NOT use LA for incrementing the contents of a register.

Assume that R3 contains 12345678

What will be in R3 after the following instruction has been executed?

**LA 3,8(,3)** **R3 = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Chapter 8 - Memory Dumps

# Dump Contents

If your program **ABEND**s (**AB**normally **END**s), ASSIST generally will provide you with a memory dump which can help you isolate the reason for the ABEND. The only instances in which you will not get a dump are those in which your job exceeds the time limit (no time remains to generate the dump), or the job generates more than the number of lines allowed (2000 is the default maximum number.)

The information available to you in a dump includes:

* **The contents of the PSW**. See your yellow card and/or page 498 of your text for a description of a BC mode PSW. You should learn to be able to extract from the PSW any of the information which it contains.
* **The completion code**. See your yellow card for a listing of program interruption codes - also, appendix D of your text gives a good explanation of the more commonly encountered completion codes, along with various programmer errors which can cause those types of interrupt.
* **A trace of the last few instructions executed**.
* **A trace of the last few branch instructions executed**.
* **Contents of the 16 general purpose registers** at the time of the ABEND. Since you are not going to be using the floating point registers, you can ignore them.
* **The contents of user storage** (the portion of main storage used by your program and its save areas) is dumped in hexadecimal. Each line contains 32 bytes of storage. In the left-hand margin you will find the address of the first of these 32 bytes (LOC.) On the right-hand margin you will find (between two \*s) a translation into character form, where alphabetic and numeric characters and blanks are identified whenever a byte contains the character's encoded form a period is printed to represent any other byte values.

When your program ABENDs, you should be able to answer questions such as:

1. What was the reason for the ABEND (interruption code)?
2. What does that interruption code mean?
3. What was the last instruction executed?
4. Did the registers contain the "right" values?
5. Were the contents of user storage correct?

## Dump Assignment 1

Type in and run the following program:

**DUMP1 CSECT**

**USING DUMP1,15**  ESTABLISH A BASE REGISTER

**L 1,ONE**  LOAD THE FIRST NUMBER INTO R1

**L 2,TWO**  LOAD THE SECOND INTO R2

**AR 1,2**  ADD THE TWO NUMBERS

**ST 1,THREE**  STORE THE RESULT

**XDUMP THREE,4**  DUMP THE RESULT

**BCR B'1111',14** RETURN TO CALLER

**\***

**ONE DC F'64'**  FIRST NUMBER

**TWO DC F'32'** SECOND NUMBER

**EOFFLAG DC C'0'** A FLAG SAVE AREA

**THREE DS CL4' '** SUM OF THE TWO NUMBERS

**END DUMP1**

After running the above program you should be able to answer the following questions:

1. What is the address of the next instruction which will be executed?
2. What is the address of the instruction that caused the abend?
3. What type of error occurred?
4. What actually causes this error?
5. Correct the error by rewriting the section of code that caused it.
6. What is the contents of register 1 in decimal?
7. What does the value in reg 1 represent at the time of ABEND?
8. Why is the LOC address of the storage area with the label ONE on it 000018 when the branch statement before it whose LOC address is 000014 only takes up 2 bytes?
9. What are the contents of the two bytes of user storage starting at address 000016? What do they represent?
10. What are the contents of the byte saved at address 00001B? Does this byte represent the first byte of a full word?
11. Which of the following are synonyms for the same length? [There may be more than one group of synonyms.]
    1. 6 hex digits d. 32 bytes g. fullword j. 64 bits
    2. 4 bytes e. 32 bits h. byte k. halfword
    3. 8 hex digits f. doubleword i. foot l. meter
12. If the dump program error were corrected, what value would the storage area at label THREE contain?
13. What two instructions have you worked with which cause data conversion to take place?
14. What is the decimal equivalent of hex 0002BA14?

## Dump Assignment 2

Type in and run the following program:

**DUMP2 CSECT**

**USING DUMP2,15**

**LA 2,TABLE**

**SR 3,3**

**XREAD DATA,80**

**LOOP1 BM ENDLOOP1**

**XDECI 4,DATA**

**ST 4,0(2,3)**

**LA 3,4(,3)**

**XREAD DATA,80**

**B LOOP1**

**ENDLOOP1 SR 3,3**

**LA 7,TABLE**

**LA 5,TABEND**

**LOOP2 CR 2,5**

**BE ENDLOOP2**

**L 6,0(,2)**

**ST 6,0(,7)**

**L 7,4(,7)**

**LA 2,4(,2)**

**B LOOP2**

**ENDLOOP2 BR 14  
\***

**LTORG**

**\***

**DATA DS CL80**

**TABLE DC 30F'-1'**

**TABEND DS 0X**

**END DUMP2**

0

1 2

50

32 24 19 62

123 456 789

987 654 321

Using the results from the program, answer the following questions:

1. What was the interruption code?
2. What instruction caused the program to abend? Why?
3. What was the condition code at the time of the ABEND?
4. How many table entries were built? How did you figure this?
5. What is the return address to the calling routine? Where did you find this? Does your answer really make any sense?
6. What are the contents of register 7?
7. Was any object code changed by this program? If so, for which instructions?
8. Explain why the program ABENDed.

# Chapter 9 - External Subroutines

Until now we have been using internal subroutines; that is, subroutines which are actually part of the program that invokes them. It is much more useful to have subroutines which can be used by any program. These routines are called **external** subroutines because they can be created and maintained separately from the routines that invoke them.

Two important factors make the use of external subroutines possible:

1. Subroutine source decks can be assembled separately and the object module can be stored into an object module library (a collection of object modules which reside on a storage device like a disk).
2. The loader accepts as input not only an object module, but also an object module library. From these, the loader can construct an executable program that uses any required subroutines from the object module library.

When we use subroutines that were written separately, there are certain things we need to know about that module:

1. What parameters does it expect?
2. What does it return to you?
3. Are any registers altered by the routine? (If any were altered, this would affect the way you write your routine, as you would have to avoid using the altered register)
4. What register is used to return from the routine? (In other words, into which register do you want to put the return address)
5. How are the parameters passed?

**NOTE:** Use these notes (rather than the text) as your reference for this subject.

## Standard Linkage Conventions

The standard linkage convention answers many of the problems mentioned above. In this convention, both the calling routine and the called routine have responsibilities.

**Responsibilities of the Calling Routine**

1. Establish an 18 word register save area, and put its address into **register 13**.
2. Set up a parameter address list of input parameters to the routine to be called. The address of this list is put into **register 1**.
3. Put into **register 14** the address of the location in the calling program to which control is to be transferred on returning from the called routine.
4. Obtain the address of the routine to be called, and put that address into **register 15**.
5. Transfer control to the called routine by branching to the address in **register 15**.

**Responsibilities of the Called Subroutine**

1. On Entry
   1. Store the contents of **R14**,**R15**,**R0**,**R1**-**R12** in the calling routine's save area.
   2. Establish addressability.
   3. Establish an 18 word register save area of its own, and link it to the calling routine's save area by:
      1. Putting the address of the calling routine's save area in it's own save area.
      2. Store the address of its save area in the calling routine's register save area.

* 1. Put the address of its own save area into **R13**.

1. Perform Function — Obtain the input parameters from the calling program (**R1** is pointing to these), perform its function, and store any results to be returned to the calling program.
2. Return

* 1. Obtain the address of the calling routine's save area from its own save area and put it into **R13**.
  2. Restore the contents of **R14**,**R15**,**R0**,**R1**-**R12** from the calling routine's save area.
  3. Return control to the calling program by branching to the address in **R14**.

## Register Conventions

The OS Subroutine Linkage convention also includes some register conventions. These are:

**Register 0** - On returning to the calling program, **R0** may contain the output from a subroutine whose entire output is a single numeric value (not common - usually done only in FORTRAN programs).

**Register 1** - On entry to the called program, **R1** contains the address of a parameter address list. This list contains the address of the input parameters to be used by the called routine and/or the addresses of locations into which the called program is to store its output.

**Register 13** - **R13** contains the address of the save area of the current routine.

1. On entering the called routine **R13** contains the address of the calling routine's save area.

1. The called routine puts the address of its own save area in **R13** while it performs its function.

1. On return from the called routine the address of the calling routine's save area is restored in **R13**.

**Register 14** - On entry to the called routine **R14** contains the address of the instruction in the calling routine to which control will be transferred on return from the called routine.

**Register 15** - On entry to the called routine **R15** contains the address of the entry point in the called program. Upon exit from a subroutine **R15** may also be used to pass a return code back to the calling routine indicating the success or failure of the subroutine.

***Coding Conventions***

On entry, a called routine may satisfy its responsibilities by using the following sequence of code. It is normal practice to set up **R12** as the base register.

You should memorize the following sections of code, so that you need not spend a great deal of time thinking about them each time you use them. In addition you should understand them, since other installations may use slight variations on them, and since you will need to interpret (not just regurgitate) them on exams and quizzes.

## Entry Linkage

Upon entry to any routine (including the main routine) the following code should appear. Standard linkages are also described in your text in chapter 6.

**1**  **rtnname CSECT**

**2 STM 14,12,12(13)**

**3 LR 12,15**

**4 USING rtnname,12**

**5 LA 14,savearea \* note**

**6 ST 13,4(,14)**

**7 ST 14,8(,13)**

**8 LR 13,14**

**NOTE:** **savearea** will be defined in the storage area as 18 fullwords.

1. Each routine must have a unique name and a CSECT statement.

1. The called routine must first store all registers in the save area reserved by the calling routine. Remember that the calling routine set up the save area and pointed R13 to it before invoking this routine.
2. R15 will not be used as a base register since it has other usages. Instead, R12 will be used as the base register. This statement puts the address of the routine in R12. (remember that R15 already has the address of the routine)

1. This statement will establish addressability for the routine, so that the labels can be converted to explicit addresses.
2. Puts the address of the 18 fullword savearea into R14. Remember, we cannot put the savearea address into R13 just yet, because we have not saved its' contents.
3. This statement fills in the backward pointer.

1. This statement fills in the forward pointer.

1. The location of the current routine's save area is now put into R13, so that any other subroutines may be called (if necessary). Remember that when a routine is called, R13 must contain the address of the calling routine's save area.

## Exit Linkage Method 1

To exit a routine (return to the calling routine), the following code should be used:

**1 L 13,4(,13)**

**2 LM 14,12,12(13)**

**3 BR 14**

1. This regains the location of the calling routine's save area, restoring R13 to the value it had when the correct routine was entered. Remember that R13 points to the current routine's save area, and the second word of that save area points to the caller's save area. So that backward pointer (4 off of R13) can be accessed to restore R13.

1. This restores all of the other registers from the save area. It is the reverse of statement #2 in the entry code previously discussed. (If R15 is used to pass back a return code - indicating the success or failure of the routine - it should not be restored. In such a case, two statements will be required here, one to load R14 and one to load R0-R12).

1. Finally R14, containing the location of the next instruction to execute upon return to the calling routine, is used to return to the calling routine.

***Exit Linkage Method 2 (Uncommon)***

For use when passing a return code (through register 15) and a value (through register 0).

**1 L 13,4(,13)**

**2 L 14,12(,13)**

**3 LM 1,12,24(13)**

**4 BR 14**

***Exit Linkage Method 3 (Common)***

For use when passing a return code (through register 15).

**1 L 13,4(,13)**

**2 L 14,12(,13)**

**3 LM 0,12,20(13)**

**4 BR 14**

Exit Linkage Method 4 (Uncommon)

For use when passing a value (through register 0).

**1 L 13,4(,13)**

**2 LM 14,15,12(13)**

**3 LM 1,12,24(13)**

**4 BR 14**

**NOTE:** The Operating System (MVS) itself follows standard linkage conventions when it passes control to a user main program (which is viewed as an external subroutine as far as the operating system is concerned).

## Calling External Subroutines

In order to call a subroutine, we need to know its address in our program. The problem is, when you code and assemble your program, you don't know where the subroutine is.

We can't use type A address data (A-data) because these addresses **must** be resolved at assembly time and an external subroutine may be maintained separately and may reside in the object module library.

Type **V** address data (V-data) solves this problem. The value generated at assembly time will appear to be a fullword of zero. Then the address of the subroutine will be filled in by the loader when the executable program is constructed in memory. Also, note that the use of V-data in a DC will not produce the address of the routine during assembly time, as shown below.

**000000 SUBADDR DC V(SUBRTN)**

------ is generated. But by execution time, this will have been filled in.

**L 15,=V(SUBRTN)**

**BALR 14,15**

**NOTE**: these two lines of code fulfill our standard linkage requirements:

**R15** contains the address of the subroutine

**R14** contains the address to return to (in the calling routine).

***Calling External Subroutines - Examples***

The following examples illustrate typical sequences of code which can be used to pass control to another routine.

**Sequence 1:**

**LA 1,PARMLIST points R1 at parameter address**

**\* list**

**L 15,=V(SUBRTN) put address of entry point of**

**\* subroutine in R15.**

**BALR 14,15 Pass control to routine**

**.**

**.**

**.**

**PARMLIST DC A(INPUT) parameter address list**

**DC A(RESULT)**

**\***

**INPUT DC F'10' some input to subroutine**

**RESULT DS F subroutine can put results here**

**\***

**Sequence 2:**

**LA 1,=A(INPUT,RESULT)**

**\* Point R1 at parameter address list**

**L 15,SUBADR Put address of entry point of**

**\* subroutine in R15.**

**BALR 14,15 Pass control to routine**

**.**

**.**

**.**

**SUBADR DC V(SUBRTN)**

**\***

**INPUT DC F'10' some input to subroutine**

**RESULT DS F subroutine can put results here**

Now we can discuss how pass and retrieve from parameter lists!!

## Parameter Lists and Passing Parameters

**Parameter Lists**

The information sent to the subroutines is referred to as parameters. You have been passing information to the subroutines in registers. For example when calling a routine for building a table you might put a table address in **R2**, input area address in **R3**, end of table (next available entry) in **R4** etc. (here address of table, address of input area, and address of end of table storage area are the three parameters).

When working with an external subroutine which is maintained separately you cannot assume that the subroutine is using **R2** for a table address and **R3** for an input address etc. Keeping these things in mind, a standard convention was developed.

Under these conventions all the addresses sent to a subroutine are placed in a separate storage area, called a parameter list. (**R1** is made to point at that storage.) When the control is transferred to the subroutine, it knows that all the necessary information is placed in a storage area where R1 is pointing. Thus, it can pick all the necessary information from the parameter list.

You have to remember that when information is sent to a subroutine, the actual storage does not move. You are just informing the subroutine where the necessary things are located in the calling routine by providing addresses.

Your parameter list must be fully documented in the **ENTRY CONDITIONS** of your subroutine. The parameter list must contain space for not only anything you pass to the subroutine, but ,also, for anything you need to return to the calling routine.

For example a parameter list might look like:

**LA 1,PLIST**

**L 15,=V(BUILD)**

**BALR 14,15**

**...**

**PLIST DC A(TABLE,INPUT,@NAV,RESULT) <-- PARAMETER LIST**

**@NAV DC A(TABLE) Address of next available entry**

**TABLE DS 90F**

**RESULT DS F**

**INPUT DS CL80**

Or

**LA 1,=A(TABLE,INPUT,@NAV,RESULT)**

Or

**LA 2,TABLE**

**LA 3,INPUT**

**LA 4,@NAV**

**LA 5,RESULT**

**STM 2,5,PLIST**

**LA 1,PLIST**

**...**

**PLIST DS 4F**

Now let's receive the parameters inside the called subroutine:

**BUILD CSECT**

Standard linkage <-- discussed in previous section

.

.

**LM 2,5,0(1)** Get all the parameters at the

\* beginning of the program

\*

\* Now: **R2** contains address of **TABLE**

\* **R3** contains address of the **INPUT** area

\* **R4** contains address of storage where **BUILD**

\* will put the address of the next available entry

\* **R5** contains the address of the result area

.

.

Now the build routine is done, and you need to send back a value (let's say in **R7**) to the calling routine. Simply, store the value into the zero off of **R5** (assuming that **R5** still has the address of the area where the result must be stored.)

