Reverse engineering and PORTACALC

Dale W. Wicks

The University of Montana

Let us know how access to this document benefits you.
Follow this and additional works at: https://scholarworks.umt.edu/etd

Recommended Citation
Wicks, Dale W., "Reverse engineering and PORTACALC" (1985). Graduate Student Theses, Dissertations, & Professional Papers. 5548.
https://scholarworks.umt.edu/etd/5548

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact scholarworks@mso.umt.edu.
COPYRIGHT ACT OF 1976

THIS IS AN UNPUBLISHED MANUSCRIPT IN WHICH COPYRIGHT SUBSISTS. ANY FURTHER REPRINTING OF ITS CONTENTS MUST BE APPROVED BY THE AUTHOR.

Mansfield Library
University of Montana
Date: 1985
Reverse Engineering and PORTACALC

by Dale W. Wicks

Presented in partial fulfillment of the requirements for

the degree of Master of Science

University of Montana

1985

Approved by:

Chairman

Dean, Graduate School

Date
Today, spreadsheets are an important classification of software. They can be used to assist in solving many problems. PORTACALC is written by Glen Everhart.

Software engineering techniques were used to study PORTACALC. A reverse engineering strategy was applied. From the existing program, a program specification and a functional description were developed. These documents describe what tasks the program performs and how the program performs those tasks.

A stated goal of PORTACALC was portability. This goal was not met. Much work is necessary in porting the program to an environment other than the one for which it was specifically designed. The author's choice of FORTRAN as the implementation language was a reasonable choice for a portable program. The problem with PORTACALC's implementation is its used of non-standard extensions of FORTRAN.
# Table of Contents

Abstract ............................................................................................................................ ii

Table of Figures .................................................................................................................. v

I. Project formulation
   1. Introduction and initial work .............................................................. 1
   2. Background on spreadsheets .............................................................. 2
   3. Background on 'PORTACALC' .......................................................... 8
   4. Statement of problem and goals ......................................................... 8
   5. Tools, techniques, and methods to be used .................................. 11

II. Reverse Engineering
   1. What is Reverse Engineering? ........................................................... 13
   2. Why is Reverse Engineering important today? ........................... 14
   3. Who will use the results of the Reverse Engineering process? .... 15

III. Implementation
    1. PORTACALC = portability? ............................................................. 18
    2. Features which promote portability ................................................. 19
    3. Features which detract from portability ...................................... 20
    4. FORTRAN's suitability for a portable program of this type ....... 25
    5. How could PORTACALC have been made more portable? ....... 28
    6. PORTACALC's implementation .................................................... 29
    7. Concluding remarks on PORTACALC's implementation .......... 30

IV. Design
    1. Overview of design section ............................................................ 32
    2. Tools used to portray PORTACALC's physical design .............. 32
    3. Design process ............................................................................. 34
    4. Important data structures of the system ..................................... 36
    5. Critique of PORTACALC's design ............................................... 37
V. Analysis
1. Analysis tools ------------------------------------- 42
2. Description of the system ------------------------- 46
3. Description and modularization of command ------- 49
   processor module

VI. Conclusion
1. Reverse Engineering and Implementation -- ------- 53
2. Critique of tools used --------------------------- 56

Appendix A. Structure Charts ------------------------ 62
Appendix B. Subroutine Descriptions ----------------- 70
Appendix C. Common Data Dictionary ------------------ 146
Appendix D. Data Flow Diagrams ---------------------- 159
Appendix E. Data Dictionary -------------------------- 164
Appendix F. Mini-Specifications of the Processes in Data Flow Diagrams 173
Appendix G. Command Descriptions --------------------- 177
Appendix H. RXVP80™, A Software Analysis, ----------- 185
   Documentation, and Test System
Bibliography ---------------------------------------- 191
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure I.1</td>
<td>3</td>
</tr>
<tr>
<td>Figure I.2</td>
<td>5</td>
</tr>
<tr>
<td>Figure I.3</td>
<td>7</td>
</tr>
<tr>
<td>Figure III.3</td>
<td>22</td>
</tr>
<tr>
<td>Figure III.2</td>
<td>22</td>
</tr>
<tr>
<td>Figure V.1</td>
<td>43</td>
</tr>
<tr>
<td>Top Level Structure Chart</td>
<td>63</td>
</tr>
<tr>
<td>CALC Structure Chart</td>
<td>64</td>
</tr>
<tr>
<td>CMND Structure Chart</td>
<td>65</td>
</tr>
<tr>
<td>DSPSHT Structure Chart</td>
<td>66</td>
</tr>
<tr>
<td>POSTVL Structure Chart</td>
<td>67</td>
</tr>
<tr>
<td>RECALC Structure Chart</td>
<td>68</td>
</tr>
<tr>
<td>XQTCMD Structure Chart</td>
<td>69</td>
</tr>
<tr>
<td>Data Flow Diagram - Context Diagram</td>
<td>160</td>
</tr>
<tr>
<td>Data Flow Diagram - Diagram 0</td>
<td>161</td>
</tr>
<tr>
<td>Data Flow Diagram - Diagram 2</td>
<td>162</td>
</tr>
<tr>
<td>Data Flow Diagram - Diagram 3</td>
<td>163</td>
</tr>
</tbody>
</table>
I. Project Formulation

I. 1. Introduction and initial work

The spreadsheet program, 'PORTACALC' was available, but needed revision in order to work on the DEC2060 under TOPS-20. The project interested me, due to my mathematical background. My goal was to find a project on which the techniques and methods learned in my graduate software engineering classes could be applied. This project seemed to fit the criteria and also gave me a chance to explore the area of reverse engineering. Therefore my project became the 'spreadsheet project', to analyze, redesign where necessary, and then to implement a spreadsheet program on the DEC2060 under TOPS-20.

My initial analysis had two thrusts: understanding spreadsheets in general, and secondly, analyzing how 'PORTACALC' works. In order to perform the first task, three focal points were taken. The first point was to run a spreadsheet such as VISICALC in order to increase my understanding of spreadsheets. Secondly, reading articles and books relating to spreadsheets informed me further on the basics of spreadsheet use along with improvements of new generation spreadsheets. My third point of interest was to talk to some users of spreadsheets to get an understanding
of how spreadsheets are used, which features are necessary, and which features are desirable.

This initial study gave me a better understanding of spreadsheet software. This also helped me to understand the importance of spreadsheet programs in the microcomputer market, and their increasing use in many areas. The rationale behind different spreadsheet products and how these products function was also important background information. This study gave me a basis from which to continue my work.

I. 2. Background on spreadsheets

A spreadsheet program is an analysis tool, which is often used by accounting and business people. Spreadsheets are very good at answering 'What if...?' types of questions. Once a spreadsheet is set up to describe a system or cost performance model, one can see how changing different factors can change the final result. Thus a spreadsheet program is a valuable aid which can be used in the problem solving process.

The spreadsheet is just an electronic version of the accountant's columnar pad, which makes it much easier to create, edit, and use financial and other types of models. Spreadsheets can be set up to solve almost any
problem which can be attacked with paper, pencil, and a calculator. They are being used increasingly in all sizes of businesses, to help speed up and simplify financial analysis.

A spreadsheet is generally made up of a large matrix or grid. Normally, each row is assigned a number and each column a letter. At the intersection of rows and columns, there is a cell. The cells are identified by their column-row coordinates. For example, cell B3 would be at the intersection of the second column and the third row, see figure I.1.

FIGURE I.1.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TABLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td></td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>A2+C2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three types of information can be associated with each cell: text, numbers, and formulas. The text can be used as labels to help format the output. It also can be used to describe or comment on what is going on in the spreadsheet. The numbers associated with the cells can be constants,
or the values of the formula in the cell when it has been evaluated. These values can be expressed in various forms such as integer, real, and octal.

A cell may contain a formula, which means the numerical value of this cell is a function of the values of the other cells. If a cell is to contain a constant, then the formula can be thought of as that constant. In Figure 1.1, cell B3 will contain the sum of the values in cells A2 and C2, which in this case would be 59. If A2 or C2 are changed, the value in cell B3 will change accordingly. This is a simple example of a formula; more complex formulas can be accommodated. Special mathematical and other types of functions can be used in the formulas of a cell. Functions such as AVERAGE, SUM, STANDARD DEVIATION, and logical functions such as IF may be included. These functions add to the power and ease of use of a spreadsheet.

The spreadsheet often has several modes of recalculation. The default mode is usually a recalculation after each entry. This can then be turned off, so that recalculation only occurs on request. The calculations are normally performed in a row-wise manner. The formula in the top left cell in the sheet is computed first, followed in a linear pattern by the rest of the cells in the top row. This is followed by calculating the second row from left to right. This pattern continues in a row by row manner until the
whole sheet has been recalculated. Many sheets also have the option performing the calculations in a column-wise manner similar to the row calculations.

Most spreadsheets offer the flexibility to show values, text, or formulas for each individual cell of the spreadsheet. This allows one to format the spreadsheet to appear however one wants.

**FIGURE 1.2.**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Fixed Cost</td>
<td>1000</td>
<td>Prod.Cost/It</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0-1</td>
<td>Q-4</td>
<td>TOTAL</td>
</tr>
<tr>
<td>4</td>
<td>Sale Cst/It</td>
<td>50</td>
<td>55</td>
</tr>
<tr>
<td>5</td>
<td>Num sold</td>
<td>300</td>
<td>275</td>
</tr>
<tr>
<td>6</td>
<td>Gr. Income</td>
<td>B4*B5</td>
<td>C4*C5</td>
</tr>
<tr>
<td>7</td>
<td>Expense</td>
<td>B1+(B5*D1)</td>
<td>B1+(C5*D1)</td>
</tr>
<tr>
<td>8</td>
<td>Net Income</td>
<td>B6-B7</td>
<td>C6-C7</td>
</tr>
</tbody>
</table>

Suppose one wishes to set up an income/expense sheet for a business. In Figure 1.2, a simple spreadsheet of this type has been set up. The first row has some information which will referred to by other cells in the sheet. A1 and B1 represent the fixed costs of the business, where C1 and D1 label
and represent the cost to produce the item the business sells. Column B from cell B3 to B8 include the information and formulas needed to study financial status during the first quarter. Cells B4 and B5 contain constant values for the income per sale and the number of items sold. The cells B6 through B8 contain formulas for gross income, expenses, and net income respectively. Column C contains the same information about the fourth quarter. Cell D8 uses the SUM function to determine the total net income for the business.

Once the model has been set up, one could do some "what if" type of analysis using the model. For instance, what if the cost of producing an item was changed? By simply changing the contents of one cell, D1, one could see how the rest of the model performs. This type of manipulation allows one easily to check out many alternatives which might not otherwise have been considered. One can save all or part of a spreadsheet for future use. An alternate view of the example in Figure 1.2, is shown in figure 1.3. Here the formulas are not shown, just the values and labels. This shows how a simple accounting system could be set up using a spreadsheet.