**ST 7,0(,5) R5-> @ of RESULT area**

**NOTE**: Remember that you can have as many parameters in a parameter list as you want (depending on your needs). Furthermore, it is possible to use just one parameter list when information sent to many subroutines is the same. This is left to the programmer's judgment.

# Chapter 10 - DSECTs

Syntax: label **DSECT**

***General Discussion***

Just like a CSECT, a dummy section or **DSECT** defines the *format* of an area in storage. Unlike a CSECT, it generates no object code. (No storage is defined.) The end of a DSECT is delimited by another DSECT, CSECT or END statement. A DSECT has no value except as the base address of a USING instruction. Of course, a CSECT or any other label can be a USING base, but a DSECT will save some storage. (See the USING Tutorial elsewhere in this book.)

Suppose TABLE consists of 40 entries and each entry contains student name and student social security number.

**TABLE DS 40CL29**

Now let us assume that R3 points at the beginning of the table. In order to refer to the name, we would use something like 0(,3)and to refer to dept, 20(,3). It would be preferable to refer to each field by a name (implicit address) rather than by a base and displacement (explicit address); it would make the program easier to understand.

The **USING** instruction makes it possible to refer to each field by a meaningful name.

**Note**: It is good beginning practice to start **DSECT** labels with a few common characters. It is not a requirement but it helps in identifying a **DSECT** label from a regular label.

So for the above table the **DSECT** could be defined as

000000 **TABLNTRY DSECT** , The name of the DSECT

000000 **TABLNAME DS CL20** The location counter always

000014 **TABLSSN DS CL9**  starts at 0 for each DSECT

00001D **TABLNEXT EQU \*** Start of next entry

The **DSECT** defines the format of one table entry. Again, it does not reserve any storage.

In our program, we still have to define the table itself:

**TABLE DS 20CL29**

To make use of the **DSECT**:

**LA 3,TABLE**  GET ADDR OF TABLE AS USUAL

**USING TABLNTRY,3**

The USING statement connects the register and the set of labels in the DSECT. It tells the computer that R3 contains the address of an area whose contents are described by the DSECT. In other words, it tells the computer to place the labels, defined by the specified DSECT, at the address contained in R3. It will not change the contents of R3, it is programmer's responsibility to make sure that register contains proper address. After the USING statement, the DSECT labels can be freely used instead of using explicit addresses.

**MVC TABLNAME,CARDNAME**

**MVC TABLSSN,CARDSSN**

instead of

**MVC 0(20,3),CARDNAME**

**MVC 20(9,3),CARDSSN**

When we are done processing the current entry, simply set R3 to the address of the next one:

**LA 3,TABLNEXT**

Now the labels in the **DSECT** refer to the second entry in the table.

Be careful where R3 points. Make sure it always points into the table. Strange results can occur if the expected value is not in the register. Also, be sure you initialize the register before you use it.

When you are finished with the **DSECT**, **DROP** the registers.

**DROP R1, R2, ...Rn.**

If you neglect to **DROP** the registers, you may get very unusual problems later in your program.

For example, if you have

MAIN CSECT

....

BUILD CSECT

USING TABLNTRY,9

....

REPORT CSECT

USING TABLNTRY,4

You want the labels in BUILD to be assembled with R9 and the labels in REPORT with R4. If you don't DROP R9 after BUILD, this won't happen. In CSCI 464, you will learn all the rules that are used to decide which base register to use, if there are two possible regs. Suffice to say for now, the assembler picks the higher number register, which is R9. If R9 had been dropped, this wouldn't be the case.

In REPORT, when your program is run, R4 contains the address of the table. Who knows what R9 contains. Addresses are resolved using the contents of R9. What exactly happens is dependent on the contents of R9.

***Advantages***

1) Readability of your program is improved

2) Easier to alter your program

If there is a change in the format of your storage, simply change the DSECT. There's no need to change a bunch of displacements.

LA 3,TABLE

LA 4,INPUT

USING TABLNTRY,3

USING INPUT,4

XREAD INPUT,80

DO1 BM ENDDO1

MVC TABLNAME,CARDNAME

MVC TABLSSN,CARDSSN

LA 3,TABLNEXT

XREAD INPUT,80

B DO1

ENDDO1 DS 0H

***Encoding DSECT Labels***

**MVC TABLSSN,CARDSSN -> D2 08 3 014 4 014**

! !

! !---> Displacement in

! DSECT

!------> Base specified by you

**LA 5,TABLSSN -> 41 50 3014**

# Chapter 11 - Macro Instructions & Conditional Assembly

## The IBM Macro Language

* Extension of basic assembler language.
* Means of generating (at assembly time) a commonly used set of instructions as many times as needed.
* The necessary statements are included in the macro. So when the macro is called (coded), it will produce the necessary statement / statements. It can be called any number of times for generating the code. For example, a macro for generating standard linkage can be written and later it can be called in all the routines, thus saving a lot of typing.
* Code only the macro name when the statements are to be generated.
* Also it must be clear that macros do NOT save any space, they just save typing and thus typing errors.
* Macros are processed prior to assembly, so a macro loop is different than a loop in assembler code.

* A macro is a separate entities from the programmers code. All macros should be coded in the beginning before the first CSECT.
* When the macro is invoked, the instructions are generated as if you had typed them in the routine itself.
* All the macro generated code has a '+' in the first column.

**Advantages:**

* + 1. Simplify the coding of programs
    2. Reduce number of coding errors

**The Difference Between Macros and Subroutines**

**Subroutines**:

* + 1. Branch out of main logic into a separate logic.
    2. Performed identically each time it is executed.
    3. Subroutines are invoked during execution time.

**Macros**:

* + 1. Generates assembler code instructions where it is coded.
    2. Depending on how it is coded, different or identical instructions may be generated.
    3. Macros are invoked during assembly time.

## Macro Definition

All code in a macro lies between **MACRO** and **MEND** statements.

MACRO

MACNAME <--- Prototype statement (macro name and

parameter declarations) is used to invoke the macro.

....

....

MEND

Macros can be defined within a program (called a **source macro definition**) or may be defined in a library of macros (where it is a **library macro definition**). Macros usually start off as source macros in a single program. They may be later placed in a macro library so that other assembler programs may have access to them. **EQUREGS** is an example of this.

In ASSIST, all source macro definitions must appear before the first CSECT/DSECT. You may not use **EJECT**, **TITLE**, **SPACE** statements in a macro to space the macro code because the macro code is processed pre-assembly. So if you have any of these statements in a macro then the affect of these statements will be seen when the macro is invoked and not within the macro itself. Remember the macro statements are processed before the program is assembled.

## Format of a Macro Definition

**MACRO** - The header statement must be first statement

**anyname** - Prototype statement

.... - Body of the Macro

.... contains Assembler code and also macro processing statements.

**MEND** - Trailer statement (provides exit)

- It must be the last statement

- It may have a sequence symbol in column 1

- You cannot have more than one **MEND** statement within a macro

***Prototype Statement***

* The name field is required for defining and invoking a macro.
* It must be right after the header statement (MACRO).

Format:

label or symbol 0-many parameters

blank

!

!\_\_\_ if used must have variable symbol e.g. &LABEL

Parameters can be positional or keyword parameters

**MACRO** <-- Header statement

**&LABEL EXMPL1 &A,&B,&C** <-- Name field

This is what could be typed inside the program.

**CALL1 EXMPL1 VAL1,VAL2,VAL3**

By this invoking statement,

* &A will be replaced by VAL1
* &B will be replaced by VAL2
* &C will be replaced by VAL3

throughout the macro expansion.

In the above example position of the parameter is important. When the macro is invoked the first value coded is assigned to the first parameter, second value to second parameter and so on so forth. So you must be very careful while coding the values, a mismatch will give you unexpected results. Otherwise, use **KEYWORD** parameters. If a combination of positional and keyword parameters are used, all the positional parameters must be coded before coding the keyword parameters.

For example, if the prototype statement is coded as follows:

**&LABEL** **EXMPL2 &D,&E,&A=,&B=,&C=20**

and it is invoked by a routine as follows:

**CALL2 EXMPL2 VAL3,VAL4,B=VAL1,A=(R5,R7),C=,**

D will get VAL3, E will get VAL4, B will get VAL1, A will get (R5,R7) and C will get null value.

In this example position of A, B, and C is not important because parameter name is also coded along with the value. D is the first positional parameter and E is the second positional parameter. So when the values are assigned D will get VAL3 and E will get VAL4.

**NOTE**: The rules are a little different if you are using a combination of keyword and positional parameters in a prototype statement than when you are using actual High Level Assembler (not ASSIST). Talk to your instructor if you want to know the differences.

If there is nothing after the '**=**', that means a null value is assigned to that parameter. For positional parameters, a null value is assigned to a parameter by coding nothing for the parameter value in the calling statement.

**Continuation**: When a macro is invoked during assembly, if the prototype statement exceeds one line then break it at a comma, put some character (usually a 'x') in column 72 and start on next line in column 16.

**Body of Macro**: Body of macro consists of the following:

* + - 1. Assembler instructions which are generated when the macro is expanded.
* Conditional assembly instructions used for generating different code depending on different requirements. These statements are not generated when macro is expanded.
* Macro inner instructions
* macro processing instructions, not generated with macro expansion.
* **MNOTE** instruction for generating error/warning messages
* **MEXIT** instruction for terminating the macro processing in middle.

* **Comments**

**.\*** - not generated when macro expands

**\*** - generated with macro expansion

## MNOTE and MEXIT

Frequently, it is necessary to generate messages during macro processing. Some times messages are just informative and at other times they could be error messages and must be considered as assembly time errors.

* **MNOTE** can be used to generate messages along with an optional severity code.
  + Severity code determines whether the generated message is an error or just informative.

* + Message with severity code of 4 or higher is considered to be an error message.

**MNOTE** 4,'message' ==> 4,message

**MNOTE**  'message' ==> message

**MNOTE** 12,'message' ==> message

* **MEXIT** provides exit point from inside the macro body.
  + MEXIT must be used whenever macro processing has to be stopped in the middle.
* All the following examples will incorporate the use of MNOTE & MEXIT.

***Conditional Assembly***

* Provides a way to alter the sequence in which source program statements are processed by the assembler.
* Can branch within the macro to generate different code depending on different requirements and requests.
* Conditional assembly statements are not generated when macro expands.
* Labels used in conditional assembly instructions are referred to as Sequence Symbols.

# Chapter 12 – Working on the Marist Mainframe

## 12.1 Allocating Your Assignment PDSE

To program on the mainframe, it is necessary to first create, or allocate, a data set known as a partitioned data set extended (PDSE) where your code will be stored. What follows are the steps necessary to allocate your assignment PDSE that you will use all semester. During the first week of class, your instructor will show you how to back up this important data set so that you have something to go back to if you delete it.

First, download a copy of Tom Brennan's Vista TN3270 at:

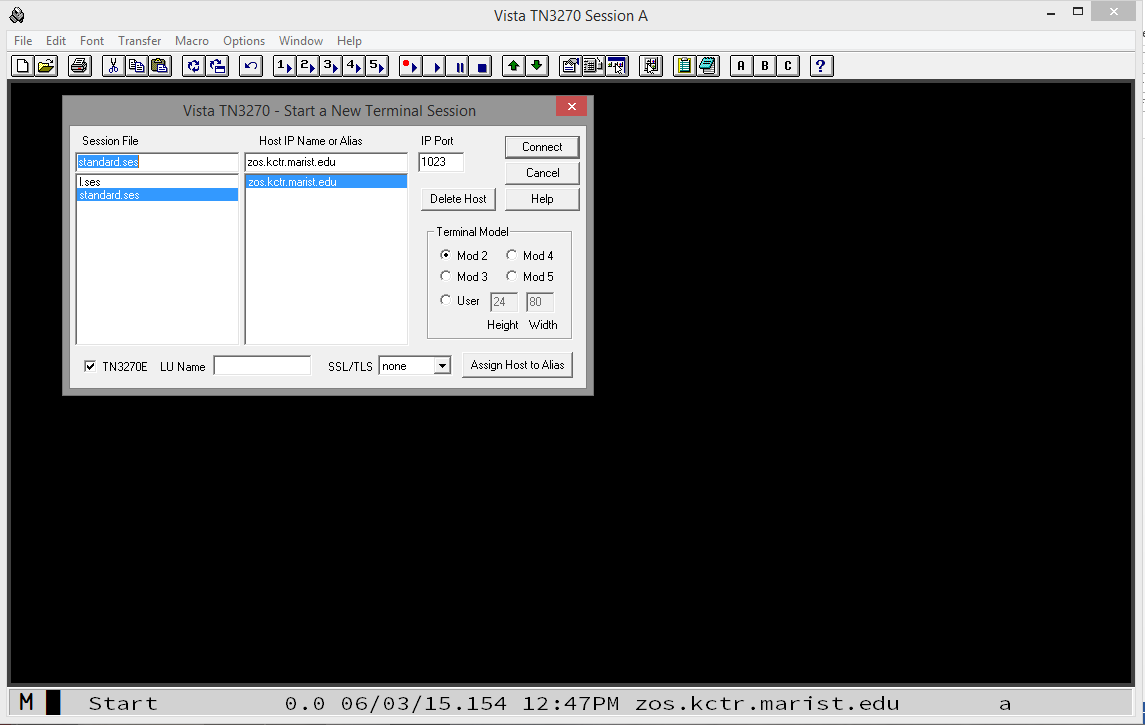
<http://www.tombrennansoftware.com/download.html>

For those of you wishing to use Adobe's *Source Code Pro* fonts, download the V2.00 Preliminary Test Version at:

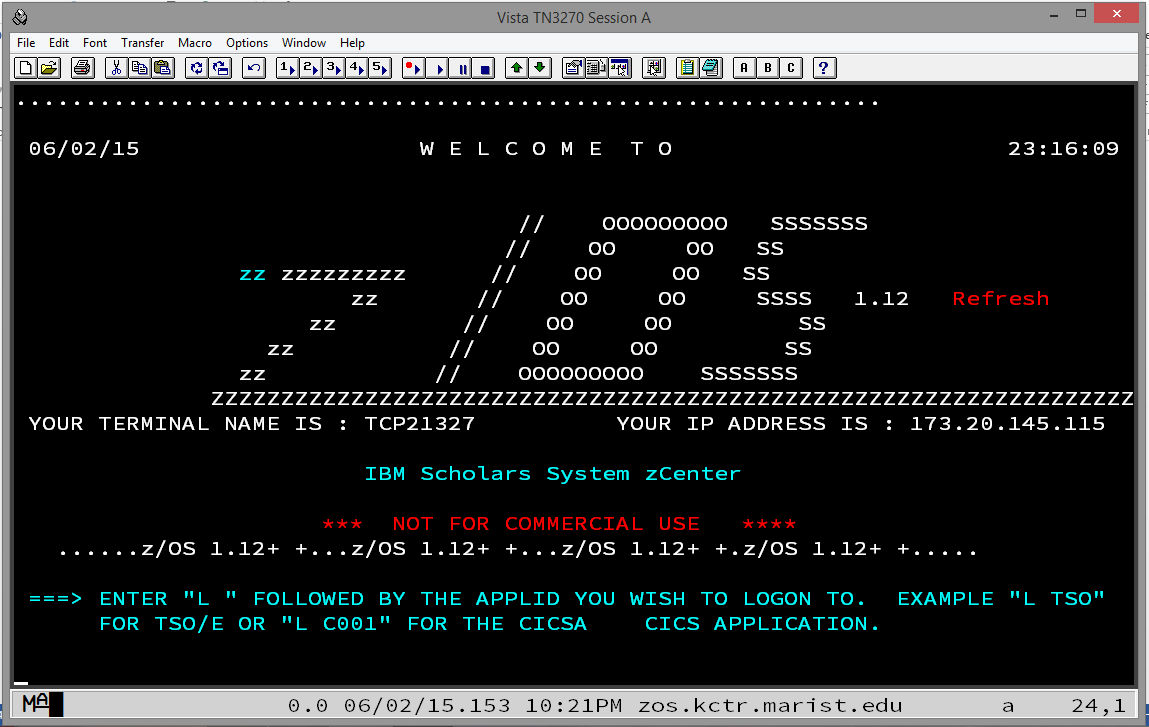
[http://www.tombrennansoftware .com/v200/](http://www.tombrennansoftware.com/v200/)

With either version, you will have to use the current Vista TN3270 registration name and code so that your copy will not expire after the software's trial period. This can be found in Blackboard's Course Documents.

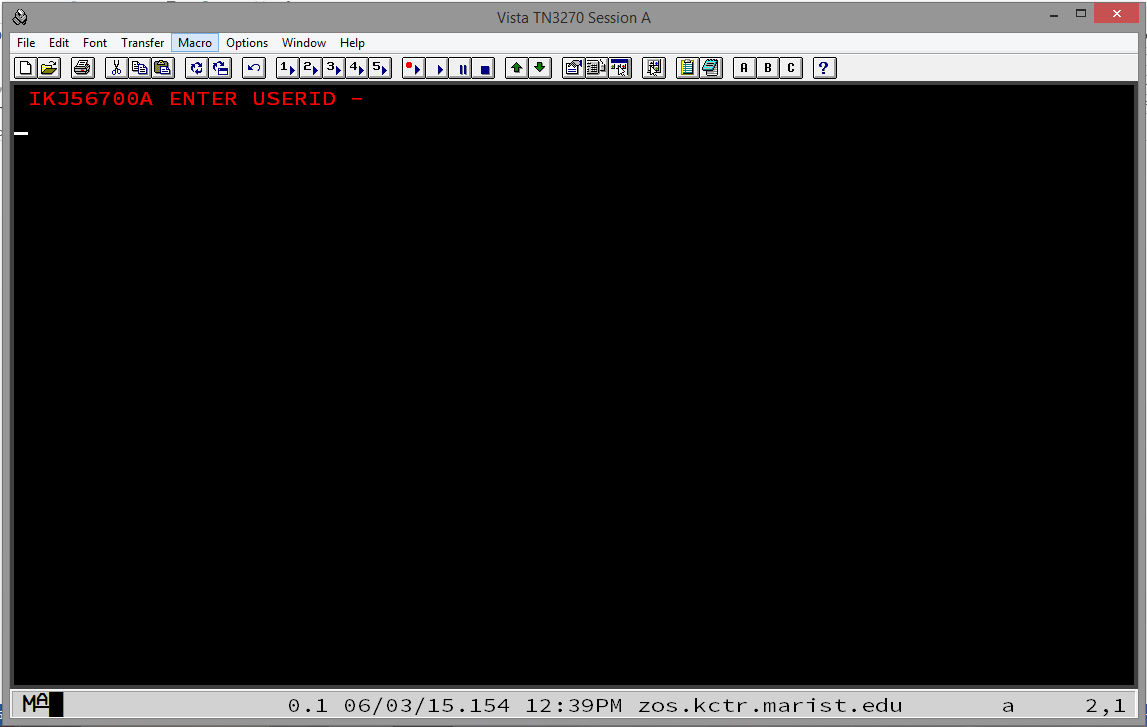
Each time you run Vista TN3270, you will be presented the following:



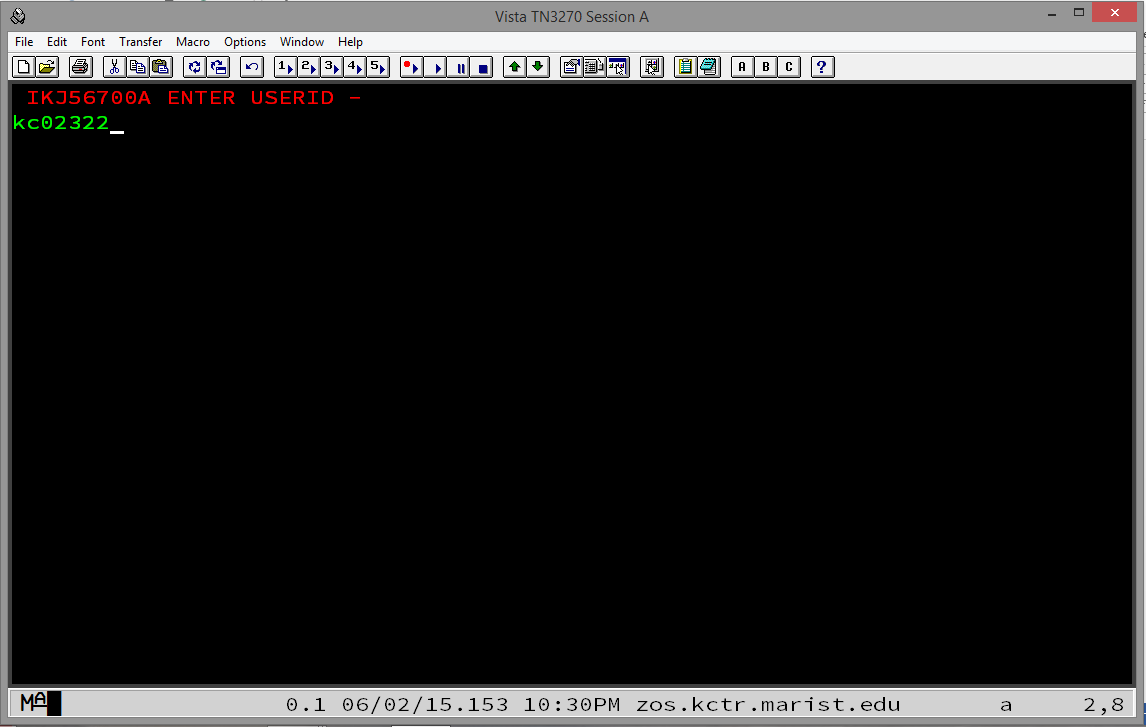
To connect, enter Host IP Name zos.kctr.marist.edu and Port 1023 and click Connect. You will be presented the following:



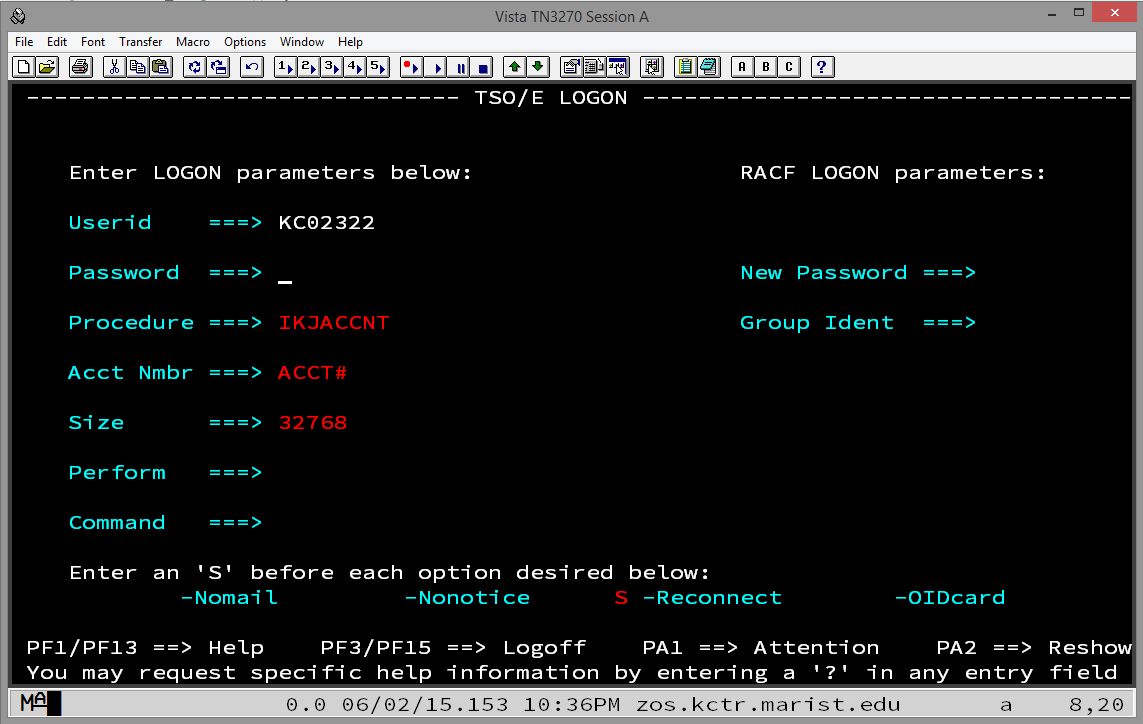
Type the letter L and press Enter. You will be presented the following:



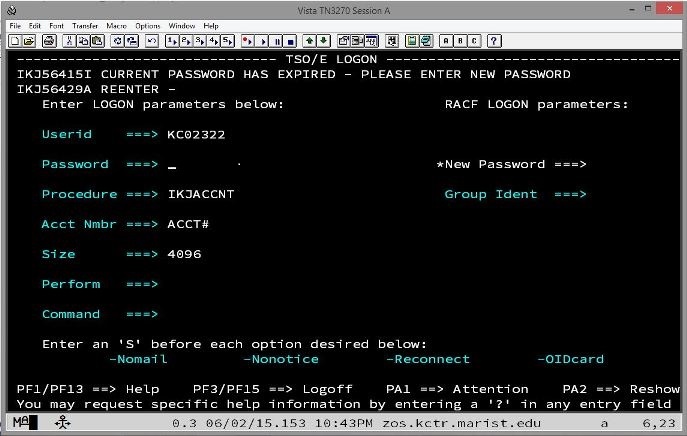
Enter **your** assigned KC-ID. The KC-ID KC02322 is shown in the following as an example:



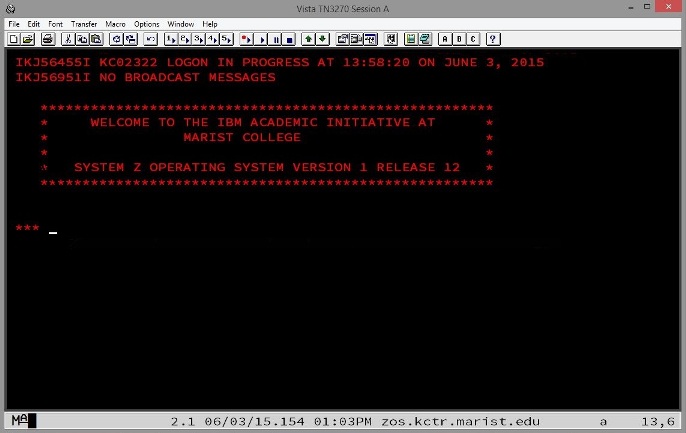
Press Enter and you will be presented the following:



The first time you sign on your password is the same as your KC-ID. Enter it carefully and press Enter. You will be presented the following:

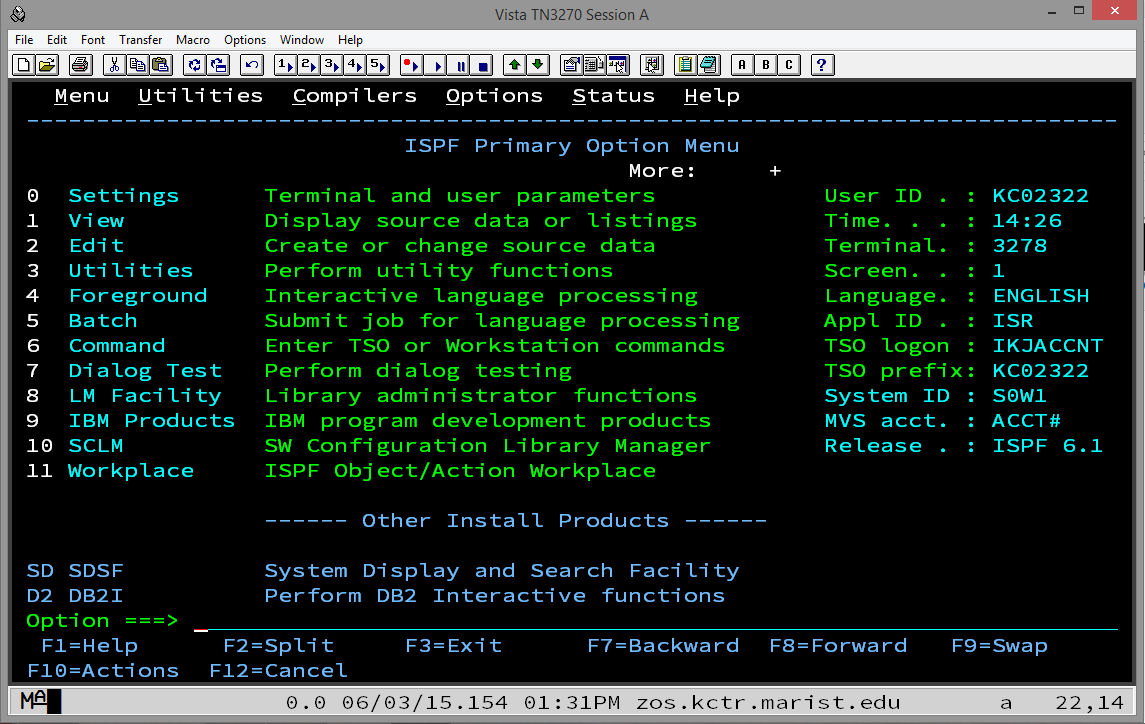


Enter a new password twice and you will be presented the following:



Note that your new password should be eight alphanumeric characters and should begin with a letter. It is NOT case sensitive. Again, you will have to enter it twice. Do so carefully. If you enter an eight-character password, the cursor will automatically tab to where you need to re-enter it to confirm it.