Some of the other powers that the electronic spreadsheet has are
described below. Spreadsheets not limited by the size of a physical page, or terminal screen. They are only limited by the memory of the host computer and limitations built into the program. Correcting errors in the set up of a spreadsheet is easily done by changing contents of cells with edit commands. If one forgets to include a column or row, it could be inserted at a later date. Once one has set up a formula in a cell, that formula can be replicated in other cells. The references by variables to other cells can be shifted as well during the replication process. Another plus is that an electronic spreadsheet does not make math errors.
I. 3. Background on 'PORTACALC'

'PORTACALC', written by Glenn Everhart, was available on tape from DECUS, a DEC users group, free of charge. 'PORTACALC' is written in FORTRAN IV PLUS, which is a dialect of FORTRAN 66. 'PORTACALC' was developed under the RSX operating system, but also runs in native mode under VMS as well.

'PORTACALC' is similar to other spreadsheets, and has some unique features. Some of the features include: multiple equations per cell, built in interactive calculator, ability to refer to a command file, and each window can become a separate window onto the physical sheet. According to the author, Glenn Everhart, 'PORTACALC' was written for portability and functionality. This may be true for portability between RSX11 and VMS, but porting the program to non-RSX-11 or VMS systems requires work. The implementation section goes into detail on the portability problems of PORTACALC.

I. 4. Statement of problem and goals

The problem was to use software engineering methods to study and analyze 'PORTACALC', to come up with a physical and logical specification
of the system and its design. Reverse engineering\textsuperscript{3} was used in the analysis process. Reverse engineering is the process of taking an existing program, analyzing it to understand the functions of the code and how the code performs these functions. FORTRAN's suitability for a program of this type and PORTACALC's success in being portable are investigated. Re-implementation of a modified spreadsheet under TOPS-20 on the DEC2060 is a goal. The modified version has goals of being more portable, and modifiable than the original PORTACALC. This may be only marginally successful, since it is very difficult to improve the structure of an existing program without completely rewriting it.

Two documents result from the reverse engineering process done in this paper: a program specification, and a functional description. The program specification describes how the program is designed and how it accomplishes the tasks of the system. This physical design of the program is described using formatted descriptions of each routine. This includes what the routine is to accomplish and the data flow into and out of the routine. The program design is specified by structure charts and description of the routine in the system. The functional description of the system, specifies what the system actually does, rather than how it is done.
Data flow diagrams, with adjoining data dictionary and mini-specifications, are used to describe the functional or logical specification of the system. These tools help describe in a visual manner what the program actually does.

These documents would be useful if one were to build a new system to accomplish the same or similar task. The first step in the analysis for creating a new system is gaining an understanding of what tasks are being done and how these tasks are being accomplished. In the past, the goal of most computer systems was to automate manual processes. But today, many new systems are replacing or consolidating existing programs. Therefore the process of reverse engineering will become an even more important process in the future. If the process is done well, the documents developed in the reverse engineering effort can help hold down the future costs of maintenance of a system, since they will help portray the design and structure of the system.

Glen Everhart, the author of the original program, declared some goals for the program. He stated⁴ "The program must be easily transportable to all presently used DEC systems (i.e. written in FORTRAN IV)." In other words, FORTRAN IV is a standard language which promotes portability.
This goal of portability is not accomplished when PORTACALC was transported to either the DEC 2060 under TOPS-20 or to Berkeley UNIX FORTRAN on a VAX. The design decisions which promote and detract from the transportability of the program are noted. The pros and cons of using FORTRAN as the implementation language for this type of program are discussed.

1.5. Tools, techniques, and methods to be used.

Many of the tools and techniques learned in software engineering courses were used in this process. RXVP80\textsuperscript{8}, an automated analysis tool was used to assist in the analysis process and proved invaluable in gaining an understanding of the system. It documents the many interrelationships of the system, and much of the information on the data of the system. For a more detailed discussion on RXVP80\textsuperscript{8} see Appendix H.

The design of the system is depicted with structure charts\textsuperscript{10} along with descriptions of all of the subroutines of the system. Included in the routine descriptions are all of the data flowing in and out of that module. A data dictionary of the COMMON variables and global parameters is also included.

The logical view of the system is shown by uses of data flow diagrams
as described by DeMarco\(^3\). A data dictionary of data flows and mini-specifications of process complete this functional view of the system.

An advantage of data flow diagrams and structure charts is that they both present a visual picture of the system, thus making it much easier to quickly understand the system as a whole and also the interrelationship between the parts of the system.
II. Reverse Engineering

II.1. What is Reverse Engineering?

Reverse Engineering is the process of taking an existing program, and analyzing it to understand the functions of the code and how the code performs these functions. This would take place in the process of updating an existing system. The results of the reverse engineering process are two documents, a program specification, and a functional description. These documents would be used to generate a program or system which produces the same or upgraded functions as the original program.

The first step in any analysis process is understanding the existing system or process. If adequate specification and requirements documents were available, this would be more easily accomplished. In the past many systems did not have an up-to-date requirements and specification document of the system. Therefore, when it comes time to update or replace the system, one is not exactly sure what needs to be replaced. An exact specification of the what the system is doing is not available. So the reverse engineering process is a way of getting the information needed for understanding the existing system or process.
II. 2. Why is Reverse Engineering important today?

In the past, most systems which were implemented were systems which automated human functions. Today and in the future, more and more of the programming effort is either maintenance programming or replacing an existing system. If the reverse engineering process can assist in and reduce the cost of these efforts, it will be an increasingly valuable tool.

Many of the existing programs do not have an adequate up to date requirements document or specification document. This is especially true with programs that have been around for a time. As updates and maintenance are performed on a program, the documentation is not always kept up to date. Eventually the patched program needs replacement. The reverse engineering process aids in developing a description of the system as it is, and a functional description of what the system must do. This information is vital in developing a new system which will perform the tasks of the old system as well as new functions.

Part of the problem is keeping the requirements and specification documents up to date after modification of the program. Changes to a program may effect more than one part of the program, and if one is to
keep the documents current, changes must occur in the corresponding parts of the documentation. This is often a tedious task that is also prone to errors. There are some techniques and tools which are now used to help alleviate these problem. Demarco uses data flow diagram with associated data dictionary and mini-specification to portray the system. This system attempts to reduce the redundancy of the description which makes modification easier. In the ISDOS project the data dictionary is automated by use of a relational data base. Then when changes to the system are made, the information only needs to be modified in one place. These tools and others make it easier to keep the documents current.

II. 3. Who will use results of the Reverse Engineering process?

There are two important users of the results of the reverse engineering effort. The systems designer, who will design the new equivalent system, and the maintenance programmer, who will do the upkeep and improvements on the existing and new system. Another user, which could be included in previous two, would be a person attempting to transport a system to another computer or operating system. They each can use the results of the reverse engineering process.
The systems designer needs to know the systems structures, program functions, and data usage to develop the new system. The system structure helps him to understand the system as a whole. The program functions inform the systems designer of what the new system must include. The systems designer must also understand how the data is used in the system. The data usage also would define the data interface to the system.

Today in many organizations, fifty percent and more of programming expenses go to maintenance programming, and this figure is increasing. The software engineering techniques are helping to gain control over expenditures on systems that are being built today. The use of reverse engineering extends the use of the software engineering techniques to maintenance and may be one way of helping to control this escalation on the reimplementation of existing systems. By providing the maintenance programmer with adequate documentation, he will be able to perform the maintenance tasks more efficiently and effectively.

The maintenance programmer needs to know the program structure, data structure, and data flow. The maintenance programmer must have a good understanding of the program structures so that he can revise the
system in such a way that it will maintain the integrity of the system. A maintenance programmer who understands the data structures, and the data flow through the system, is well on his way to understanding the system. This knowledge helps the programmer limit the side effects of his/her changes.

The reverse engineering process is not very different from other software engineering techniques. Instead it is an extension of software engineering techniques to the areas of maintenance and redevelopment. Any system which has been used for period of time has probably evolved. When the time comes to replace this system, the reverse engineering process can help to provide the requirements and specifications for the new system.
III. IMPLEMENTATION

III.1. PORTACALC = portability?

This section deals with the implementation of PORTACALC. How well does it meet Mr. Everhart's goal of being portable? FORTRAN's use for this project is also discussed. How appropriate is FORTRAN for a project of this type with goals of portability?

Mr. Everhart listed portability as one of the goals of the system. Accordingly some decisions were made to support this goal. Other decisions by the authors of the system do not increase the system's portability, but rather detract from it in various ways.

Ideally, a program could be run off a tape on to a machine, compiled, and then run. This would be a truly portable program. The second degree of portability is a program which has a few well-documented routines to do specific tasks which must be modified to implement the program on a new system. Anything beyond these, one must re-write parts of the system. This is where design is important, for what actually is occurring is maintenance. PORTACALC fits into the third category, since it did not compile and run, nor does it have a few well-documented special purpose routines to be modified in order for it to run. Instead it has many modules
which need modification, and all systems dependent parts are not contained in well specified, well documented routines.

III. 2. Features which promote portability

On the plus side, a number of decisions have a positive effect in making the system more portable. The FORTRAN IV language was a reasonable choice for a language. A more in depth discussion follows in section III.4. on FORTRAN's suitability for a program such as this.

Though PORTACALC is a large program, with over 12,000 lines of code, it has been broken down into many subroutines. Many are of reasonable size, with sufficient documentation to describe their functions. This helps to make a program more understandable, which is essential before modification.

Some of the systems dependent routines have been separated into their own routines. Therefore when the program is moved to a new computer, one can alter the system dependent routine and have a running system. Examples of system dependent routines are the TTYINI routine, which performs system dependent initializations and the UVT100 routine which performs screen manipulations. The UVT100 routine must be customized
for the type of terminal the program is to run on. Copies of the UVT100 routine have been supplied for some of the commonly used terminals such as the VT52 and VT100. One must get the routine ready and copy it into the UVT100 routine, which is called by the other routines of the program for screen manipulations such as cursor placement and erasing screen.

III. 3. Features which detract from portability

There are a number of design and implementation decisions which detract from the program's portability. This section will list some of the those decisions.

Although many of the routines are reasonable in length, a few are excessive in length. For example subroutine XQTCMD has 1327 statements and 1647 lines and subroutine CMND has 465 statements and 786 lines. These two routine are command processors and would have been easier to understand if they had done part of the work of the command processing in other smaller special purpose routines.

Since PORTACALC is written in FORTRAN IV, there are no character variables. The characters of commands and formulas must be stored in some other type of variable. In PORTACALC, characters are stored in
logical variables. This does increase the complexity of the program, since it is not clear immediately how this is done.

Characters can not be stored directly in a computer since all computer memory is binary bits, 1's or 0's. This means there is not an 'A' bit that can be stored in a location. Instead the characters, digits, and common punctuation characters are coded by binary numbers. There are two common coding schemes which are used for this task: EBCDIC - Extended Binary Coded Decimal Interchange Code, and ASCII - American Standard Code for Information Interchange: EBCDIC uses an eight bit code where as ASCII uses a seven bit code to represent characters. With both coding schemes a character will fit in a single byte of memory. A byte is usually 8 bits long, although on the DEC 20 it is 7 bits long.

This leads to the consideration of how these character codes are stored in memory. There are two basic ways characters can be stored in a word of memory on a computer: packed and unpacked (using an A5 and A1 formats respectively in FORTRAN).

The packed format puts as many characters into a word as will fit. The number is dependent on the size of the word on the machine one is working on. If one is working on a VAX which has a thirty-two bit word,
four characters could be packed into one word of memory. The DEC-20 using a packed format would store five ASCII characters per word. Since all but one bit of each word is used to store binary coded characters, this is a very space-efficient way to store characters (see Figure III.1) The problems come when one tries to access individual characters in this packed form. There is no efficient or easy portable way to access individual characters.

FIGURE III.1.

Unpacked form, one character stored in byte 1.

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
<th>Byte 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
<td>00000</td>
</tr>
</tbody>
</table>

Parity bit

FIGURE III.2.

Packed format, five characters per word, in this case A,B,C,D,E respectively.

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
<th>Byte 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>10000</td>
<td>10000</td>
<td>11100</td>
<td>01001</td>
</tr>
</tbody>
</table>

Parity bit
The second choice is storing characters using an unpacked format. This stores one character per word in the left most byte and fills the rest of the word out with blanks as in figure III.2. This obviously wastes space, but balanced against this is an increased ease and speed of manipulation of individual characters.

PORTACALC has a number of problems relating to the above discussion. The first relates to how the way that variables are declared. A star representation is used such as LOGICAL*1, which is not standard FORTRAN. This usually means that one byte of memory is to be allotted for a LOGICAL*1 variable. The LOGICAL*1 representation allows PORTACALC to use packed and unpacked formats without any explicit change in representation. This works fine on a byte oriented machine such as the VAX assuming that LOGICAL*1 variables are legal in the FORTRAN and have the implementation as described above. Then the system has advantages of both packed and unpacked formats.

When this type of variable is declared on the DEC-20, problems occur. First the DEC-20 ignores the star type declarations, thus a LOGICAL*1 variable is stored in a whole word instead of a byte. This causes problems when attempting to access these variables, since the program expects them
to be stored in a byte. On the DEC-20, explicit changes must be made to change characters from a packed to unpacked format or vice versa. The PORTACALC program does not take this into consideration.

PORTACALC uses a non-standard method of storing and retrieving character information. Sometimes the ENCODE and DECODE statements are used. The ENCODE statements is similar to a WRITE statement except that output goes to a specified array in memory. The string that would be output in the write statement is stored in packed format in the output array. The DECODE statement has a corresponding relationship to the READ statement. The problem with the use of these two statements is that they are not standard FORTRAN IV, thus they detract from PORTACALC's portability.

The use of variables of type BYTE again is a non-standard extentions of FORTRAN IV. This must be changed if the system is to work on a word oriented system. The BYTE variable is used for storage efficiency reasons while lowering the transportability of the system.

Another problem was the use of recursion in the program. Recursion is used when an IF is used in a cell. DOSTMT calls DOIF which recursively calls DOSTMT. On many FORTRAN compilers this will cause a crash of the
program. There are ways of getting around this problem such as limiting
the level of recursion. Then one could have separate copies of each routine
for each level of calls. Each copy would have a different name. When the
DOIF routine is to call DOSTMT, it calls a copy of DOSTMT instead, which in
turn could call a copy of DOIF if necessary.

Another problem is lack of sufficient documentation in some places.
Some of the COMMON variables are not described adequately. This forces
anyone modifying the code to dig to find what the function of the various
data structures are. There is not one location to look for descriptions to the
data structures; instead minimal comments are spread throughout the code.

III. 4. FORTRAN's suitability for a portable program of this type

The authors choice of FORTRAN for the implementation language was
partially for the portability of the system. He stated that since most
systems have a FORTRAN compiler, it would be easily portable. The fallacy
of this argument is that many of the decisions made in the implementation
of the system for reasons of space and efficiency were dependent on a
specific machine or operating system. This took away many of the
advantages of a FORTRAN implementation.
A "standard" version of FORTRAN called FORTRAN IV was defined in 1966 by ANSI, American National Standards Institute. Most computer developers and software developers provide the standard features of this FORTRAN IV as a minimum and then add on special features of their own. These special features may make their systems more powerful and separate their system from other systems. This addition of features can cause problems when one wishes to transport a program to a different machine or a different operating system. If one has portability of the program as a goal, the extended features of the systems should not be used. The special features of FORTRAN IV PLUS are used in PORTACALC, and it does make the resulting program non-portable.

FORTRAN has achieved good portability in applications dealing mainly with numeric computations. One of the problems with FORTRAN IV in problems of this type is that there is no character variable type. In order to handle character manipulations, various methods can be used. These manipulations are possible, since in FORTRAN there is no type checking. This means characters can be stored in something other than a character variable, and the FORTRAN compiler does not complain. These methods used to handle character manipulations are often difficult to understand.
and not usually straight-forward. Some of these methods are machine or operating system dependent. Some of the problems with characters have been corrected somewhat with the addition of character variables in the FORTRAN 77 update of the language.

FORTRAN 77 allows for both character variables and character arrays. With character variables one must specify the maximum number of characters that can be represented. If the number of characters is less than the maximum, the character string is left-justified and the variable is padded with blanks. Even in FORTRAN 77 there are some problems with storage and access of character information. Manipulations of character quantities is somewhat cumbersome because of the restrictions imposed by the fixed number of characters that can be stored in each variable.

With any implementation, there are trade offs of time efficiency, storage efficiency, understandability, and portability. In order to make a more time efficient program, it may take up more memory and may lower the programs portability. If one tries to limit the memory of program, it may reduce the portability of the program and slow its execution time. A very portable program may be slower, larger, and use more memory for data than a non-portable version.
III. 5. How could PORTACALC have been made more portable?

Making PORTACALC truly portable would be a major undertaking. It would require major changes in the program. It would probably be easier to write a complete new version than to modify PORTACALC to be portable. This section describes some of the decisions which would promote a more portable program.

The first and most important thing which needs to be accomplished to a much greater degree is to separate the machine dependent parts of the program into some very well defined modules. Then when the system is transported, one knows exactly which routines must be changed. The rest of the program, which is machine independent, would hopefully work unchanged. Then one would be working on much smaller pieces than the complete program, which makes the problem of fixing up the system much simpler.

There are portable inefficient methods of accessing character codes. PORTACALC uses the packed format to access characters more efficiently. If the goal of the system is to have a truly portable program, then the inefficient portable method should be used. There is a space efficiency trade off. The choice should be made for the most portable version.
With FORTRAN IV, the use of the unpacked format would be better than the packed form used in PORTACALC. Unpacked format relies less on the host machine, and characters can be accessed in a more standard fashion.

If a FORTRAN 77 version were desired, the main alteration would be first not to use any systems dependent features. This might be possible, with the exception of input/output of character data. The second main alteration would be to store characters in character variables instead of logical variables. This would be a much more understandable method than using FORTRAN IV. The straight-forward and obvious method is usually the best.

Another language which might be considered for a portable program is PASCAL, although PASCAL is not without drawbacks of its own. PASCAL's problems include the non-standardization of interactive input/output, PASCAL has no standard way to handle direct-access files. These problems not insurmountable but they are significant.

III. 6. PORTACALC's implementation

Many problems were encountered in the implementation of
PORTACALC. Initially, attempts were made to implement PORTACALC on a VAX under UNIX. UNIX FORTRAN seemed to be quite different from FORTRAN IV which was used in writing PORTACALC. Then attempts were made to implement the system on the DEC-20 under TOPS-20 operating system. These attempts were more successful than the previous efforts, but not completely successful. A reason the efforts on the DEC-20 were more successful was that TOPS-20 FORTRAN uses many of the same non-standard features used in PORTACALC. The system compiled after many changes were made, but soon after starting the run of PORTACALC, the system crashed. There still remained some problems which have not been corrected. They deal with character storage, retrieval, and manipulation problems, and file input/output.

III. 7. Concluding remarks on PORTACALC's implementation

The author of PORTACALC failed to achieve his goals of having a portable program. Machine dependent features are spread throughout the program. This makes it very difficult to transport PORTACALC to a machine other than the one for which it was intended.

FORTRAN on the other hand, does have the capabilities to be portable.
In order for this to happen, the programmer must have the discipline to use only standard and machine independent code. If machine dependent routines are used, they should be small, and more importantly, well defined and documented. Then the porting problems will be decreased. If PORTACALC's goal was to be truly portable, perhaps an inefficient version using only 'standard' FORTRAN should have been supplied.

FORTRAN was originally developed for numerical computation, and thus is not the best at performing manipulations with characters. FORTRAN 77 has much improved facilities for handling characters. Character manipulations can be done in most versions FORTRAN, but they may be somewhat cumbersome. FORTRAN is still often considered as the best means for implementing portable programs. For example, Systems Research Laboratories is selling a LISP system implemented in FORTRAN. It contains a LISP to FORTRAN cross-compiler. This is a very complex non-numeric application which, according to its makers, has achieved portability with the use of FORTRAN.
IV. DESIGN

IV. 1. Overview of Design Section

This section looks at PORTACALC from two points of view. The first describes the tools and methods used to describe the design of the program PORTACALC. The second critiques the design using STRUCTURED DESIGN by Yourdon and Constantine as guide. PORTACALC's design is analyzed with respect to coupling, cohesion, and general structured design principles.

IV. 2. Tools Used to Portray PORTACALC's Physical Design

The program PORTACALC is a relatively large program, with over 12,000 lines of code and over sixty subroutines. The techniques used to describe PORTACALC are structure charts, routine descriptions and a data dictionary of COMMON variables.

Structure charts are used to show the structure or architecture of the system. From a structure chart, one gains a better understanding of the calling hierarchy of the system. Normally, a structure chart will also show the data flow and control flows from one module to the next. This information has instead been included in the subroutine description part of the document. A reason for doing this is to reduce the redundancy of the document.
When describing the design of a system, a goal is to reduce redundancy\(^2\). Thus when changes are made to the design, the documentation needs only to be altered in one place. This is important to keep the documentation accurate and up to date. Documentation is not viable if it is not current.

Each subroutine of PORTACALC was described in a similarly formatted description. The important information dealing with each routine was included. A description of the format is given below.

A short narrative on the purpose of each routine starts the form. The data interface of each routine with the rest of the program is listed. This is followed by the calling information.