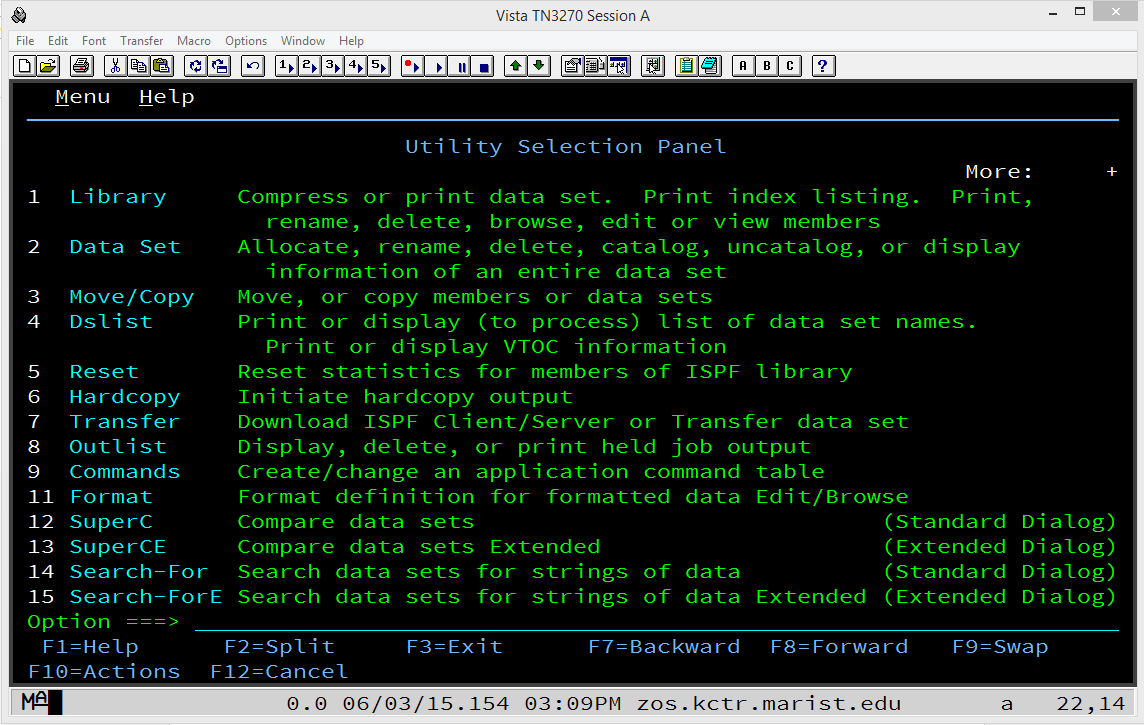
Press Enter again and you will be presented the following, the ISPF (Interactive System Productivity Facility) Primary Option Menu:



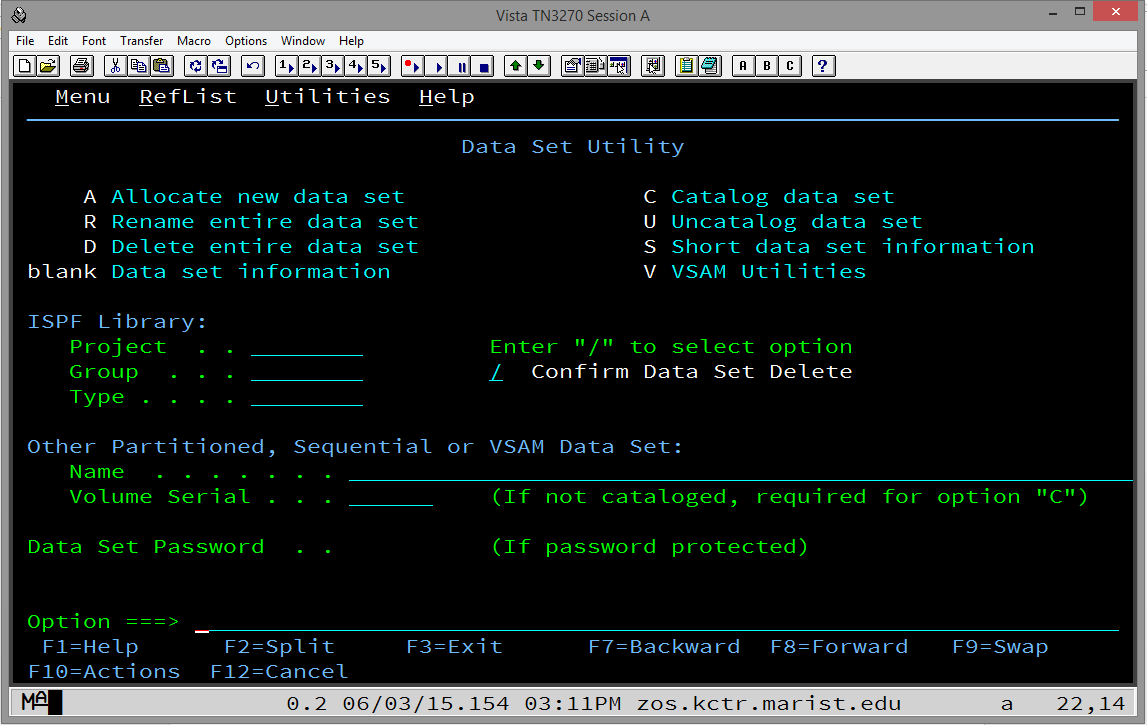
Although this looks complicated, you will only use Options 2 (Edit), 3 (Utilities), and SD (SDSF = System Display and Search Facility) for the remainder of the semester and, if you continue in mainframe studies, in CSCI 465/565 – Enterprise Application Environments.

A PDSE is sometimes called a "library." This is only because a PDSE, unlike a sequential data set, or "flat file," is separated into different members which are, in themselves, sequential files. Each of these members of a PDSE is somewhat like a book on a bookshelf in a library, hence the alias "library." So, a PDSE is a collection of members. You will create a new member for each assignment or other programming exercise this semester. The first one you will create will be named ASSIGN1 but first we need a PDSE to which we will add the member ASSIGN1.

To allocate a PDSE, enter 3 for Utilities and press Enter. You will be presented the following, the Utility Selection Panel:



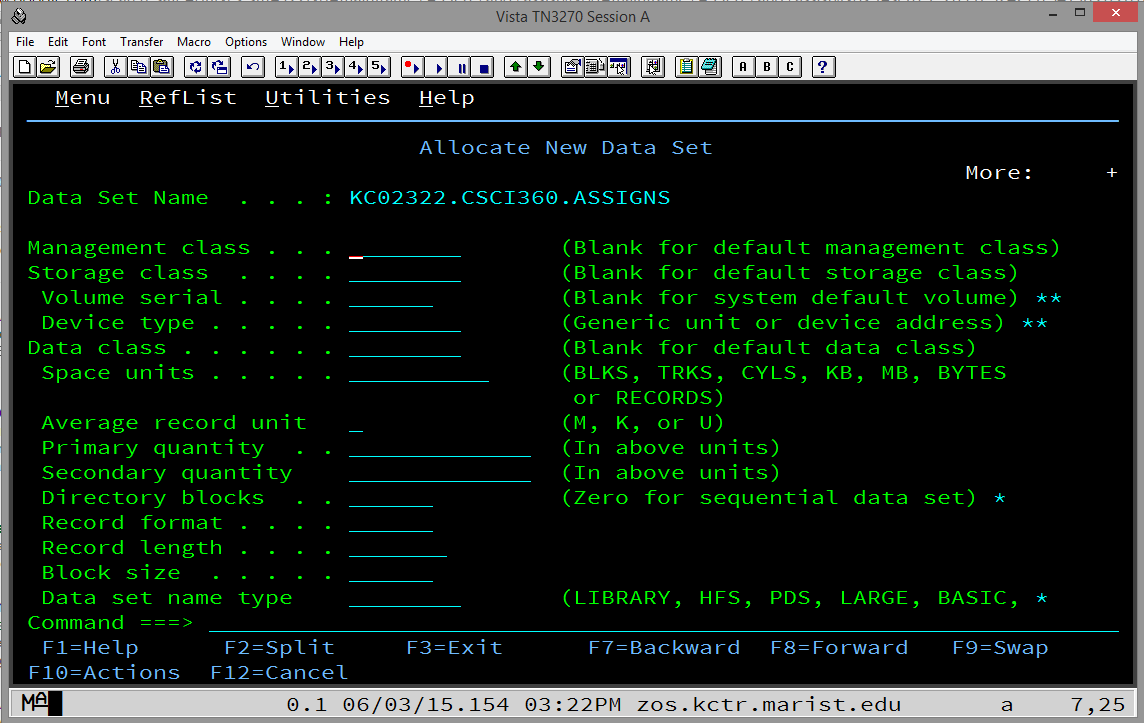
On this screen, enter 2 (Data Set) and press Enter. The first time you enter, you will be presented the following, the Data Set Utility panel:



Move the cursor by tabbing or with your mouse to the line to the right of Project under ISPF Library: Type your KC-ID. Tab again to the line to the right of Group and type csci360. Tab again to the line to the right of Type and type the word assigns. Tab four times or move the cursor with your mouse to the Option ===> line at the bottom. Enter the letter a (for allocate) and press Enter.

Note that you can enter everything on the ISPF screens as lower case letters and the system will automatically change them to upper case. (This is not necessarily true for the Edit option that will be covered below.)

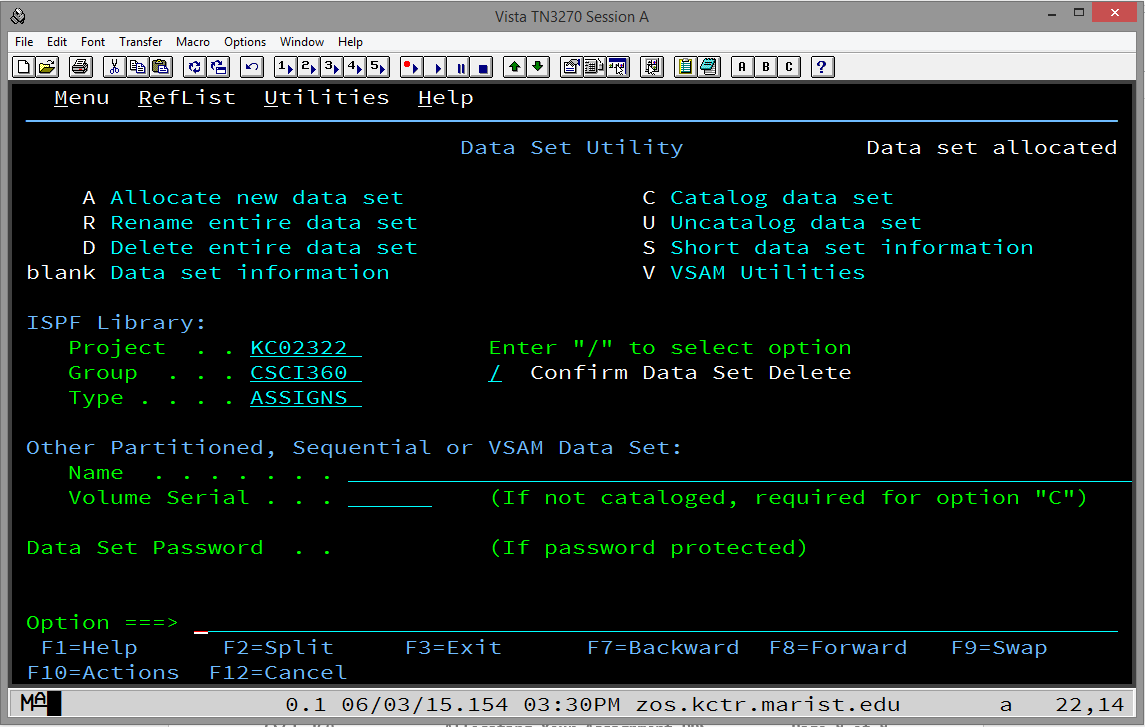
After pressing Enter, you will be presented the following, the Allocate New Data Set panel:



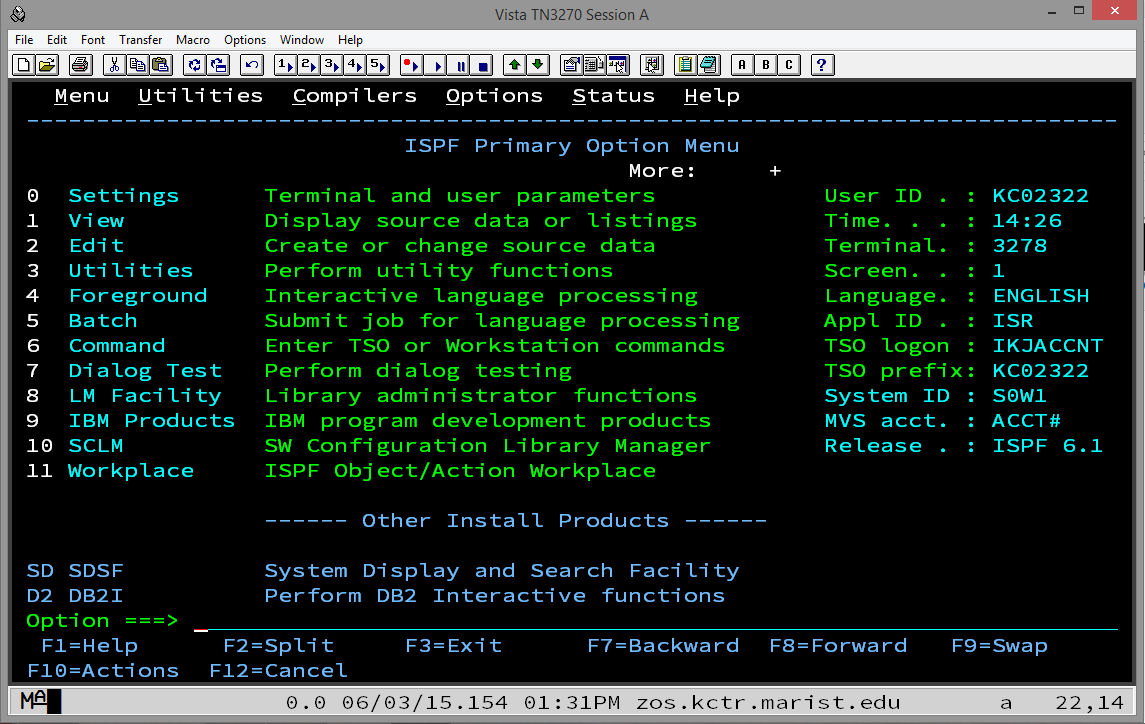
The first time you enter this panel, nothing will be pre-filled. But, the next time, it will be pre-filled with the parameters from the previous time you used this panel, i.e., the last time you successfully allocated a data set.

First, tab or use your mouse to move the cursor to the line to the right of Space units and enter CYLS for cylinders. Tab twice and enter 1 for Primary Quantity. Tab again and enter 1 for Secondary quantity. Tab again and enter 10 for Directory blocks. Tab again and enter FB for Record format. Tab again and enter 80 for Record length. Tab again and enter 880 for Block size. Tab again and enter LIBRARY for Data set name type.

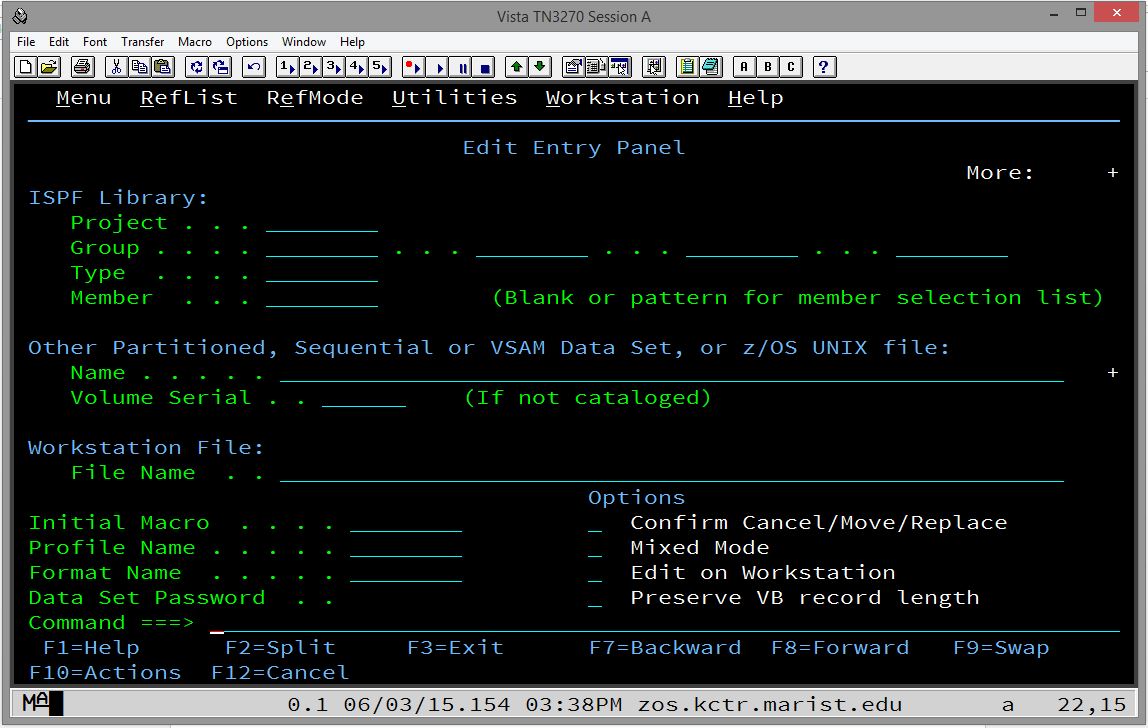
After pressing Enter and, if the data set does not already exist (and it shouldn't!), you will be presented the Data Set Utility panel again with the message "Data set allocated" in white lettering in the upper right hand corner of the panel:



This indicates success! At this point, press F3 twice and you will once again be presented the following:



Now it is time to create and edit your first PDSE member. Enter 2 (Edit) at the Option ===> line and press Enter and you will be presented the following, the Edit Entry Panel:



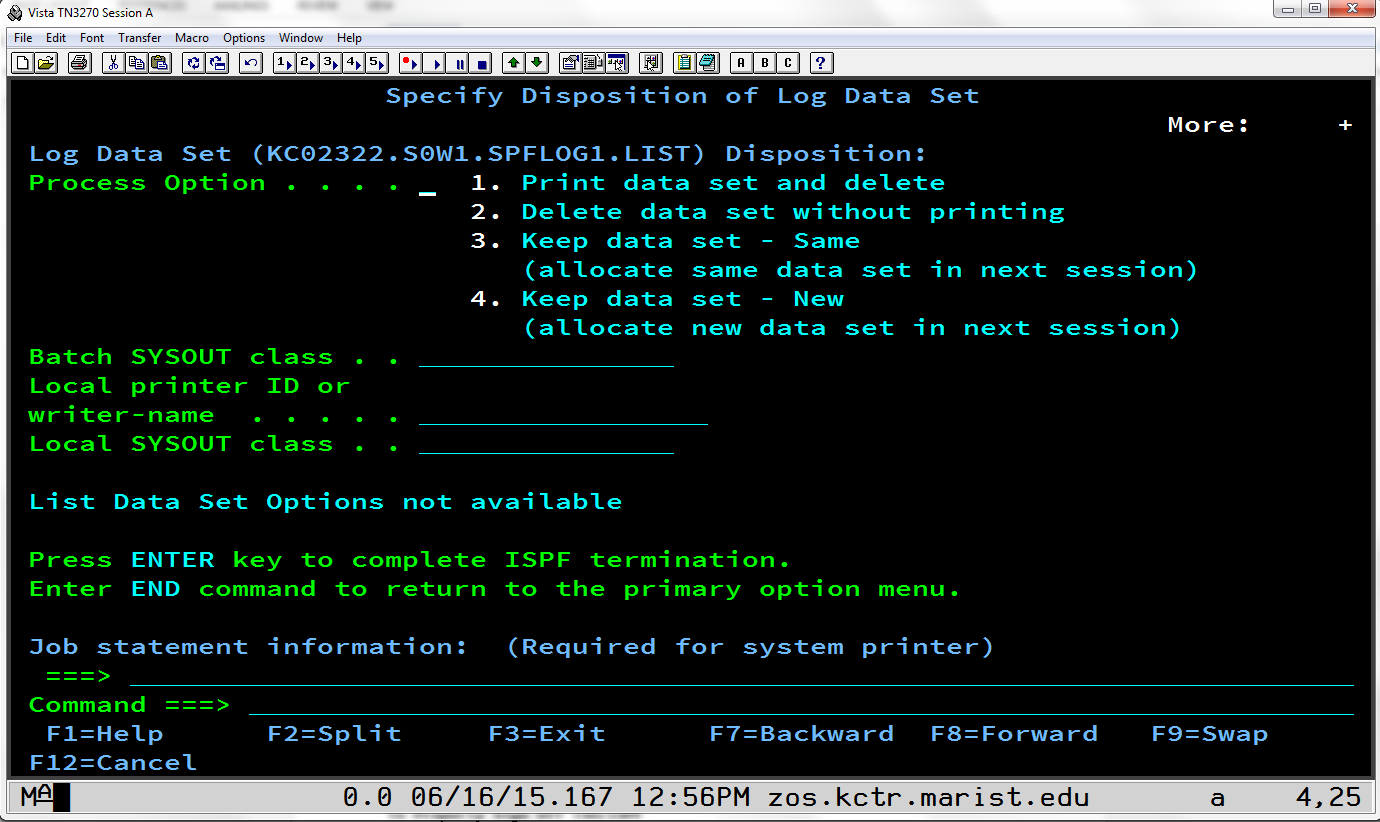
The first time you enter this panel, you will need to fill in some fields that will be pre-filled the next time you come back to it.

Tab a few times or move your cursor with your mouse to the line to the right of Project and enter your KC-ID. Tab once again and enter CSCI360 on the line just to the right of Group. Ignore the second, third, and fourth blanks and tab to the line just to the right of Type and enter ASSIGNS.

Finally, tab once more to the line just to the right of Member. It is here that you will enter the name of the member that you want to create and begin editing or you can enter the name of an existing member that you want to return to and continue editing.

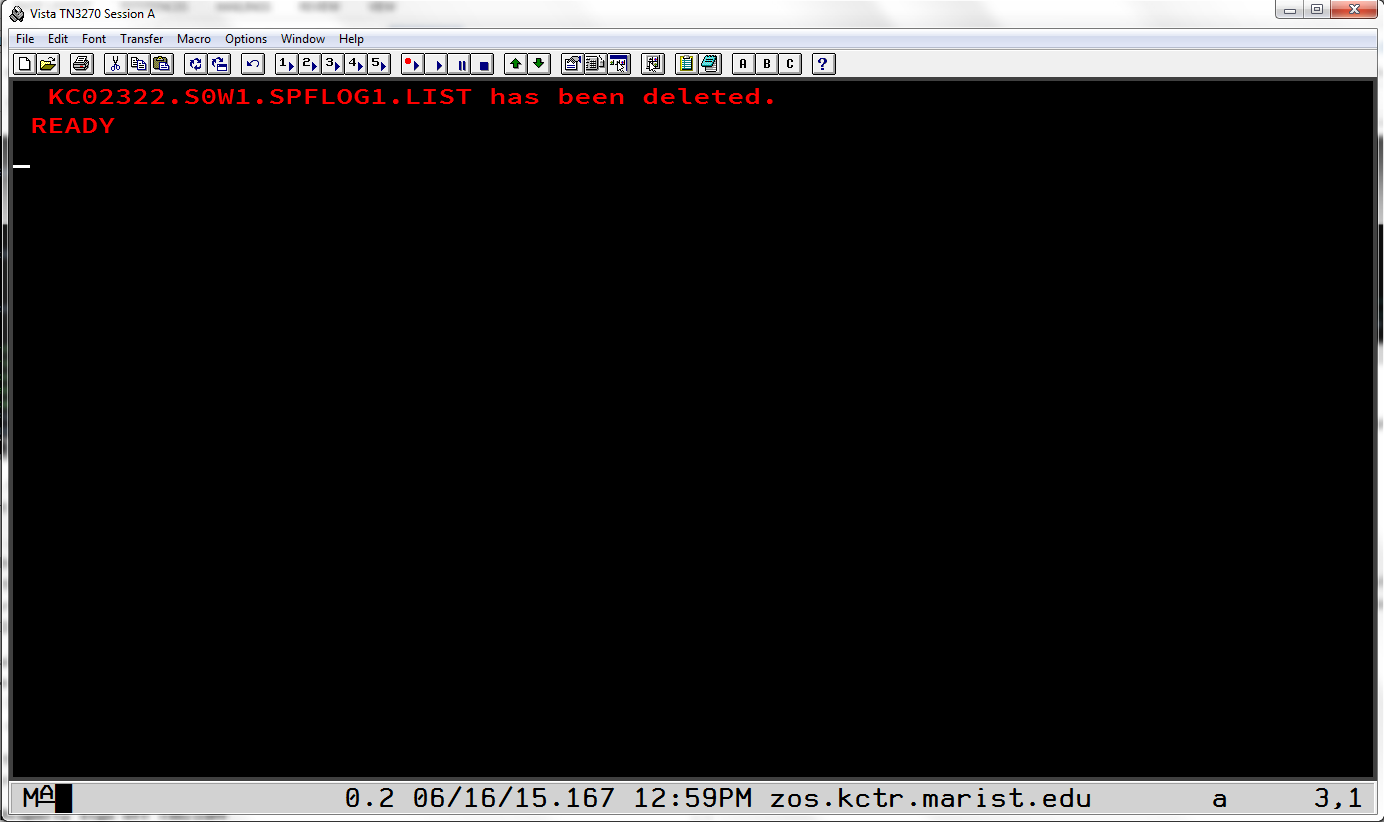
At this point, please refer to the document named Editing in ISPF to learn how to edit a new or existing member of your PDSE.

To properly sign off TSO/ISPF, press F3 while in the ISPF Primary Option Menu. If you have made changes while signed on, you will be presented the following screen:



Type the number 2 and press Enter.

You will then be presented a screen with a red-lettered READY displayed as follows:



Type the word logoff or LOGOFF and press Enter and you will now be logged off Marist.

## 12.2 Editing in ISPF

It is first important to know that names of entities on the mainframe can have 1 to 8 characters. They can only contain letters A-Z (upper case only), digits 0-9, and international characters $, # and @ (dollar sign, pound sign/hash tag, and at sign). They can only begin with a letter or one of the three international characters. Please do NOT use the international characters in this class (although we will discuss using the $ later).

It is also useful while editing in ISPF to press Caps Lock as 99.9% of everything you type in ISPF will need to be in capital letters.

Please note that the screen shots below display the editing of a member named ASSIGN0. This is not your first assignment but rather an example of how to edit using ISPF.

A) To edit a PDS member:

From the ISPF Primary Option Menu select Option 2 (Edit). Then, on the Edit Entry Panel, type the name of the new member you want want to create and press Enter.

Or to begin editing an already existing member in a PDS:

Follow the same instructions from above or you can press Enter in the Edit Entry Panel and move the cursor to the dot across from the name of the member you wish to edit and type either s, S, e, or E and press Enter.

B) To save a member you are editing:

Type the word save or SAVE anywhere on the command line (Command ===>).

It is recommended that you save the member often as there is no autosave in ISPF!

Please note that items C through G below are usable on most panels in ISPF.

C) To exit a panel:

Use F3 to exit and back up one level.

Be sure that, if you are editing a PDS member, that you save it first!

D) To scroll up and down:

Use F7 for scrolling up and F8 for scrolling down.

E) To scroll left or right:

Use F10 to scroll left and F11 to scroll right.

F) To go to the top or bottom:

Type the word top or TOP on the command line to go to the top of the panel or type the word bot or BOT or bottom or BOTTOM to go to the end.

G) Setting the Scroll ===>:

It is strongly suggested that you change Scroll ===> PAGE to Scroll ===> CSR on every panel that you can in ISPF. ISPF will retain this setting if you exit the panel normally. See lower right hand corner of Figure 1.

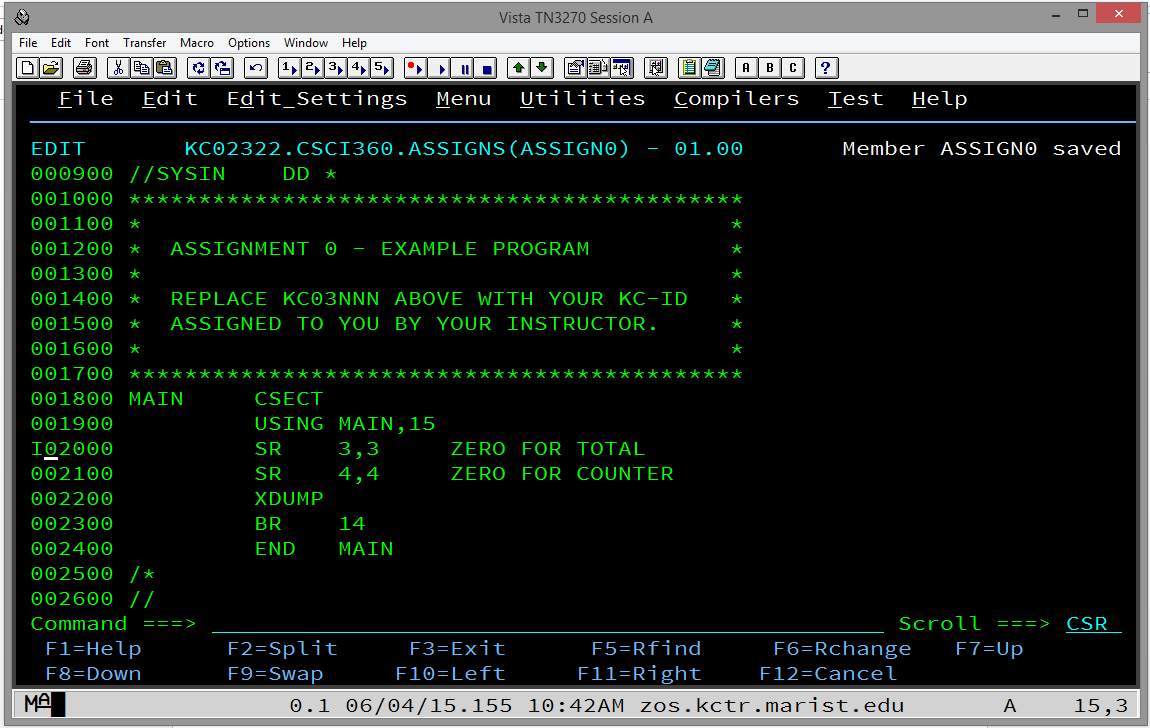


Figure 1.

H) To insert a line while editing:

Move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type the letter i or I. See Figure 1. Press Enter to have the line inserted.

See Figure 2.

I) To insert multiple lines while editing:

Do the above but follow the letter i or I with an integer between 2 and n. It will insert the number of lines you have requested but it will not scroll to show all of your inserted lines. It will fit as many on the panel as it can depending on where you began inserting the lines.

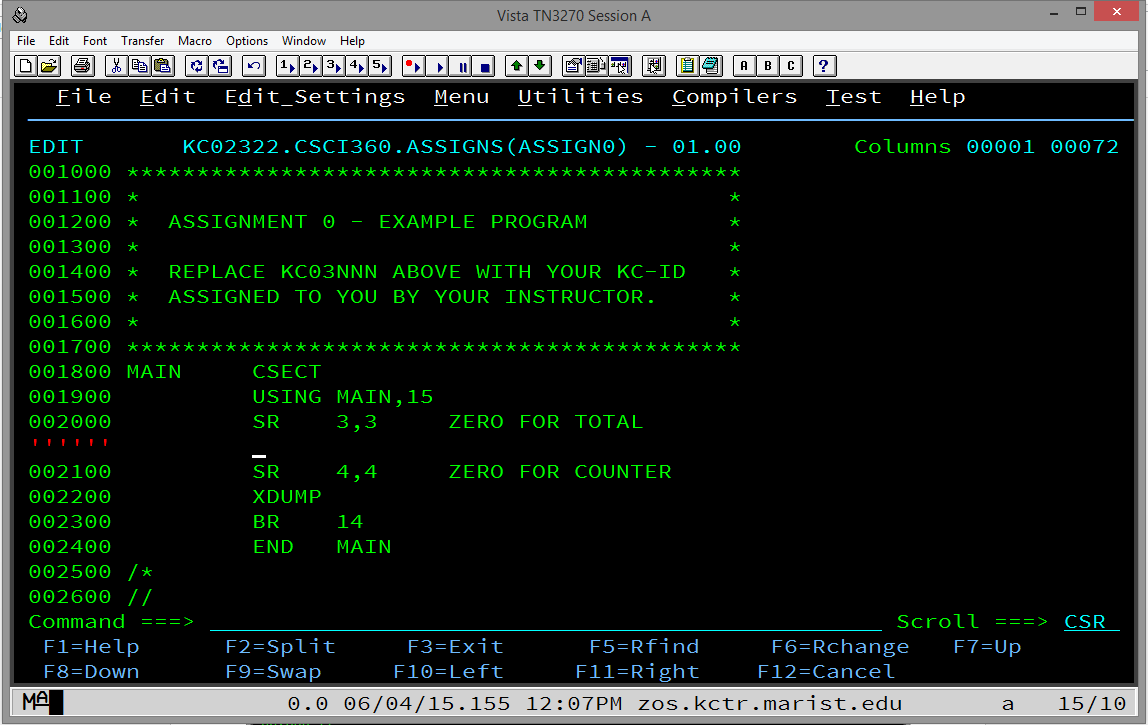


Figure 2.

Note that, if you hit enter at this point, any lines that do not at least have a space (one press of the space bar on your keyboard) will simply be deleted.

Also, if you insert a line and type something on it, a new blank line with be automatically inserted if you press Enter.