The data interface or input/output arguments of a routine hold various types of information. Two types of arguments were noted, explicit arguments, and implicit arguments. Explicit arguments are those variables which are passed into and out of a routine through the parameter list. The implicit arguments are those which are implicitly passed by use of common blocks. The implicit arguments are not always used, and this information is also included. Both types of arguments are also organized by whether information was being passed into the routine or out of the routine or both
in and out the routine.

Three pieces of calling information are included. First, the information of how the routine is to be called. This would include the explicit arguments, thus showing how many and the order of the parameters. A routine's place in the calling hierarchy is noted by the answering of the following questions: "What routines call it?" and "What routines it calls?" This information is essential in understanding the effects of changes on the rest of the system.

A data dictionary of COMMON variables is included. This is organized by COMMON area. The type of the each variable is given. A short description of what it is used for is presented. Each routine where it is used is listed. The information of whether it is modified is given in the routine description, so it need not be included in the data dictionary.

IV. 3. Design Process

The design and analysis process both took a great deal of time and effort. Sometimes the efforts did not make any headway and this was very frustrating. Originally attempts were made to gain an understanding of the system by reading the code. This often left me against a brick wall,
for the intricacies of the code were not always straightforward. Progress was made when a global view of the system was strived for. This helped to break down the system into smaller pieces to attempt to understand. The documentation was often lacking, especially in reference to common variables and exactly what routines were to do. Some of the common variables were well described but the documentation that existed was scattered throughout the code.

Attempts were made to implement PORTACALC in hopes that a running program would increase the understanding of the design of the program. These attempts failed but my understanding of the system was increased through these efforts.

RXVP80™ was an immense aid in documenting the design of the system. More details are presented on RXVP80™ in appendix H. RXVP80™ listed many of the interrelationships of the system. It was especially an helpful in finding the data connections of the system. The process of writing the routine descriptions of the system promoted a clearer conception of the program. The routine descriptions and common data dictionary were an invaluable resource when creating the logical view of the system using data flow diagrams.
IV. 4. Important Data Structures of the System

The data structures which must be understood to follow the program are described below. The formulas of a system are stored in a direct access file. (LUN7 is used) This file can be read from or written to. For each cell of the spreadsheet, there is room for a formula or value, the display format, and a number of flags to check validity and other features. The coordinates of the physical sheet are used as a indices into this file. The physical sheet is the complete spreadsheet with the size equal to the maximum number of columns and rows. The calculated values of this physical sheet are stored in a two dimensional array (VBLS). Another two dimensional array stores the type of the value (TYPE). A third array flags what is in each cell, a formula, a constant, or nothing (FVLD).

There is another sheet, called the display sheet, which is a part of the physical sheet. It is sized so that is will fit on a screen. The display sheet is mapped back onto the physical sheet. The arrays NRDPS and NCDSP give the correspondence back into the physical sheet. DROW and DCOL are used to move in the display sheet.

There are also 27 accumulators or global variables. These can be used
in formulas of the spreadsheet. In PORTACALC, they are labeled by the letters A to Z with % completing the list. The % accumulator always contains the last value calculated.

There are other variables but these are the essential ones for understanding the program. More information on these variables and their use is given in the common data dictionary in Appendix C.

IV. 5. Critique of PORTACALC's design

In this section, a critique of PORTACALC's design will be given. Yourdon and Constantine's Structured Design\(^{10}\) will be used as a frame of reference. PORTACALC's design will be investigated in terms of module size, cohesion and coupling of modules.

PORTACALC is a big program with over 12,000 statements. It is broken down into over sixty modules, which is a step in the right direction. Many of the modules do one task, and are easy to understand. Some modules are large and almost impossible to follow. An example of a large module is the subroutine XQTCMD. XQTCMD has 1647 lines, which is not easily understood or modified, due to the number of lines.

Two criteria which can be used to discuss the quality of a design are
coupling and cohesion. Coupling is the degree to which the modules of the system are interrelated, and minimal coupling is desired. Cohesion is the functional relatedness of the parts of the same module; or how tightly related the elements of modules are to one another. Modular cohesion can be described in relative terms. Yourdon and Constantine\textsuperscript{10} have broken down cohesion into seven levels: coincidental, logical, temporal, procedural, communicational, sequential, and functional. The last three levels are the most desirable and have the highest level of cohesion.

With respect to cohesion, the designers of PORTACALC did a good job on some modules, and less so with others. Routines such as BASCNG, which change a digit from one base to another, and GETNNB, which get the next non-blank element from LINE, have functional cohesion. This means the routine does one task, and all elements of the routine are related to that task.

Modules like XQTCMD, which is the process command module has many facets. It has procedural or communicational cohesion. XQTCMD has communicational cohesion since the parts of routine act on the same input, and procedural cohesion since the unit is a decision process. Cohesion of this routine could be improved. Some of the pieces could be taken out and
put in their own modules, which would help to shorten the XQTCMD routine. The commands which perform related tasks could be put into a lower level subcommand module. This process would also strengthen the cohesion of XQTCMD and the cohesion of the new modules. In sections V.3 of the Analysis section, a more detailed version of how this could be done is given.

Coupling of a system is related to cohesion, but is not as easy to quantify. There are four major factors which influence coupling, they are expressed by Yourdon and Constantine\textsuperscript{10} as follows:

1. Type of connection between modules; a minimally connected system is desirable, where the lowest number on interconnections and interfaces with which the system can work effectively
2. Complexity of interface between modules, how many and how complicated is the data being transferred between modules
3. Type of information flow along the connection.
4. Binding time of the connection.

One of the factors which make PORTACALC more difficult to understand is the common-environment coupling. Modules are interacting through common data elements. Though they are related through this
common data, they may not be logically related. This problem is somewhat improved by the use of labeled common blocks, but still the problem persists. Many modules have access to the same common blocks which makes it difficult to follow how data is flowing through the system and just how the modules are inter-related. As Yourdon and Constantine\textsuperscript{10} stated,

"A small number of elements shared among a few modules can enormously complicate the structure of a system - from the point of view of understanding it, maintaining it, or modifying it."

The complexity of the interface between modules of PORTACALC is high. Complexity is measured in number of pieces passed between modules. Many of the modules have many more than fifteen distinct variable arguments. This is a great deal of information to pass into the modules and increases the coupling of the system.

Another measure of complexity is how the straightforward the interface is. The complexity of the interface between modules is increased by the use of COMMON blocks. The FORTRAN interface has two types of arguments, explicit through a parameter list and implicit arguments, through a COMMON block. The implicit arguments increase the complexity of the interface, because they are not as straight-forward. Also the elements of the COMMON blocks are not always all used, which adds
unnecessary coupling.

Control coupling as well as data coupling are found between parts of the PORTACALC system. Control coupling increases the interdependence of the system. Data coupling is essential to have a working system; without it pieces of a systems have no way to communicate.

PORTACALCs design gets the job done, but has much room for improvement. The modules could have been organized so that there is stronger cohesion. In doing this, some of the large routines would be broken into several smaller, more cohesive routines which would be much easier to understand and modify. As the program is designed, the large modules have direct connection with too many other modules. A certain amount of coupling is needed between the modules of the system, although the amount and complexity of the coupling could be reduced significantly.
V. ANALYSIS

V. 1. Analysis tools

The analysis process gives a logical specification of the program. This specification shows what the system actually does by presenting the functions of the system. The analysis does not show the particulars of how the system performs the functions, rather just what they are.

Three tools are used to represent the logical picture of the system. Data flow diagrams, a data dictionary, and mini-specifications of the processes of the system. These tools are presented in DeMarco's *Structured Analysis and System Specification*.

The use of data flow diagrams is a powerful tool in understanding and specifying a system. Data flow diagrams graphically show how the data flows through a system. When one understands how the data moves through a system, a better understanding of the system is gained. The graphic representation of data flow diagrams displays a visual picture of the system, and portrays a large volume of information rapidly. The data flow diagrams show the whole system at once, and show how it is connected together. Data flow diagrams also allow decomposition of the system to show the various parts in more detail.
Data flow diagrams are made up of four main elements, data flows, processes, files, and data sources and sinks. Each of these elements has a geometric representation. Data flows are shown by named vectors, processes are shown by bubbles, files are shown by straight lines, and data sources and sinks are represented by boxes.

A simple data flow diagram is shown in Figure V.1. The USER, illustrated with a box, is a source of information for the system. The SCREEN, which is a sink or destination of information from the system, is
also shown with a box. The COMMAND INDEX FILE, shown with a straight line, is a file which is accessed by the COMMAND EXECUTER process. The data flow to and from the COMMAND INDEX FILE is the information which makes up the file. Two data flows on the system are the COMMANDS and DISPLAY INFORMATION. The COMMAND EXECUTER process, shown with a circle on the diagram, accepts COMMANDS and uses the COMMAND INDEX FILE to generate DISPLAY INFORMATION. This process could be broken down into another data flow diagram which would give more detailed information about the transformation occurring in the process. In this way, complicated parts of the system can be partitioned into understandable pieces.

Data flows are pathways between processes, which have a known composition. These data flows may be as simple as an integer or boolean value, or as complicated as a multi-dimensional record of information. Data flows are not used to show flow of control, just data. This helps simplify the view of the system, and leaves the design and algorithmic decisions to the designer. Then each data flow is explicitly defined in a data dictionary.

The processes of a system, signified by bubbles on a data flow diagram, show where actions are performed on the data. A process
transforms incoming data into outgoing data. The name of the process should accurately describe what action is performed in the process.

A file is a place to temporarily store information. This could be a tape, or a special place in the memory of a computer. Data can flow into and out of a file.

Sources and sinks are outside of the boundaries of the system. They are the net originator or receiver of the system's data. Sources and sinks are shown on the data flow diagram with a box.

The two associated tools which go hand in hand with data flow diagram are mini-specs and data dictionary. A data dictionary allows the data flows and files of a system to be defined. The data flows can be built from smaller components which are also defined in the data dictionary.

When a bubble or process is not broken down any further a mini-specification is written for that process. The mini-specification or mini-spec is a description of what happens in the process. A common way of specifying this is structured English³ or a combination of the control structures of structured programming and straightforward English.
V. 2. Descriptions of system

This section gives a narrative description of the spreadsheet program. First is a discussion of the top level interfaces of the system. This is followed by a discussion of Level 0 on the data flow diagram.

The spreadsheet program interfaces with five sources or sinks outside the system. Commands come into the system either from the keyboard or from a command file. The keyboard and the command file are strictly sources of information. The commands which can be used are described in XQTCMD SUBROUTINE COMMAND and in CMND SUBROUTINE COMMANDS in Appendix G. The program works on a command verb syntax. This means all user input to the system is in the form of a command followed by its arguments.

An existing spreadsheet, stored in a file, is both a source and a sink of information of the system. The old cell contents go into the system from the file and updates of cell contents are returned to the file. The cell contents are the elements which are associated with each cell on the spreadsheet. Cell contents include the formula to be evaluated in the cell and the value of the cell. The display format and some flags are also included in the cell contents. This file is often accessed by the program.
A new spreadsheet file with either formulas or values can be created by the spreadsheet. This information can be returned to system at a later time.