J) To delete a line while editing:

Move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type the letter d or D. See Figure 3. Press Enter to have the line deleted.

K) To move a line while editing:

Move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type the letter m or M. Then, move the cursor with the tab key or your mouse to where you want the line moved and type either a or A for inserting the line you are moving after the line where your cursor is or type either b or B for inserting the line you are moving before the line where your cursor is.

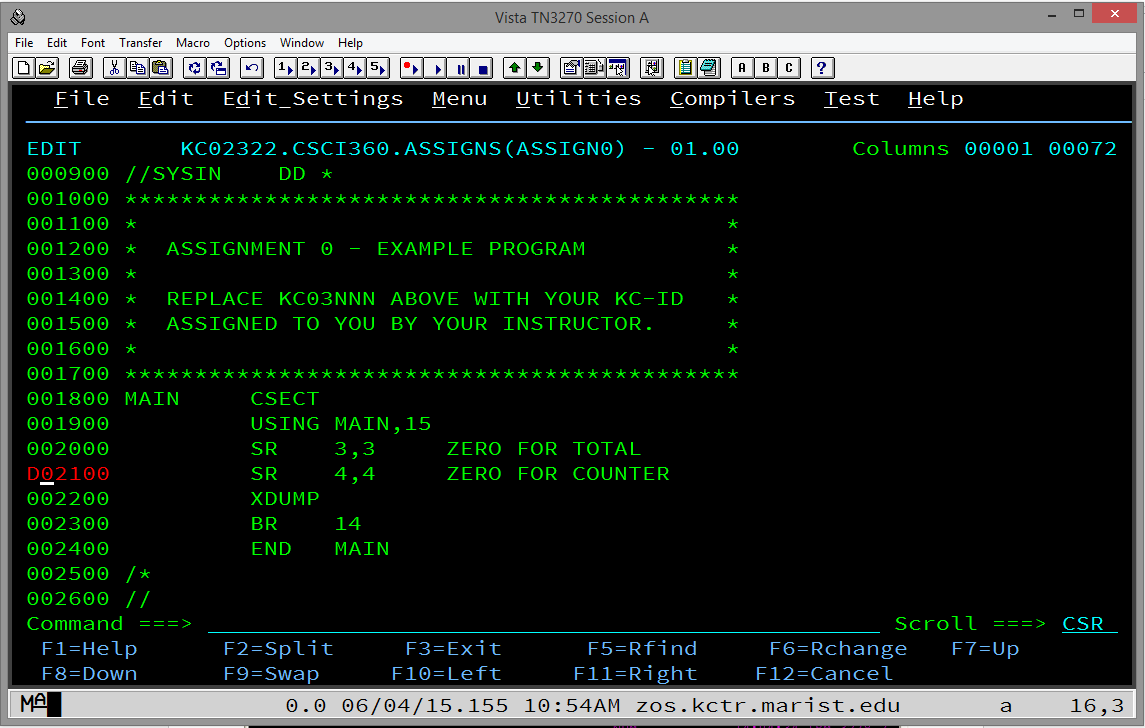


Figure 3.

Note that you can scroll up or down with F7 or F8, respectively, and also use the command line top or bot to go to the top or bottom of the panel while the move is still active.

Also note that, when you start a move, you must complete it before you can go on editing. In other words, if you change your mind, you will still have to move the line but you can then delete it if necessary.

L) To copy a line while enditing:

Move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type the letter c or C. Then, move the cursor with the tab key or your mouse to where you want the line copied and type either a or A for inserting the line you are moving after the line where your cursor is or type either b or B for inserting the line you are moving before the line where your cursor is.

M) Deleting, moving or copying blocks, or multiple lines while editing:

To delete a block of contiguous lines, type dd or DD on the first line you want to delete and type dd or DD on the last line you want to delete. These two lines and every line in between will be deleted.

To move a block, use mm or MM on both the first line and the last line you want to move and the a or A for after or b or B for before as in item K above. The block of contiguous lines will be deleted from its   
original place.

To copy a block, use cc or CC on both the first line and the last line you want to copy and the a or A for after or b or B for before as in item K above. The block of contiguous lines will remain in its original place and a copy will be inserted where you indicated it to be inserted.

N) To split a line of text:

To move the end of a line to the next line, or split the text, move the cursor with the tab key or your mouse to the line numbers on the left hand side of the screen and anywhere within those 6 digits type ts or TS for 'text split'. Then, before you press Enter, move your cursor to the character where you want to begin the split. See Figure 4.

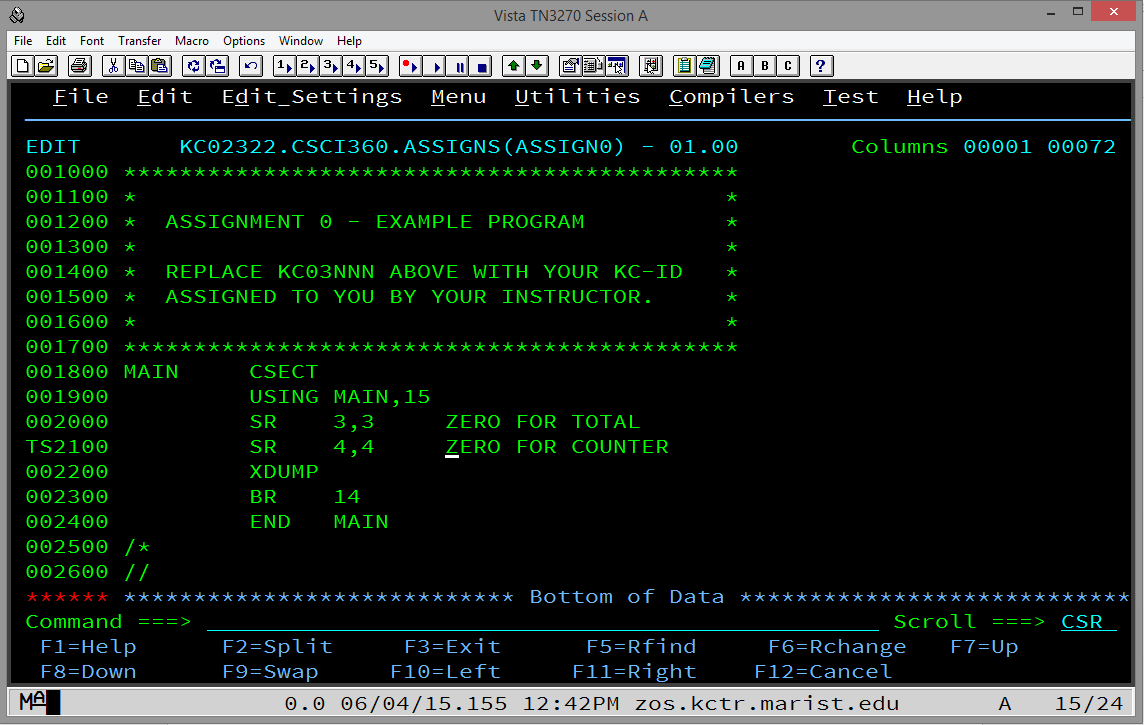


Figure 4.

Now, press Enter. Everything from the character you indicated as the beginning of the split will be pushed and inserted two lines down with a new line inserted in between. See Figure 5. Simply press Enter again and the new blank line will be removed.

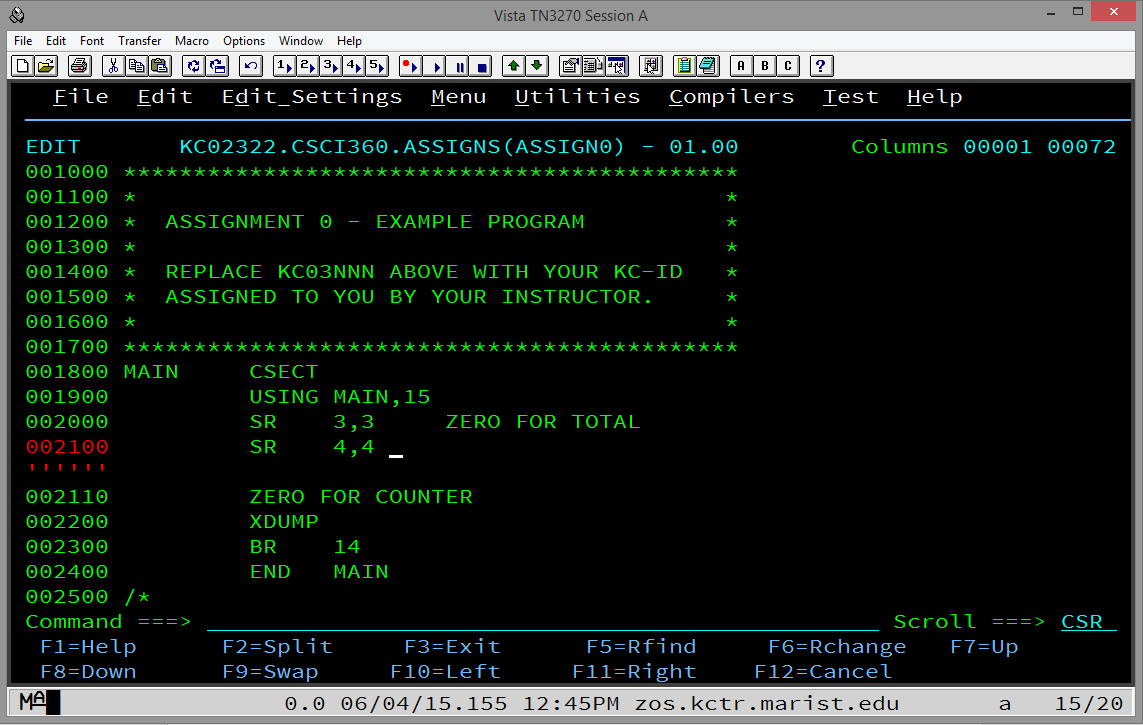


Figure 5.

O) To collapse (hide) lines while editing:

Because it is sometimes scrolling can be frustrating while editing, ISPF allows you to collapse lines. To collapse one or more lines, move the cursor with the tab key or your mouse to the line numbers on the   
left hand side of the screen and anywhere within those 6 digits type x or X to collapse one line or xn or Xn to collapse n lines.

To collapse a block, type xx or XX on the first line you want to collapse and scroll to the last line you want to collapse and type xx or XX. Press Enter and a dashed line will appear telling you how many   
lines are collapsed. See Figure 6.

If you collapse some lines but not enough, you can either type x or X and continue doing it but it is more efficient to move your cursor to the dashed line and type x or X or xn or Xn to collapse another line or another n lines, respectively.

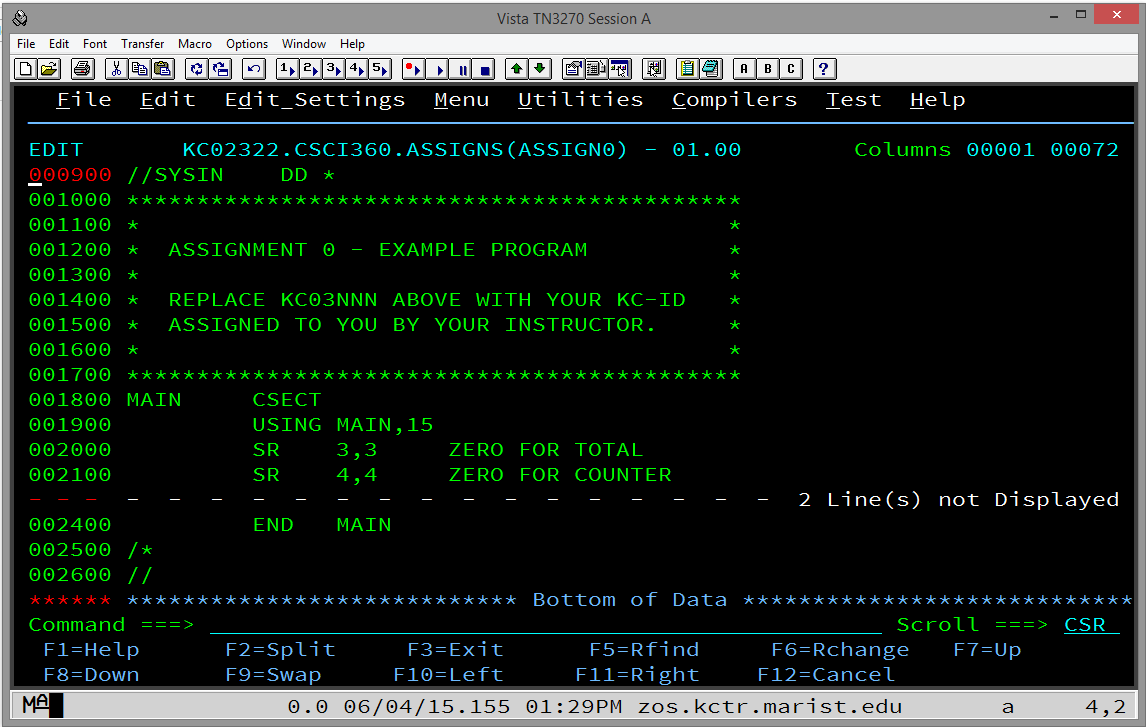


Figure 6.

P) To uncollapse (reveal) lines while editing:

Type res or RES or reset or RESET on the command line and press enter and all of the lines will be uncollapsed or revealed.

To uncollapse or reveal some lines but not others, go to the dashed line and type f or F to reveal the first collapsed line or fn or Fn to reveal the first n collapsed lines. You can also type l or L to reveal the last collapsed line or ln or Ln to reveal the last n collapsed lines.

## 12.3 Submitting and Viewing Results in ISPF

To review output in the output queue in SDSF on TSO/ISPF at Marist, enter SD (for SDSF) from the ISPF Primary Option Menu command line. From the SDSF Primary Option Menu enter ST for status on the command line.  This will display the queue of completed jobs, both successful and unsuccessful.

Note that the first time the user comes to this screen, he or she will need to enter OWNER KC03nnn and press enter (KC03nnn represents the user's KC-ID).  This will then only display the user's jobs in the queue.

Be sure not to let these completed jobs pile up. To get rid of jobs in the queue, put a P on the line in the margin just to the left of the job to be purged and press enter.  A P can be entered on multiple jobs at once but the user may have to press enter a few times to get the jobs to roll off.

By the way, the user can enter SD.ST from the ISPF Primary Option Menu to go directly to the status queue.  If somewhere else within TSO, the user can enter =SD.ST to go directly to the status queue. For example, if editing a PDS member and it has been saved, the user can submit his or her job by typing SUB on the command line. A red message will pop up if a successful submission has been made. Press enter again and then enter =SD.ST and press enter to go to the SDSF Status queue to see the results of the recently submitted job.

Once in SDSF status queue, select the job to be reviewed in the queue by typing a letter S in the margin just to the left of the job. It is important to review at least the first few 'pages' of output.  For example, about 8-9 lines down from the top of the job in the queue for Assignment 0 will be similar to the following:

11.22.38 JOB01444 -STEPNAME   PROCSTEP    RC   EXCP   CONN  TCB  SRB   
11.22.38 JOB01444 -JSTEP01                00      7     1  .00  .00

Notice that there is a line with the stepname (JSTEP01 in this example) and, to the right, there is a RC (return code) of 00.  This is a good sign meaning that that step of your job completed successfully.

Scrolling a little further, you will find the following:

IEF142I KC02322A JSTEP01 - STEP WAS EXECUTED - COND CODE 0000

This is another place to check to be sure that the COND CODE (condition code) of each step is 0000 which indicates the step completed successfully.  Of course, there might still be a logic error of some sort but at least that step ran to a successful conclusion.

To submit a file for grading, use mar\_ftp.exe to get the file down to your laptop or PC.  It is recommended that the user first download the most recent version of mar\_ftp.exe from the Computer Science Department's website.  Put the mar\_ftp.exe file in a folder of its own perhaps in the My Documents/Documents library.  Downloaded files from the queue automatically download into the same folder from which mar\_ftp.exe is running.

Note that mar\_ftp.exe will download two files, one is a .txt file and one is not.  Be sure to open the .txt file and review it carefully from top to bottom before submitting it.  Make sure that nothing is truncated -- especially at the bottom of the file -- and all of the output expected is there.  Submitting a truncated file will earn a 0 so it is very important to be sure it is all there!

Also be sure to keep copies of all of the .txt files that are submitted for assignments.

## 12.4 Using mar\_ftp.exe to Download Marist Output

**Overview**

This document offers one of several ways to print jobs that have been run on the Marist mainframe computer.  It is not meant to be comprehensive.

**Instructions**Download mar\_ftp.exe and save it into a folder of your choice on your PC:  ["Ftp program for fetching run programs from Marist"](http://www.cs.niu.edu/compresource/mar_ftp.exe)

Follow the mar\_ftp.exe instructions to sign onto Marist, fetch the job to be printed, and save it on your PC:  ["Running Instructions"](http://www.cs.niu.edu/compresource/marist_readme.txt)

Although most students will not have to make these changes, once the text file of the Marist job output that is to be printed is saved on your PC, you ***MAY*** have to make the following changes to it using, for example, Notepad or Wordpad:  
  
1)  Go to File ---> Page Setup…  
2)  Change Orientation to Landscape and click on OK.  
3)  Go to Format ---> Font...  
4)  Change Font to Source Code Pro, Courier New, Font style to Regular (which is the default), Size to 9, and click on OK.  (It must be 9 to get 133 bytes to fit across the landscape display line.)

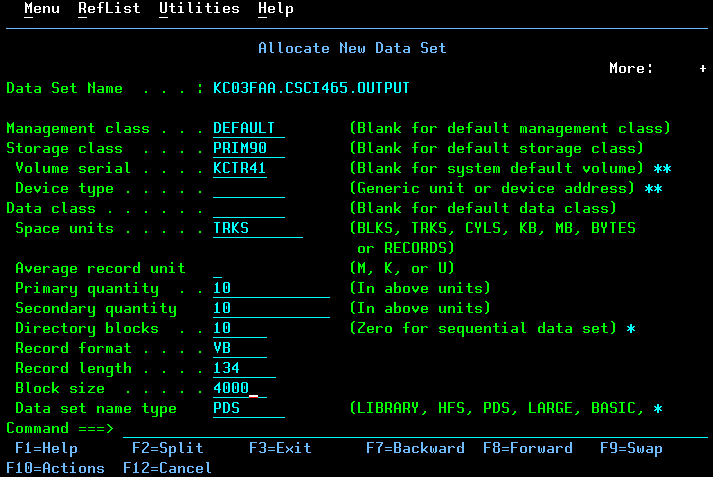
Now the text file is ready to print.  Go to File ---> Print... and print your job.

Note:  There are two files saved on your PC.  Be sure that you make the changes to the .txt file.  Once you have printed it, you can delete both files.

## 12.5 Using FileZilla to Download Marist Output

**Allocate an Output Data Set**

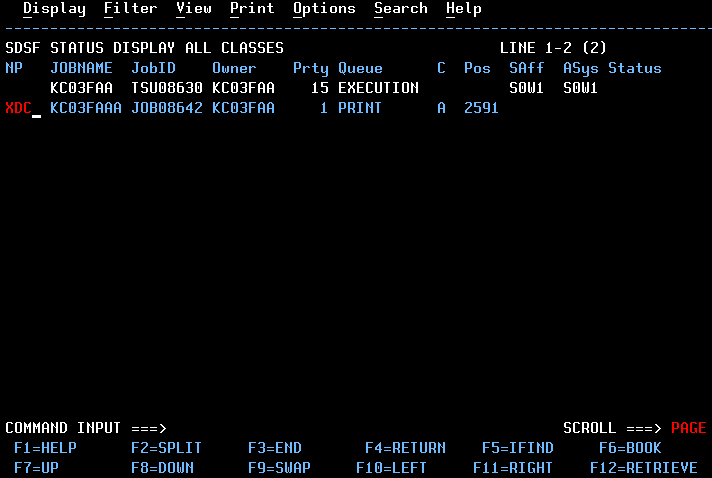
1. If you do not have a data set allocated for your output, you will need to do so
   1. To reach the **Data set Utility** screen, enter option **=3.2** from the **ISPF Primary Option Menu**
   2. Under the **ISPF Library** section, fill in the following:
      1. Project – your KC-ID
      2. Group – CSCI360 (You will see CSCI465 in these screen shots but change them to CSCI360)
      3. Type – OUTPUT
   3. Select option **A** to allocate a new data set and press [Enter]



* 1. Allocate your data set with the following properties and press [Enter]
     1. **Space Units** TRKS
     2. **Primary Quantity** 10
     3. **Secondary Quantity** 10
     4. **Directory Blocks** 10
     5. **Record Format** VB
     6. **Record Length** 134
     7. **Block Size** 4000
     8. **Data Set Name Type** PDS
  2. You should now have a data set **KC03nnn.CSCI360.OUTPUT**

**Saving Jobs on the Spool to Output Data set**

1. Once you have run a job, navigate to your output queue with the **=SD.ST** command.
2. Use the **XDC** command next to the output that you want to save



1. You will need to fill out the following fields, and press [Enter]
   1. **Data set name**  'KC03nnn.CSCI360.OUPUT'

Be sure to include the tick marks, and replace KC03nnn with your own KC-ID

* 1. **Member to use** name of the output file
  2. **Disposition** OLD

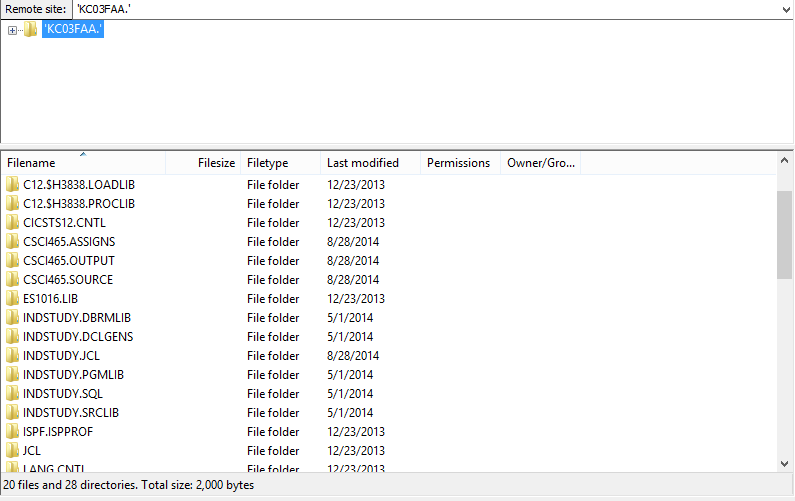
1. You should now have your output file saved in your output data set. You can verify this by going to the 2nd option from the **ISPF Primary Option Menu** and enter the following information under the **ISPF Library** section:
   1. **Project** KC-ID
   2. **Group** CSCI360
   3. **Type** Name of output data set
   4. **Member** Name of output file that you saved in step 3

**Using FileZilla to Download the Output File**

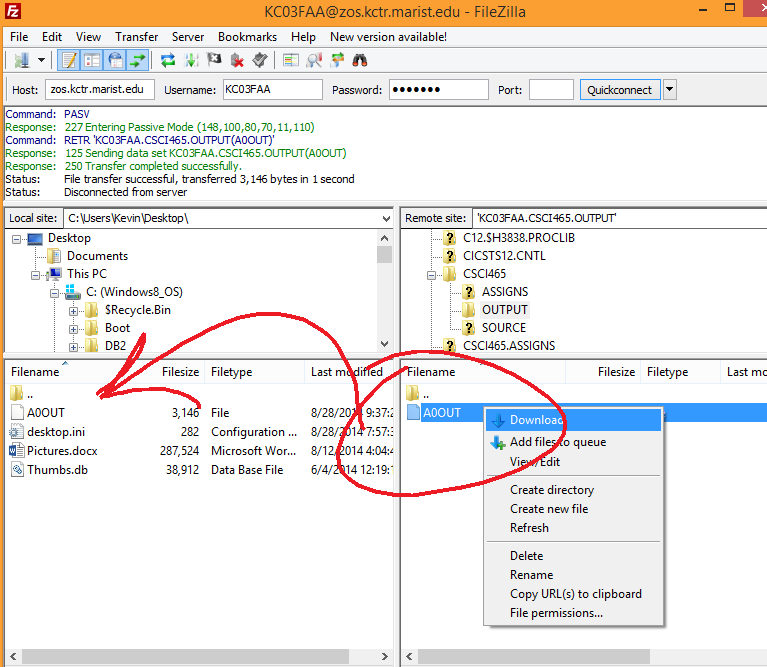
1. Download FileZilla at the following location & install:

<https://filezilla-project.org/download.php?type=client>

1. Open FileZilla and enter the following fields in the **Quickconnect** toolbar at the top
   1. **Host** zos.kctr.marist.edu
   2. **Username** KC-ID
   3. **Password** KC-ID password
2. Press the **Quickconnect** button
3. Under the ‘Remote Site’ window on the right side, you should see your KC-ID with a plus box next to it. Click on your KC-ID and you should see a list of your data sets (with folder icons to the left) in the window below.



1. Navigate to your output data set in the bottom pane and double click it to open your data set. You should see your output files.
2. You can either drag and drop your output file to a local site, or you can **right click** the file and click **Download** (it will download the file to the Local Site location that is currently selected on the right side



Please noted that there is a very good chance that the report created by your job at the bottom of the .txt file that you have dragged into a folder on your computer is NOT correctly formatted. Open the file with a text editor, change the page layout to landscape and scroll down to the end of the .txt file to make sure you have double-spaced lines as you would expect (if you have double spacing indicated in your mainframe program).

Scroll all the way to the left of the lines that you requested to be double spaced. If you see a number 0 at the beginning of those lines or if you see the number 1 at the beginning of lines that should be at the top of the page, continue on to #7 below.

1. Download the software named format.exe found in the Software Section of Blackboard. To make things simpler, move your .txt file you dragged down from Marist using Filezilla into the same folder you have downloaded format.exe into. Run format.exe, tell it the name of the .txt file you dragged down from Marist and it should properly format the report at the end of the .txt file. Make sure it works, though!

## 12.6 Transferring Files to Marist

**Overview**

This is helpful if you have .txt files or something of the same format that you want to transfer to Marist.  This is not meant to be comprehensive and it is also not meant to be a way for you to avoid using ISPF to edit your homework. Please do not do this on a regular basis.

**Instructions**

Create a PDS at Marist into which you will store the files you transfer there.  Add at least one "junk" member to this PDS.

Using Filezilla or another ftp program, sign onto Marist using the following parameters:

Host:  zos.kctr.marist.edu  
Username:  KCnnnnn        (your user ID at Marist, of course)  
Password:  your Marist password

Let the port number default.

At this point, you should be able to "open" your PDS at Marist and then click and drag files from your PC into your Marist PDS.

**Sometimes the PDS MUST have at least one member in order to click and drag members into it!  
  
Also, it is critical that the transfer type is ASCII and not AUTO or BINARY.  There is a chance that AUTO will work but BINARY will not.  The transfer occurs with any of the three but BINARY gives you unreadable characters in your PDS member when you view it to edit in TSO/ISPF.**

Note:  You will have to remove any file extensions, such as .txt or .jcl, from your files before you click and drag to transfer them to Marist.  The names of these files must be NO longer than eight characters, contain ONLY letters and digits, and must begin with a letter. Also, NO spaces allowed!

## 12.7 How to Fix Invalid Characters Found in ISPF

The following is NOT provided to encourage students to use Notepad, Wordpad or some other editor to write mainframe programs and applications! You should ONLY use TSO/ISPF’s editor!

Students sometimes have problems with invalid characters that cause problems in ISPF. Most invalid characters come from copying data from an outside editor like notepad. One example of this is the ASCII CHAR(10).

In ISPF Edit (Primary Option Menu Option 2), you can issue the command:

FIND P"."

This will find all characters in the member/dataset that ISPF cannot display. It highlights what looks to be empty spaces and you can issue the following command to fix it:

CHANGE P"." " "

This will replace the invalid characters with spaces.

## 12.8 Customizing TSO/ISPF

When you first enter a TSO/ISPF session, you will find that panels display basic IBM defaults. This document covers some optional changes that you can make to customize your TSO/ISPF session. Be careful and do not become too enthusiastic to start. Perhaps it would be best to try a few things before making most or all of the changes suggested here.

Follow the instructions below to modify your screen display. Your new configuration and setup options will automatically be saved when you log off.

**Move the Command Line to the Top of the Screen**

1. Select the "Menu" action bar from the ISPF Primary Menu
2. Select "Settings" from the pull-down menu
3. Set the ISPF settings as follows:

\_ Command line at bottom

\_ Panel display CUA mode

\_ Long message in pop-up

\_ Tab to action bar choices

\_ Tab to point-and-shoot fields

/ Restore TEST/TRACE options

/ Session Manager mode

/ Jump from leader dots

\_ Edit PRINTDS Command

/ Always show split line

1. Press PF3 to exit

Tip: You can also type **Settings** from any screen to bring up the settings panel.

**Remove the PF-Key Display**

1. Select the "Menu" action bar from the ISPF Primary Menu
2. Select "Settings" from the pull-down menu
3. Select Function Keys from Settings action bar
4. Type "6" (Remove function key display)
5. Press Enter
6. Press PF3 to exit

Tip: You can also type PFSHOW from any screen to see Pfkey displays. You can modify your Pfkeys by typing ZKEYS from any screen.

**Remove Underscores on Panels**

1. Select the "Menu" action bar from the ISPF Primary Option Menu
2. Select "Settings" from the pull-down menu
3. Select Colors/CUA Attributes (2) from the Settings action bar
4. Scroll to the next page
5. Modify 'Normal Entry Field" by entering NONE in Highlight column
6. Press PF3 to exit

Tip: You can also type CUAATTR from any panel to go directly to CUA Attributes.

**Display the Calendar**

1. Select the "Status" action bar from the ISPF Primary Option Menu
2. Select Calendar (3) from the pull-down menu
3. Press Enter

All fields on the calendar are point-and-shoot fields that function as follows:

* Click the right bracket ( > ) and the months go forward
* Click the left bracket ( < ) and the months go backwards
* Click the month (e.g. February) and the Calendar Month pop-up lets you specify the month to be displayed
* Click the year (e.g., 1998) and the Calendar Year pop-up lets you specify the year to be displayed
* Click Saturday, Sunday, or Monday to specify the start day for your calendar week
* Click any Day (e.g., Mo, Tu, We) to display the Start-Day pop-up
* Click any number in the month and the Julian Date pop-up displays (but you cannot enter anything in this pop-up)
* Click "Time" to display the pop-up for 12 or 24 hour format
* Click "Day of Year" to translate the Julian Date to a standard date (ccyy/mm/dd)

The colors of the calendar can be changed, for example:

1. Select the Menu action bar.
2. Select Status Area (9) from the list.
3. Select the Options action bar (on the Status Area pop-up).
4. Select Calendar colors (2).
5. Select 2 (Red) for Heading Date and Current Day, then press Enter. You'll see your changes immediately on the sample calendar.
6. After modifying the colors, Press PF3 twice to exit the two pop-ups.
7. You must now exit the Main Menu for your color choices to be displayed. Go to another option and then return to the Main Menu.

**Change the View Entry panel (Option 1) to default to Browse**

1. Type **=1** and press Enter.
2. Scroll or tab down (press PF8).
3. Place a slash (/) in front of the "Browse Mode" option.

**Remove the Warning for Recovery**

The following sequence of commands will set RECOVERY OFF. With recovery set to off, the "UNDO not available" message will not be displayed.

1. Type **=2** and press Enter.
2. Edit any dataset or PDS member.
3. At the Command line type:  **rec off nowarn**
4. Press Enter.