The screen is a sink for information from the system as well. Screen codes, which perform terminal screen manipulations, are included in the data flow to the screen. Manipulations include erasing all or part of the screen, setting the mode of the terminal screen, and movement of the cursor on the screen. A cell's contents are also sent to and displayed on the screen.

The level 0 data flow diagram depicts how the system fits together and works. There are five main functions at this level: COMMAND PROCESSOR, CALC, TRANSLATOR OF VIDEO COMMANDS, RECALC, AND DISPLAY SHEET.

COMMAND PROCESSOR is the input command processor. It corresponds to the XQTCMD subroutine in PORTACALC. The module interprets PORTACALC commands and either performs the appropriate action itself or calls a corresponding module to complete the requested action. It would perform actions such as setting up the display sheet, also building and altering the contents of the cells of the sheet. It calls the other routines to
perform calculation, recalculations, and displaying of the sheet.

The CALC module is an arithmetic expression evaluator. It receives an expression as input and yields the value of the expression as output. It has access to the cell values and can modify cell values.

RECALC is the recalculation or update module. This is used when an update is required. Recalculation can be set up to be done after each command automatically or just upon request. The latter mode would be used when setting up the sheet, so the time would not be wasted on recalculation after each entry. The recalculation is performed on each cell of the sheet where a formula is available for evaluation.

DISPLAY SHEET is used to update the display of the spreadsheet. Only part of the total spreadsheet is normally seen, since the whole spreadsheet is usually too large to all fit in on the screen. Flags determine what to display for each cell.

TRANSLATE VIDEO DISPLAY COMMANDS is just a translator. It accepts a code for a screen manipulation and translates the code into an escape sequence which causes the terminal screen to act as directed.
V. 3. Description and modularization of command processor routine.

The COMMAND PROCESSOR ROUTINE or XQTCMD module processes the PORTACALC commands of the system. As it is written, it does too much of the work in one very large module. A possible remedy of this situation would be to farm out some the processing done in XQTCMD to lower level subcommand modules. These subcommand modules would be organized with commands doing similar type of operations being handled by the same subcommand module. This organization would put together commands which access many of the same variable.

A possible way to modularize this routine is described below. There are sixteen types of commands processed by XQTCMD. They are described in more detail in XQTCMD SUBROUTINE COMMANDS in appendix G. Six subcommand modules would be set up to handle all the commands. The six groupings of commands would be recalculate commands, sheet organization commands, cell set up commands, cursor position commands, save/restore commands, and exit/help commands.

The recalculate commands (R) would be in their own processing module, and would handle the recalculation and set up flags for determine when to recalculate.
Sheet organization commands would be put in a module. This module would handle three types of commands:

(Z) Zero commands, which will zero out all or part of the spreadsheet

(D) Display formatting commands, which are used to change formats, and other display characteristics for cells and screen

(V) Viewscreen update commands, which determine whether to display normal contents of cells or just formulas, and whether to do an update of the screen

Cell set up commands are used to enter and alter contents of cells on spreadsheet. Commands that would be included in this module are listed as follows:

(E) Enter commands, which are used to put formulas or values into cells

(C) Copy commands, which are used to replicate formulas or values of other cells.

(A) Add/remove row/column commands, which add columns or rows to spreadsheet, or remove columns or rows.
Another routine would handle movement on the spreadsheet. It would handle four commands.

(#{) Move direction commands, which with numbers one can move up, down, right and left

(L) Move to cell given by a variable.

(O) Move origin of the display sheet, this move the whole window of the display sheet around on the physical sheet.

(M) This sets the default for motion after an enter command.

A module to handle commands for saving and restoring the spreadsheet would be included in a module. The three following commands would be:

(P) Put or write out formulas or values from the spreadsheet to a file.

(S) Save work file

(G) Get values or formulas from a file.

A final module would be used to process exit and help commands. The module would handle three commands.

(H) Prints out help information to the screen.
(K) Drop into the calculator mode, CALC bare, gives the user direct control of the CALC program

(X) Exits the program to the operating system.

This organization is one way one could modularize the command processor, and it does group together those commands which perform similar tasks. Also the commands as grouped require many of the same variables to access and modify, which simplifies the data interfaces into the individual modules. In other words, the coupling between the modules would be reduced. This organization of grouping would also increase the cohesion of the modules, since commands being processed in each process are related tasks accessing similar data.

A goal of the organization of any system is to create the best program structure possible. The organization described in this section is an improvement over the current organization of PORTACALC. It breaks this part of the system into more cohesive modules with less coupling between the modules.
VI. CONCLUSION

VI. 1. Reverse Engineering and Implementation

This project has been a good learning experience, as it has given me an opportunity to use many of the tools and techniques learned in my graduate courses. The best way to gain a better understanding of tools and techniques is to use them, as I did in working on this project.

A reverse engineering project is part of a maintenance effort. Today much of the cost of software goes to the maintenance process. Using reverse software engineering techniques can hopefully help slow the increase in maintenance costs.

Maintenance programming is not an easy task. Before any maintenance work can be done, one must gain an adequate understanding of the system. This requires the skill of being a good program reader. Reading and understanding others' code is never an easy task, but it can be made easier if there is adequate documentation in the program and if the program has been written in a structured manner. Tricks used in writing code make the maintenance programmers job much more difficult.

The process of reverse engineering takes an existing program, and from it develops a program specification and functional description. These
documents describe what the system does and how the system performs its functions. These documents are invaluable to the maintenance programmer, by giving him/her an overall view of the system, as well as its data structures, modules, and interfaces between modules. This information is essential if the maintenance programmer is to modify the program and keep the program's integrity intact as much as possible.

The reverse engineering process was successful to a point. The process helped me gain an understanding of the system. Much initial effort was put into getting a system running in hopes that a running system would be more easy to comprehend, since a running system would increases the depth of understanding of the system. This process would have worked well except for the fact that the system never did run.

With the documents produced by the reverse engineering effort, the implementation would probably have been more successful. The understanding of what is happening in the system would be much easier. Time spent in figuring out pieces of code would not be as important, since one would have a picture of the program, with its data structures and routines. One could then concentrate effort on the problems of transporting the program instead of comprehending the program.
The reverse engineering effort would not solve all the problems. In this case, an in depth knowledge of FORTRAN, and in particular, DEC's various implementations of FORTRAN, is necessary to solve all the problems relating to this program. Many of the problems that were encountered in the process of implementing and understanding PORTACALC, were problems in understanding how the authors implemented various tasks in FORTRAN. The authors of PORTACALC used non-standard extensions of FORTRAN which were not adequately documented.

PORTACALC was written to be portable, but it is not easily portable to a DEC-20 under TOPS-20 or a VAX under UNIX. The main portability problems of PORTACALC are its use of non-standard extensions of FORTRAN and its method of storing data. The design of PORTACALC could be improved as well. The common environmental coupling, which increases the complexity of the system could be reduced. The system dependent parts of the system need to be organized in well defined small pieces of the system, instead of spread through out the program.
VI. 2. Critique of tools used

At this point a discussion on the tools and techniques that were used in this project is given. The structure charts and the routine descriptions were very helpful in getting an overview of the system. They were a good resource when the data flow diagrams and data dictionary were made. A problem with the structure charts is that at times they didn't convey the accurate meaning of the system. An example is the top level structure chart. There are a number of modules called by the main program which are not very significant, but are shown on the structure chart with the most important subroutines. When a number of tasks are performed in a single module, the various tasks are not portrayed in the structure charts., which is mostly due to the unbalanced design of PORTACALC.

Another problem with the normal structure chart was the interface between many modules was very complex. In PORTACALC, this points to a problem with the design for there was much information being passed between modules. A chart with all of the information being passed would have been unreadable. This information instead is given in the routine description section of the design. This eliminates some of the redundancy, which makes it easier to keep the design document up to date. On the
other hand, the method used is not as easily used for quick reference.

The data flow diagrams on the other hand showed the system from a more abstract and logical point of view. In other words, it shows what the system is doing, not how the particular system does the tasks the system performs. The data flow diagrams better portray the system and can be understood by people who are not computer professionals. One could, in a period of time, gain a reasonable understanding of the functions of a system by studying the data flow diagrams of that system. The data flow diagrams capture the actions of the system by following the data. There is a correspondence between the data flow diagrams and the structure charts, although it is not a one-to-one correlation. The routines which perform many tasks are broken down in the data flow diagrams. At times, more than one routine was put together in a single process on the data flow diagrams. Again data flow diagrams give a more accurate picture of the system, and this would be especially true if one was going to build a new system with the same function.

If this whole process was to be performed again, there are some things which would be done differently. Since data is the key to any system, an accurate description of all global data would be made initially.
This would facilitate more rapid comprehension of the system as a whole. What actions are to be performed on this data, and how it is to be modified would be more explicitly stated.

Depending on the size of the system, it would also be feasible to make a giant data flow diagram of the system, which would show the interrelationships of the program. The data flow diagrams are a powerful tool when used along with the data dictionary and mini-specifications. They give a very good correspondence to the actual structure of a system.

The most important thing that would be done differently would be to document the pieces of the actual system in a more organized manner from the start and to more carefully document my actions dealing with the system. With a larger base of information pertaining to the system, evaluation and further analysis would be more easily accomplished.

Transporting a system has some special problems. A thorough knowledge of FORTRAN and its implementation is essential. This is especially true since FORTRAN has the flexibility do many things that it was not initially built for. One reason that FORTRAN has this flexibility is that it does not have strong type checking facilities. This allows a programmer to store data however he/she wishes, for example characters
can be stored in logical variables. These extensions are often not portable, which means that the transporter must figure out what task is being done and then find a way to implement that task on the new host system. There is a problem relating to the process figuring out exactly what is being accomplished in a particular piece of code. When a problem in understanding code is encountered, textbooks are not usually helpful as only the basics of FORTRAN programming are given. With practice and work coding a particular FORTRAN on a particular operating system, one acquires knowledge of the tricks and methods used in the implementation of FORTRAN programs. One of the best resources is a person who has experience in working with FORTRAN programs on your particular system. In other words, there is no substitute for experience.

There are also problems with understanding the interface of FORTRAN with the particular operating system. Each operating system has peculiarities of its own, that must be comprehended by the transporting programmer. System calls and how input and output are handled are examples of systems dependent information.