**Set Log/List Defaults**

The Log dataset used by ISPF should be deleted at the end of each session. To delete the log:

1. Select the Menu action bar from the ISPF Primary Option Menu.
2. Select "Settings" from the pull-down list.
3. Select Log/List action bar from ISPF Settings panel.
4. Select "Log Data Set Defaults" (1) and enter:

Process option 2

SYSOUT class A

Local Printer ID blank

Lines per page 60

Primary pages 1

Secondary pages 1

Log Message ID blank

1. PF3 to exit.

# Appendix A - Search Algorithms

## Linear Table Search

**Notes**: This algorithm assumes that routine has received the table beginning and ending addresses.

READ A SEARCH REQUEST

DO WHILE (NOT END OF FILE)

SET A POINTER TO THE BEGINNING OF TABLE

DO WHILE (NOT END OF TABLE AND ENTRY NOT FOUND)

INCREMENT TABLE POINTER

ENDDO

IF (ENTRY IS FOUND)

PROCESS THE FOUND ENTRY

PRINT THE NECESSARY DETAIL LINE

ELSE

PRINT "KEY NOT FOUND MESSAGE"

ENDIF

READ NEXT REQUEST RECORD

ENDDO

***Binary Search***

BINSRCH(SRCHKEY,TAB@,EOT,RET)

FOUND\_FLAG = 'N' - HI TOP SUBSCRIPT

DO WHILE (HI GE LO AND FOUND\_FLAG = 'N') - LO LOW SUBSCRIPT

MID = (HI + LO) / 2 <--- ONLY USE INTEGER

IF (ENTRY FOUND)

FOUND\_FLAG = 'Y'

ELSE

IF (SRCHKEY > MIDDLE KEY)

LO = MID + 1

ELSE

HI = MID - 1

ENDIF

ENDIF

ENDDO

IF FOUND\_FLAG = 'N'

PRINT ERROR MESSAGE

ELSE

PRINT SUCCESS MESSAGE

ENDIF

## Hashing

In your linear search assignment, you filled the table sequentially and when a record was to be processed, you searched the whole table until the requested entry was found. That process can be referred to as linear search. For large tables, linear search would be a waste of time. So instead of using linear search, random search is used. Hashing is one of many ways to incorporate a random search.

The basic idea of hashing is to take a key and convert it through some fixed process to a number in the range from 0 to n-1, where n is the maximum number of slots in the table. The resulting number can then be used to determine which entry in a table should be used to store the data item associated with the key. When a record needs to be retrieved, the same function must be used to calculate the slot number and find the requested record directly.

Trouble occurs, however, when two keys hash to the same table entry. This situation is called **COLLISION**. Since only one key-data pair may be stored in an entry of the table, it is necessary to find another location for the second (or subsequent) key-data pairs hashing to a particular entry.

**Linear Probe**: Collision problems can be solved by linear probe. If collision occurs, a linear search is done to find the next empty slot, where the current entry may be stored.

**Wrap Around**: If physical end of table is reached during the linear search, the search continues from the top of the table. A table is not considered full until you come back to the point where youstarted from.

**Hash Function**: The function used to calculate the table entry number. Same function must be used whenever referring to entries in that table.

There are many ways of hashing a key to determine the correct table slot into which an entry is to be stored or from which an entry is to be retrieved. The hash function for this program should produce a hash value in the range 0 through 29, which will accommodate a table of exactly 30 slots.

Hash values are similar to subscripts in that they both represent an ordering of entries. For a table with 30 slots, a subscript can range from 1 through 30, compared to the hash value range of 0 through 29.

A hash value multiplied by the length of an entry yields the displacement of that table slot from the beginning of the table, which is also the index for that slot.

## Hash Search

HASHSRCH(SRCHKEY,RET)

CALL HASH ROUTINE

CALCULATE TABLE ENTRY ADDRESS (SLOT ADDRESS)

(HASH VALUE \* ENTRY LENGTH + BEGINNING OF TABLE

SAVE DISPLACEMENT

RC = -1 SET THE RC TO NEG. ONE

DO WHILE (RC = -1) DO WHILE RC IS NEG. ONE

IF (TABLE ENTRY = 0) IF EMPTY SLOT FOUND

RC = 4 SET THE RC FOR EMPTY SLOT

RETURN TABLE ADDRESS

ELSE

IF (TABLE ENTRY = SRCHKEY) IF MATCHING KEY ARE FOUND

RC = 0 SET THE RC FOR MATCHING KEYS

RETURN TABLE ADDRESS

ELSE

INCREMENT TABLE POINTER

IF (EOT) IF END OF TABLE

WRAP AROUND TO THE BEGINNING

ENDIF

IF (TABLE POINTER = SAVE DISP.) IF TABLE IS FULL

RC = 8 SET RC FOR TABLE FULL

ENDIF

ENDIF

ENDIF

ENDDO

**NOTES**: RC = **0** - Entry found in the table

**4** - Empty entry found, key not in table

**8** - The table was full

**Example:**

----------------

0 ! 230 !

---------------- **Hash function**: Divide by 10 and

1 ! 328 ! take the remainder (REM)

---------------- **DISP = REM \* ENTRY-LENGTH + TABLE-ADDR**

2 ! 000 !

----------------

3 ! 000 ! **BUILD** **SEARCH**

---------------- 230 325

4 ! 000 ! 425 230

---------------- 888 328

5 ! 425 ! 369

---------------- 328

6 ! 000 !

----------------

7 ! 000 !

----------------

8 ! 888 !

----------------

9 ! 369 !

----------------

# Appendix B - Sort Algorithms

## Bubble Sort

**Description:**

A table of records is sorted in ascending order by key.

The keys of the 1st and 2nd records are compared and records exchanged if out of order.

This continues with the 2nd and 3rd, 3rd and 4th, etc., until the largest key is at the end.

The index of the final exchange is saved, as all records past it are in order.

If no exchanges took place, all records are in order and the sort is complete.

This process is repeated for the “sub-table” which ends at the saved index, again and again, with the index decreasing each time, until no exchanges take place, and the sort is complete.

**Pseudocode:**

I, J are SUBSCRIPTS

END is initially the END OF TABLE SUBSCRIPT

FLAG = 1

DOWHILE (FLAG=1 AND END>1)

I=1

J=2

FLAG=0

DOWHILE (J<=END)

IF (ENTRY(I)>ENTRY(J))

FLAG=1

SWAP ENTRY(I) AND ENTRY(J)

ENDIF

I=I+1

J=J+1

ENDDO

END=END-1

ENDDO

## Selection Sort

**Description:**

A table of records is sorted in ascending order by key.

Beginning at the first entry, the entire table is scanned for the smallest key.

The record with that key is exchanged with the first record in the table.

(Now the first entry in the table contains the record with the smallest key.)

This process is repeated for the “sub-table” which begins at the second entry, then the one beginning at the third entry, etc., until the final sub-table beginning at the next-to-last entry.

**Pseudocode:**

BEGIN is initially the subscript of the first entry

END is the subscript of the last entry

LOW and I are additional subscripts

DOWHILE (BEGIN<END)

LOW=BEGIN

I=BEGIN+1

DOWHILE (I NOT> END)

IF(ENTRY(LOW) > ENTRY(I))

LOW = I

ENDIF

I=I+1

ENDDO

SWAP ENTRY(BEGIN) WITH ENTRY(LOW)

BEGIN=BEGIN+1

ENDDO

## Insertion Sort

**Description:**

A table of records is sorted in ascending order by key, using a “bridge hand” process.

Assume that 1 < j ≤ N and that records R1, ..., Rj-1 have been rearranged so K1 ≤...≤ Kj-1 .

Compare the new key Kj with Kj-1 , Kj-2 , ... in turn until discovering that Rj should be inserted between records Ri and Ri+1; then we move records Ri+1, ..., Rj-1 up one space and put the new record into position i + 1. (In the algorithm, the comparing and moving operations are interleaved.)

**Pseudocode:**

LOW, HIGH are subscripts initially set to the first and last entries in the table

J,K are temporary subscripts

J=LOW+1

DOWHILE (J<=HIGH)

K=J

DOWHILE(K>LOW AND ENTRY(K)<ENTRY(K-1))

SWAP ENTRY(K) AND ENTRY(K-1)

K=K-1

ENDDO

J=J+1

ENDDO

## Insertion Sort With a Bucket

LOW, HIGH are subscripts initially set to the first and last entries in the table

J,K are temporary subscripts

TEMP is a temporary bucket to hold an entry in the table

J=LOW+1

DOWHILE (J<=HIGH)

TEMP=ENTRY(J)

K=J

DOWHILE(K>LOW AND ENTRY(K)<ENTRY(K-1))

ENTRY(K)=ENTRY(K-1)

K=K-1

ENDDO

ENTRY(K)=TEMP

J=J+1

ENDDO

# Appendix C - Merge Algorithms

**Note**: These algorithms assume that the arrays to be merged are already sorted on ascending order of the key and the resulting array will also be sorted in ascending order.

You might have to change the algorithms little bit depending on your requirements.

**Parameters Expected**: Address of TABLE1

Address of TABLE2

Address of MERGED TABLE

End of TABLE1

End of TABLE2

End of MERGED TABLE

DO WHILE (NOT END OF EITHER TABLE)

IF (TAB1(KEY) < TAB2(KEY))

PUT TAB1 ENTRY INTO MERGED TABLE

INCREMENT TAB1 POINTER

ELSE

IF (TAB1(KEY) > TAB2(KEY))

PUT TAB2 ENTRY INTO MERGED TABLE

INCREMENT TAB2 POINTER

ELSE

PUT TAB1 OR TAB2 ENTRY IN THE MERGED TABLE

INCREMENT TAB1 POINTER

INCREMENT TAB2 POINTER

ENDIF

ENDIF

INCREMENT MERGED TABLE POINTER

ENDDO

**(Now, at end of at least one table)**

DO WHILE (NOT END OF TAB1)

PUT TAB1 ENTRY INTO MERGED TABLE

INCREMENT TAB1 POINTER

INCREMENT MERGED TABLE POINTER

ENDDO

DO WHILE (NOT END OF TAB2)

PUT TAB2 ENTRY INTO MERGED TABLE

INCREMENT TAB2 POINTER

INCREMENT MERGED TABLE POINTER

ENDDO

RETURN ENDMERGE POINTER

**Another Merge Algorithm**

MERGE(TAB1,EOT1,TAB2,EOT2,MERGE,ENDMERGE)

DO WHILE (NOT END OF BOTH)

IF (END TAB1)

PUT B INTO MERGE

INCREMENT TAB2 POINTER

ELSE

IF (END TAB2)

PUT A INTO MERGE

INCREMENT TAB1 POINTER

ELSE

IF (A < B)

PUT A INTO MERGE

INCREMENT TAB1 POINTER

ELSE

IF (A > B)

PUT B INTO MERGE

INCREMENT TAB2 POINTER

ELSE

PUT A INTO MERGE

INCREMENT TAB1 POINTER

INCREMENT TAB2 POINTER

ENDIF

ENDIF

ENDIF

ENDIF

INCREMENT MERGE POINTER

ENDDO

RETURN MERGE POINTER

# Appendix D – The Marist Mainframe

zos.kctr.marist.edu is the address of the Marist mainframe. In the computer science labs you can use TN3270 and port 1023.

PASSWORD for the USERIDs are their USERIDs.   
The third position of the USERID is the number ZERO.

When you first logon use the userid as your password and you will be prompted for a new one. Make your new one exactly 8 characters long, beginning with a letter and containing one or two digits. It is NOT case sensitive!  
  
To get on to TSO

logon kcXXXXX

which takes you to the password screen

enter your password.

Wait a few moments and then hit enter a couple of times to get to the main TSO screen

TSO commands are done on a command line that is usually at the bottom of the screen. You enter numbers on the command line. You can proceed through the menus/screens or you can jump to different ones (if you know where you want to go) by using an =.

You can do most things necessary for this class through 3 options on the ISPF PRIMARY OPTION MENU:  
  
2 Edit

3 Utilities

13 SDSF ( Spool Display and Search Facility)

Learn how to use TSO/ISPF as it is the industry standard! It is also just another thing you can add to your resumé when you enter the working world!

**Whatever you do, do NOT write your programs using a text editor and then ftp them to Marist to only run them. You could be opening a huge can of worms if you do!**

This is what is on the UTILITY SELECTION PANEL:

1 Library Compress or print data set. Print index listing.   
 Print, rename, delete, browse, edit or view members

2 Data Set Allocate, rename, delete, catalog, uncatalog, or   
 display information of an entire data set

3 Move/Copy Move, or copy members or data sets

4 Dslist Print or display (to process) list of data set names.

Print or display VTOC information

5 Reset Reset statistics for members of ISPF library

6 Hardcopy Initiate hardcopy output

7 Transfer Download ISPF Client/Server or Transfer data set

8 Outlist Display, delete, or print held job output

9 Commands Create/change an application command table

11 Format Format definition for formatted data Edit/Browse

12 SuperC Compare data sets (Standard Dialog)

13 SuperCE Compare data sets Extended (Extended Dialog)

14 Search-For Search data sets for strings of data (Standard Dialog)

15 Search-ForE Search data sets for strings of data Extended  
 (Extended Dialog)

Option ===>

Type the number down in the Option field which will take you to another screen.

To see what data sets you have, choose option 4 (Dslist which is IBM speak for data set list) once there:

put your id in the dsname level field and hit enter

to look at contents of a particular file put b on line to left and   
 hit enter

to edit contents of a particular file put e on line to left and hit  
 enter

After editing:

to submit a job for execution type submit or sub on the command line  
  
Then to look at it, you need to go to the SDSF screen as noted below:

This is what is on the SDSF PRIMARY OPTION MENU:   
  
 ----------------- SDSF PRIMARY OPTION MENU -----------------------

DA Active users INIT Initiators

I Input queue PR Printers

O Output queue PUN Punches

H Held output queue RDR Readers

ST Status of jobs LINE Lines

NODE Nodes

LOG System log SO Spool offload

SR System requests SP Spool volumes

MAS Members in the MAS

JC Job classes RM Resource monitor

SE Scheduling environments CK Health checker

RES WLM resources

COMMAND INPUT ===>   
  
The first time you use this, type

OWNER your user id

in the command line. This will then show only your jobs when you check ST.

ST, or STATUS, will display the status of all tasks owned by you

To look at the output, SE to the left of the file. To delete output P to the left and follow directions.

If you want to save an output file (to ftp back for debugging) type XDC to the left of the file you want to save. This takes you to a screen that allows you to save it in a file. Use new as the disp the first time, old if you use the same file for output. Record length should be 134 with variable length records.

You can get back to the main screen or any previous screen by F3. To get to a different screen directly (assuming you know which one you want) =screennum.

For example to get to SDSF, =13 on any command line.

To get back to your dslist =3.4 on any command line.

If you have VISTA:

|  |
| --- |
| Download and install a 3270 emulator |
|  |
| In order to access the Marist mainframe you'll need a 3270 terminal emulator. Follow the instructions based on your operating system:   * If you are running Windows, pick up an emulator from Tom Brennan Software here(http://www.tombrennansoftware.com/download.html). Download the Vista V1.24 .exe file. (Windows Vista users: use the same link, but download Vista V1.26 instead.)   Double-click on the .exe file and follow the installation instructions.   * If you are using a Mac machine, you can pick up a 3270 emulator here(http://www.versiontracker.com/dyn/moreinfo/macosx/15091). * If you are running Linux, install the following package: x3270 -port 623. See this page (http://x3270.bgp.nu/)for more information.   Once installed, the default location to access the 3270 emulator on a Windows machine is: Start-> Programs -> Vista tn3270 -> Vista Standard Session (or Vista TN3270 Standard Session for Windows Vista users).  Open the emulator. The first time you do this, you might get this error: Vista Connection Error 2  If you do, simply click "OK" to proceed. You are now ready to set up your emulator and connect to the mainframe. |
| Use this one first:  To Reach ZOSKCTR7 system on port 1023, go to: zos.kctr.marist.edu  If it does not work, try this one:  To Reach ZOSKCTR7 system on PORT80 go to:  portforward.kctr.marist.edu |
|  |