Software engineering tools and methods are guidelines to use in the process of development of software. The reverse engineering process
extends the use of some of the software engineering techniques to the maintenance and redevelopment process. These are not recipes to follow, but rather guidelines to be used as an aid in the process. As one gains experience, these tools and methods can be more efficiently applied to the project at hand.
## APPENDICES

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Structure Charts</td>
<td>62</td>
</tr>
<tr>
<td>B</td>
<td>Subroutine Descriptions</td>
<td>70</td>
</tr>
<tr>
<td>C</td>
<td>Common Data Dictionary</td>
<td>146</td>
</tr>
<tr>
<td>D</td>
<td>Data Flow Diagrams</td>
<td>159</td>
</tr>
<tr>
<td>E</td>
<td>Data Dictionary</td>
<td>164</td>
</tr>
<tr>
<td>F</td>
<td>Mini-Specifications of the Processes in Data --</td>
<td>173</td>
</tr>
<tr>
<td></td>
<td>Flow Diagrams</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>Command Descriptions</td>
<td>177</td>
</tr>
<tr>
<td>H</td>
<td>RXVP80™, A Software Analysis, Documentation, and</td>
<td>185</td>
</tr>
<tr>
<td></td>
<td>Test System</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX A.

STRUCTURE CHARTS
Top Level Structure Chart

SPREADSHEET - Spreadsheet main program

UVT100 - Terminal driver program
TTYINI - Terminal or system dependent initializations
RECALC* - Spreadsheet recalculation program
ASSIGN - Assigns file to LUN
DSPSHT* - Display spreadsheet to file or screen
HELP - Prints help information
XQTCMD* - Execute command processor
CLOSE - Close LUN

Routines with * are broken down with structure chart of their own.
CALC Structure Chart Descriptions

Calc - Evaluates arithmetic expressions

ASSIGN - Assigns label to LUN
CLOSE - Close LUN
CMND - Execute \* commands in CALC
ERRCX - Performs initial syntax checking on input line
ERRMSG - Prints out appropriate error message
INPOST - Converts input string to postfix notation
MOUT - Output multiple precision variable
NEXTEL - Returns the next element for LINE.
POSTVL - Evaluates postfix expression on stack
PRTCON - Converts zero to appropriate number in
SLEND - Finds the last non-blank in LINE
VAROUT - Output variable
ZNEG - Determines if variable is zero, negative or undefined as opposed to positive.
CMND Structure Chart Descriptions

CMND - Execute * commands in CALC

ASSIGN - Assigns label to LUN
BASCNG - Changes the default base for constants.
CLOSE - Close LUN
DECLR - Analyzes line to determine what
variables get their types changed.
ERRMSG - Prints out appropriate error message
GETNNB - Get next non-blank in LINE
STRCMP - Compares two strings for match
ZERO - Zeros out all variables
ZNEG - Determines if variable is zero, negative or
undefined as opposed to positive.
DSPSHT Structure Chart

DSPSHT Structure Chart Descriptions
DSPSHT - Displays spreadsheet to screen or file

UVT100 - Terminal driver
IN2AS - Convert row numbers to letter labels
POSTVL Structure Chart Descriptions

POSTVL - Evaluates postfix expression from stack

CALBIN - Performs binary operation
CALUN - Performs unary operations
CONTYP - Performs constant type conversion
ERRMSG - Prints appropriate error message
MOUT - Outputs multiple precision variable
MULADD - Multiple precision addition and subtraction
MULCON - Converts multiple precision number to new base
MULDIV - Performs multiprecision division
MULMUL - Performs multiprecision multiplication
VAROUT - Outputs variable
RECALC Structure Chart Descriptions

RECALC - Recalculates spreadsheet

DOENTR - Checks for multiple statements per line
DOSTMT - Parses one statement
INDEX - Finds location of character in string
FNAME - Searches LINE for function names
DOMFCN - Performs math functions, gets all information
DOIF - Handles if in cell of formula
CALC - Evaluates arithmetic expression
MTHINI - Initializes before math functions performed
VARSCN - Converts row letter labels to numbers
DOMATH - Does actual math functions
GETLOG - Checks logical type relationships of two variables
TEST - Test logical relations of statements for use with IF statements
XQTCMD Structure Chart Descriptions

XQTCMD - Command processor

ASSIGN - Assigns label to LUN
CLOSE - Close LUN
CALC - Evaluates arithmetic expressions
CMDMUN - Munges command line to allow special keys to be recognized.
DSPSHT - Displays spreadsheet to screen or file
GN - Get number
HELP - Prints help information
INDEX - Finds location of character in a string
IN2AS - Convert row numbers to letter labels
ISGN - Determines if argument is positive, negative, or zero
RELVBL - Relocate variables
SED - String edit routine
SSCMP - Compares two arrays
UVT100 - Terminal driver
VARSCN - Converts row letter labels to numbers
APPENDIX B.

SUBROUTINE DESCRIPTIONS
Subroutine AT

**PURPOSE:** When the `calc command` is encountered, the subroutine AT is called. It changes the value of the level which holds the number of the logical I/O unit where input command lines are to be obtained. The file associated with that I/O unit is opened under proper conditions.

**DATA INTERFACE I/O**

**Explicit Arguments**
- **IN:** NONE
- **OUT:**
  - `RETCD` - return code, 1=ok, 2=error

**Implicit Arguments (through common)**
- `COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED`
- `COMMON /CONS/ ALPHA, COMMA, BLANK, RPAR, LPAR, EQ`
  - CONS are constant commons.
- `COMMON /ITERA/ITCNTV`
  - **IN:**
    - `LINE` - (i*2) - Holds command input line
  - **OUT:**
    - `NONBLK` - (i*2) - Points to the last non-blank character in line(80)
    - `ITCNTV(6)` - Indexed by level, holds 0 if no iteration on that level, otherwise index into VBLS for the variable that controls iteration.
  - **IN/OUT:**
    - `LEVEL` - (i*2) Holds number of logical i/o unit where next input line is expected
  - **NOT USED:**
    - LEND, VIEWSW, BASED,
  - **CONSTANTS USED:**
    - ALPHA, BLANK
  - **CONSTANTS NOT USED:**
    - COMMA, RPAR, LPAR, EQ

**HOW ROUTINE IS TO BE CALLED**
- `CALL AT(RETCDE)`
AT routine

WHAT ROUTINES CALL IT?
CMND

WHAT ROUTINES IT CALLS?
ASSIGN
ERRMSG
GETNNB
ZNEG
**Subroutine BASCNG**

**PURPOSE:** Called by CMND, when the "B calc command is encountered. This command indicates that the default base for constants is to be changed. The routine reads in one or two digits and changes the default base specification as appropriate.

**DATA INTERFACE I/O**

**Explicit Arguments**

- **OUT:**
  - RETCD - 1 = o.k., 2 = error

**Implicit Arguments (through common)**

**COMMON** LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED

**COMMON /DIGV/ DIGITS**

**IN:**
- LINE - Command line

**OUT:**
- NONBLK - Points to the last non-blank in line(80)
- BASED - Holds the default base

**NOT USED:**
- LEVEL, LEND, VIEWSW

**CONSTANTS USED:**
- DIGITS - Digits for each of the bases 8, 10, 16.

**HOW ROUTINE IS TO BE CALLED**

CALL BASCNG(RETC)D

**WHAT ROUTINES CALL IT?**

CMND

**WHAT ROUTINES IT CALLS?**

ERRMSG
GETNNB
Subroutine CALBIN

**PURPOSE:** Performs a binary operation on two constants.

**DATA INTERFACE I/O**

**Explicit Arguments**

- **OUT:** RETCD -
  - 1 - Normal return
  - 2 - Operation complete (result has been output)
  - 3 - Error return

**Implicit Arguments (through common)**

- COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED
- COMMON /V/ TYPE, AVBLS, VBLS, VLEN
- COMMON /STACK/ STACK1, STACK2, ST1PT, ST2PT, ST1TYP, ST2TYP
- ST1LIM, ST2LIM
- COMMON /DECIDE/ DTBL1

**IN:**
- VLEN, ST1PT, ST2LIM

**IN/OUT:**
- STACK1, STACK2, ST2PT, ST1TYP, ST2TYP

**OUT:**
- AVBLS, TYPE, VBLS

**NOT USED:**
- LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED, ST1LIM

**HOW ROUTINE IS TO BE CALLED**

**CALL CALBIN(RETCD)**

**Conditions:**
- In: Operand 1 is in STACK1 (ST1PT-1)
- Operand2 is on top of STACK2 (ST2PT-1)
- Operator is below operand2 (ST2PT-2)

- Out: result is in STACK1
- STACK2 has been cleaned up.

**WHAT ROUTINES CALL IT?**

- POSTVL

**WHAT ROUTINES IT CALLS?**

- CONTYP    ERRMSG    MULADD
- MULDIV    MULMUL

**CALBIN routine**
Subroutine CALC

PURPOSE: This program evaluates arithmetic expressions input to it and allows variables to be assigned value. It features multiple precision arithmetic in base 10, octal, and hexadecimal. See CALC.MEM for a complete description in the form of a users guide. Type ? to obtain a list of possible commands.

DATA INTERFACE I/O

Explicit Arguments
IN: None
OUT: None

Implicit Arguments (through common)
COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED
COMMON /OAR/ OSWIT, OCNTR, OARRY
COMMON /ILN/ ILENFG, ILNCT,ILINE
COMMON /KLVL/ KLVL
COMMON /ITERA/ ITCNTV
COMMON /CONS/ ALPHA, COMMA, BLANK, RPAR, LPAR, EQ
COMMON /DIGV/DIGITS
IN:
DIGITS - Constant which holds digits for base 8, 10, or 16
ILINE(106) -(LOG*1) - This is put into LINE(80) without nulls
ILNCT - (1*2) - Length of line
ILNFG - (1*2) - Flag 0 all Is well
ITCNTV(6) - (1*2) - Indexed by level. 0 indicates that no iteration on the indirect command file is to take place. If positive, it holds the index into VBLS and represents the variable used to control iteration. This variable is guaranteed to be 1-27.
KLVL - (1) - IF KLVL = ONE, LEVEL is set to KLVL
LEND - (1*2) - Length of LINE

OUT:
NONBLK- (1*2) - Points to the last non-blank character in LINE(80)
CALC routine

IN/OUT:
- LINE(80) - (i*2) - Command input line
- LEVEL - (i*2) - Holds the logical i/o unit where the next calc command lines come from.
- VIEWSW - (i*2) - View switch, 0= output error messages
  1= Output error messages and file command lines
  2= Output error and value of expressions evaluated
  3= Output everything

NOT USED:
- BASED

CONSTANTS USED:
- ALPHA, BLANK

CONSTANTS NOT USED:
- COMMA, RPAR, LPAR, EQ

HOW ROUTINE IS TO BE CALLED
CALL CALC

WHAT ROUTINES CALL IT?
- DOSTMT XQTCMD

WHAT ROUTINES IT CALLS?
- ASSIGN CLOSE
- CMND ERRCX
- ERRMSG GETMCR
- INPOST LIST
- MIN0 POSTVL
- SLEND VAROUT
- ZNEG
Subroutine CALUN

PURPOSE: Performs a unary operation
Upon entrance: operator is on stack 2
operand is on stack 1
Upon exit: operator has been popped off stack 2
result is on stack 1

DATA INTERFACE I/O
Explicit Arguments
IN: NONE
OUT:
    RETCD - (i*2) - 1=ok, 2= error

Implicit Arguments (through common)
COMMON /STACK/ STACK1, STACK2, ST1PT, ST2PT, ST1TYP, ST2TYP,
    ST1LIM, ST2LIM
IN: NONE
IN/OUT:
    ST1PT - (i*2) - Points to top of stack 1
    ST1TYP(40) - (i*2) - Type for each element on stack 1
    STACK1(100,40) - (LOG*1) - Holds operands
    ST2PT - (i*2) - Points to top of stack 2
    ST2TYP(40) - (i*2) - Type for each element on stack 2
NOT USED:
STACK2 holds operator but is not accessed in this routine.

HOW ROUTINE IS TO BE CALLED
CALL CALUN(RETCDO)

WHAT ROUTINES CALL IT?
POSTVL

WHAT ROUTINES IT CALLS?
CONTYP  DABS  DATAN  DCOS
DEXP    DLOG  DLOG10 DSIN
DSQRT   DTANH ERRMSG FLOAT
IABS    IDINT
Subroutine CMDMUN

PURPOSE: Munges up command lines passed in argument to allow special keys to be recognized. Default version just returns.

DATA INTERFACE I/O
Explicit Arguments
IN: NONE
OUT: NONE
IN/OUT:
   LINE(80) - (LOG*1) - Line to be munged up.

Implicit Arguments (through common)
NONE

HOW ROUTINE TO BE CALLED
CALL CMDMUN(LINE)

WHAT ROUTINES CALL IT?
XQTCMD

WHAT ROUTINES IT CALLS?
NONE
Subroutine CMND

PURPOSE: This routine is called by CALC who has identified the '*' indicating a command is desired.

Upon entrance:
NONBLK points to the '*' in LINE indicating a command. This routine determines which command is desired and calls the appropriate subroutine.

DATA INTERFACE I/O

Explicit Arguments
IN:
NONE
OUT:
RETC 1 = Normal,
2 = Bypass next read because read command has been executed to change LINE(80)
3 = Error, so go to LEVEL=1

Implicit Arguments (through common)
COMMON /IOLVL/ IOLVL
COMMON /DCTL/ PROW, PCOL, DROW, DCOL, DRWV, DCLV
COMMON /V/ TYPE, AVBLS, VBLS, VLEN
COMMON /FVLDC/ FVLD
COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED
COMMON /ITERA/ITCNTV
COMMON /DIGV/DIGITS
IN:
ITCNTV(6) - (i*2) - 0 if no iteration. If positive, index of variable used to control iteration on that level.
OUT:
FVLD(RRW,RCL) - (LOG*1) - Valid flag for cells
IN/OUT:
AVBLS - (100,27) -(LOG*1), equivalenced to XAC as follows
EQUIVALENCE (XAC, AVBLS(1,1,1))
IOLVL - (i*2) - Used to set and reset LEVEL
CMND routine

LEVEL - (i*2) - Holds logical i/o unit where next command comes from.
LINE(80) - (log*1) - Holds command line.
NONBLK - (i*2) - Pointer for LINE(80)
PROW, PCOL - (i*2) - Physical row and column
TYPE(RRW,RCL) - (i*2) - Type of content of cell
VBLS(8,RRW,RCL) - (log*1) - Equivalenced to JVBLS and XVBLS as follows:
  EQUIVALENCE ( VBLS(1,1,1), JVBLS(1,1,1))
  EQUIVALENCE ( VBLS(1,1,1), XVBLS(1,1,1))

NOT USED:
  DROW, DCOL, DRWV, DCLV, VLEN, LEND, VIEWSW, BASED

CONSTANTS USED:
  DIGITS - Constant that holds digits for bases 8, 10, 16.

HOW ROUTINE IS TO BE CALLED
CALL CMND(RETCDE)

WHAT ROUTINES CALL IT?
CALC

WHAT ROUTINES IT CALLS?
  ASSIGN  AT
  BASCNG  CHAR
  CLOSE   DECLR
  ERRMSG  GETNNB
  INDEX   MAXO
  MINO    SCMP
  STRCMP  USRFCT
  VARS CN  ZERO
  ZNEG
Subroutine CODES

PURPOSE: This file gives the variable and function code. Table also gives compare values and stack value of functions that occur when expressions are evaluated. This file contains no compilable code.

DATA INTERFACE I/O -NONE

Explicit Arguments
NONE

Implicit Arguments (through common)
NONE

HOW ROUTINE TO BE CALLED
NOT CALLED

WHAT ROUTINES CALL IT?
NONE

WHAT ROUTINES IT CALLS?
NONE
Subroutine CONTYP

**PURPOSE:** Converts constant in STACK(1,INDEX) from OLDTYP to NEWTYP. If OLDTYP.EQ.NEWTYP a return is made immediately. Note that TYPE(INDEX) is not change by this routine.

**type codes:**
- 0 NO CHANGE
- 1 ASCII
- 2 DECIMAL
- 3 HEXADECIMAL
- 4 INTEGER
- 5 M10
- 6 M8
- 7 M16
- 8 OCTAL
- 9 REAL

**DATA INTERFACE I/O**

**Explicit Arguments**

**IN:**
- INDEX - (i*2) - Pointer to variable being converted.
- OLDTYP - (i*2) - Data type of the variable to be converted.
- NEWTYP - (i*2) - New data type requested.

**OUT:**
- RETCD - 1=OK, 2 = ERROR

**IN/OUT:**
- STACK(100,40) - (log*1) - Holds variable to be converted.

**Implicit Arguments (through common)**

NONE

**HOW ROUTINE IS TO BE CALLED**

CALL CONTYP(STACK, INDEX, OLDTYP, NEWTYP, RETCD)

**WHAT ROUTINES CALL IT?**

CALUN CALBIN

**WHAT ROUTINES IT CALLS?**

ERRMSG MULCON
Subroutine DECLR

PURPOSE: Analyzes vector line to determine what variables get their types changes. The new type is specified as an argument in the call.

Type code
1 ascii
2 decimal (real but decimal point for output)
3 hexadecimal
4 integer
5 multiple precision (base 10)
6 multiple precision (base 8)
7 multiple precision (base 16)
8 octal
9 real

If negative, type is defined by variable has not been assigned a value

DATA INTERFACE I/O

Explicit Arguments
IN:
ITYP - (i*2) - Code that gives the type of variable for a particular call to this routine. Variables are either declared to be of this type or, if no variables are specified, a list of all variables of that type are given.

OUT:
RETCD - 1 = ok, 2 = error

Implicit Arguments (through common)
COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED
COMMON /V/ TYPE, AVBLS, VBL, VLEN
COMMON /CONS/ ALPHA, COMMA, BLANK, RPAR, LPAR
IN:
LEND - (i*2) - Last non-blank in vector LINE(80)
VIEWSW -(i*2) - View switch
IN/OUT:
LINE(80) - (log*1) - Holds input command line. If declaration has no argument, this vector is then used to output a list of variable of the type specified.
TYPE(RRW,RCL) - (i*2) - Holds the type code for each variable.
DECLR routine

NOT USED:
  LEVEL, NONBLK, BASED, AVBLS, VBLS, VLEN
CONSTANTS USED:
  ALPHA, BLANK, COMMA
CONSTANTS NOT USED:
  EQ, LPAR, RPAR

HOW ROUTINE IS TO BE CALLED
CALL DECLR(ITYP, RETCD)

WHAT ROUTINES CALL IT?
  CMND

WHAT ROUTINES IT CALLS?
  ERRMSG  VARSCN  IABS
Subroutine DOENTR

PURPOSE: Do an entry. Must scan for multiple statements per line and also recognize function names.

DATA INTERFACE I/O

Explicit Arguments

IN:
  LOW, LHIGH - (i*2) - Limits for scan of LINE.

IN/OUT:
  FORM(128) - (log*1) - Contains Formula.

Implicit Arguments (through common)

COMMON /DCTL/ PROW, PCOL, DROW, DCOL, DRWV, DCLV
COMMON /V/ TYPE, AVBLS, VBLS, VLEN
COMMON /FVLDC/ FVLD
COMMON /D2R/ NRDSP, DCDSP

None of above are used in this routine, but most are used in routines which are called by this routine.
  NOT USED:
    All

HOW ROUTINE IS TO BE CALLED

CALL DOENTR(FORM, LOW, LHIGH)

WHAT ROUTINES CALL IT?

RECALC

WHAT ROUTINES IT CALLS?

DOSTMT    INDEX
Subroutine DOIF

PURPOSE: Handle if in cell or formula

DATA INTERFACE I/O

Explicit Arguments

IN:
- LINE(110) - (log*1) - Line being processed
- LLB, LRB - (int) - Line left and right bracket
- LLAST - (int) - Position of last character in LINE.

OUT:
NONE

Implicit Arguments (through common)
NONE

HOW ROUTINE IS TO BE CALLED
CALL DOIF(HLINE, LLB, LRB, LLAST)

WHAT ROUTINES CALL IT?
DOSTMT

WHAT ROUTINES IT CALLS?
DOMFCN   DOSTMT
GETLOG   INDEX
MINO     TEST
DOMATH routine

Subroutine DOMATH

PURPOSE: Do math functions, minimum, maximum, average, sum, standard deviation depending on INDEXF.

DATA INTERFACE I/O

Explicit Arguments

IN:
   INDEXF - Flag math function to do.
   VAR -(real) - Variable to work in.

OUT:
   ACX -(real) - Result of math function

IN/OUT:
   AC - (real) - Accumulator
   CTR - (real) - Counter
   SS - (real) - For figuring standard deviation.

Implicit Arguments (through common)

NONE

HOW ROUTINE IS TO BE CALLED

CALL DOMATH(INDEXF, VAR, AC, SS, CTR, ACX)

WHAT ROUTINES CALL IT?

DOMFCN

WHAT ROUTINES IT CALLS?

NONE
Subroutine DOMFCN

**PURPOSE:** DO MATH functions.
First get a variable name. All math functions require variable names since their variables are their only valid arguments.

**DATA INTERFACE I/O**

**Explicit Arguments I/O**

**IN:**
- ACX - (real*8) - Result of evaluation.
- INDEXF - (int) - Tells which function
- LINE(110) - (LOG*1) - Line or formula being processed.
- LLB - (int) - Left bracket
- LRB - (int) - Right bracket

**OUT:**
- NONE

**Implicit Arguments (through common)**
- COMMON /DCTL/ PROW, PCOL, DROW, DCOL, DRWV, DCLV
- COMMON /V/ TYPE, AVBLS, VBLS, VLEN
- COMMON /FVLD/ FVLD
- COMMON /D2R/ NRDSP, NCDSP
- COMMON /DOT/ KDRW, KDCL
- COMMON /ILN/ ILNFG, ILNCT, ILINE

**IN:**
- TYPE(RRW,RCL) - (i*2) - Type in cell.

**IN/OUT:**
- AVBLS - Equivalenced as follows
  - EQUIVALENCE ( ACY, AVBLS(1,27))
- VBLS - Equivalenced as follows
  - EQUIVALENCE ( VBLS(1,1,1), XVBLA(1,1))

**NOT USED:**
- PROW, PCOL, DROW, DCOL, DRWV, DCLV, VLEN, FVLD, NRDSP, NCDSP, KDRW, KDCL, ILNFG, ILNCT, ILINE

**HOW ROUTINE IS TO BE CALLED**
CALL DOMFCN(LINE,LLB,LRB,INDEXF,ACX)
WHAT ROUTINES CALL IT?
DOIF  DOSTMT

WHAT ROUTINES IT CALLS?
ABS  DOMATH.
MTHINI  VARSCN
Subroutine DOSTMT

PURPOSE: Handle 1 statement parsing (does a bit more of the work with the part of the line stripped to have exactly one command in it.

DATA INTERFACE I/O

Explicit Arguments

IN:
  LINE(110) - (LOG*1) - Contains line or formula
  LLAST - (int) - Position of last character in LINE.

OUT:
  NONE

Implicit Arguments (through common)

COMMON /DCTL/ PROW, PCOL, DROW, DCOL, DRWV, DCLV
COMMON /V/ TYPE, AVBLS, VBLS, VLEN
COMMON /FVLD/ FVLD
COMMON /D2R/ NRDSP, NCDSN
COMMON /DOT/ KDRW, KDCL
COMMON /ILN/ ILNFG, ILNCT,ILINE

IN:
  KDCL, KDRW - (i*2) - Index into AVBLS and TYPE.

OUT:
  ILINE(106) -(LOG*1) - Copy of LINE from 1 to LLAST-1
  ILNCT -(i*2) -Length of line passed to CALC
  ILNFG- (i*2) -Line flag set to 1 here

IN/OUT:
  AVBLS(100,27) - (LOG*1) - Equivalenced to ACY as follows.
   EQUIVALENCE (ACY, AVBLS(1,27))
  VBLS(8,RW,RCL) - (LOG*1) - Equivalenced to XVBLS
   EQUIVALENCE (VBLS(1,1,1), XVBLS(1,1))
  TYPE(RW,RCL) - (i*2) - Type in cell.

NOT USED:
  PROW, PCOL, DROW, DCOL, DRWV, DCLV, FVLD, NRDSP, NCDSN

HOW ROUTINE IS TO BE CALLED

CALL DOSTMT(LINE, LLAST)
WHAT ROUTINES CALL IT?
   DOENTR   DOIF

WHAT ROUTINES IT CALLS?
   ABS      CALC
   DOIF     DOMFCN
   FNAME    INDEX

DOSTMT routine
Subroutine DSPSHT

PURPOSE: Display spreadsheet.

DATA INTERFACE I/O

Explicit Argument
IN:
   ICODE -  2 = Display to screen
            10 = Display to file

Implicit Arguments (through common)
COMMON /NMSH/ NMSH
COMMON /ICPOS/ IC2POS, ICPOS
COMMON /DCTL/ PROW, PCOL, DROW, DCOL, DRWV, DCLV
COMMON /D2R/ NRDSP, NCDSF
COMMON /ILN/ ILNFG, ILNCT, ILINE
COMMON /OAR/ OSWIT, OCNTR, OARRAY
COMMON /V/ TYPE, AVBLS, VBLS, VLEN
COMMON /FFFG/ FORMFG, RCFGX
COMMON /FVLDC/ FVLDC
COMMON /DSPCMN/ DVS, CWIDS
COMMON /INITD/ IBITMP
IN:
   CWIDS(DRW) - (i*2) - Widths of cell
   DCLV, DRWV - (i*2) - Display column and rows in DVS
   FORMFG - (i*2) - Format of display globally
   NCDSP(DRW, DCL), NRDSP(DRW, DCL) - Physical coordinates of displayed sheet
   NMSH(80) - (LOG*1) - Default numberic format for spreadsheet
   TYPE(RRW, RCL) - (i*2) - Type of of variables

OUT:
   IC1POS, IC2POS (i*2) - Position on spreadsheet

IN/OUT:
   DVS(DRW, DCL) - (REAL*8) - Display array
   EQUIVALENCE (LDVS(1,1,1), DVS(1,1))
   FVLDC(RRW, RCL) - (LOG*1) - Flags contents of cells of sheet
   IBITMP(BRRCL) - (LOG*1) - Bitmap of used to speed up process
DSPSHT routine

VBLS(8,RRW,RCL) -(LOG*1) - Value of variables
EQUIVALENCE (XVBL(1,1), VBLS(1,1,1))

NOT USED:
PROW, PCOL, DROW, DCOL, ILNFG, ILNCT, ILINE, OSWIT, OCNTR,
OARRY, RCFGX

HOW ROUTINE IS TO BE CALLED
CALL DSPSHT(ICODE)

WHAT ROUTINES CALL IT?
SPRDSHT
XQTCMD

WHAT ROUTINES IT CALLS?
INZAS
MAXO
MINO
UVT100
Subroutine ERRCX

PURPOSE: This subroutine does initial syntax checking on the input line. The checks make sure that parentheses are balanced and that the equal sign is not misused.

DATA INTERFACE I/O

Explicit Arguments

IN: NONE
OUT:

RETCD - 1 = No errors detected, 2 = error found

Implicit Arguments (through common)

COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED
COMMON /CONS/ ALPHA, COMMA, BLANK, RPAR, LPAR, EQ

IN:

LEND - (i*2) - Last non-blank character in LINE(80)
LINE(80) - (log*1) - Line being checked.
NONBLK - (i*2) - Last non-blank in LINE.

OUT: NONE
NOT USED:

LEVEL, VIEWSW, BASED

CONSTANTS USED:

BLANK, EQ, LPAR, RPAR

CONSTANTS NOT USED:

ALPHA, COMMA

HOW ROUTINE IS TO BE CALLED

CALL ERRCX(RETCD)

WHAT ROUTINES CALL IT?

CALC

WHAT ROUTINES IT CALLS?

ERRMSG VARSCN
**Subroutine ERRMSG**

**PURPOSE:** Prints out error messages as requested by code in MSG.

**DATA INTERFACE I/O**

**Explicit Arguments**

IN:
- MSG -(i*2) - Code for the message to print out.

OUT:
- NONE

**Implicit Arguments (through common)**
- NONE

**HOW ROUTINE IS TO BE CALLED**

CALL ERRMSG(MSG)

**WHAT ROUTINES CALL IT?**

AT    BASCNG
CALBIN  CALC
CALUN   CMND
CONTYP  DECLR
ERRCX   INPOST
MULADD  MULDIV
MULMUL  NEXTEL
POSTVL  VAROUT
ZNEG

**WHAT ROUTINES IT CALLS?**

NONE
Subroutine FLIP

PURPOSE: Flips the non-zero digits up to PT in the vector VEC in reverse order. Used to place numbers in proper order into VBLS that have been read in high order first.

DATA INTERFACE I/O

Explicit Arguments

IN:
  SIZE - (i*2) - Size of VEC
  PT - (i*2) - Holds the range of the flipping action. (usually the high order non-zero digit)

IN/OUT:
  VEC(SIZE) - (LOG* 1) - Vector which holds the vectors to be reversed.

Implicit Arguments (through common)

NONE

HOW ROUTINE IS TO BE CALLED

CALL FLIP(VEC,SIZE,PT)

WHAT ROUTINES CALL IT?

NEXTEL

WHAT ROUTINES IT CALLS?

NONE
Subroutine FNAME

PURPOSE: Return function name.

DATA INTERFACE I/O
Explicit Arguments
IN:
  LINE (110) - (log*1) - Line to search for function in.
  LLAST - (int) - Passed but not used here.
OUT:
  INDEXF - (i*2) - Code for name of function

Implicit Arguments (through common)
NONE

HOW ROUTINE TO BE CALLED
CALL FNAME(LINE, LLAST, INDEXF)

WHAT ROUTINES CALL IT?
DOSTMT

WHAT ROUTINES IT CALLS?
NONE
Subroutine GETLOG

**PURPOSE:** Checks logical type relationship of 2 variables.

**DATA INTERFACE I/O**

**Explicit Arguments**

**IN:**
- LINE(110) - (log*1) - Line where variables occur.

**OUT:**
- LLAST -(int) - Set where match is.
- LMX - (int) - Length of LINE.
- LOGTYP - (int) - Type of logical relationship found.

**Implicit Arguments (through common)**

NONE

**HOW ROUTINE IS TO BE CALLED**

CALL GETLOG(LINE, LMX, LOGTYP, LAST)

**WHAT ROUTINES CALL IT?**

DOIF

**WHAT ROUTINES IT CALLS?**

NONE
Subroutine GETNNB

PURPOSE: Get next non-blank element from LINE starting at NONBLK+1.

DATA INTERFACE I/O
Explicit Arguments
IN:
  NONE
OUT:
  RETCD - 1 = ok, 2 = No non-blank found
  IPT (i*2) - Points to position in LINEW where next non-blank is found. It is up to calling program to reset NONBLK for next scan.

Implicit Arguments (through common)
COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED
COMMON /CONS/ ALPHA, COMMA, BLANK, RPAR, LPAR, EQ
IN:
  LEND (i*2) - Last non-blank in LINE(80)
  LINE(80) - (LOG*1) - Line being scanned.
  NONBLK - (i*2) - Holds character to left of the start of the scan.
OUT:
  NONE
NOT USED:
  LEVEL, VIEWSW, BASED
CONSTANTS USED:
  BLANK
CONSTANTS NOT USED:
  ALPHA, COMMA, RPAR, LPAR, EQ

HOW ROUTINE IS TO BE CALLED
CALL GETNNB(IPT, RETCD)

WHAT ROUTINES CALL IT?
  AT     BASCNG CMND
  NEXTEL STRCMP

WHAT ROUTINES IT CALLS?
  NONE
Subroutine GN

PURPOSE: Get number routine.

DATA INTERFACE I/O

Explicit Arguments

IN:
  LINE(110) - (LOG*1) - Line to search in
  LEND - (int) - Last non blank character in LINE

OUT:
  LAST - (int) - Index of last number processed
  NUM - (int) - The number found is returned here.

Implicit Arguments (through common)

NONE

HOW ROUTINE IS TO BE CALLED

GN(LAST, LEND, NUM, LINE)

WHAT ROUTINES CALL IT?

XQTCMD
VARSCN

WHAT ROUTINES IT CALLS?

MINO
Subroutine HELP

PURPOSE: Print help information on screen using first 22 lines. Assumes XQTCMD invalidates the display.

DATA INTERFACE I/O
Explicit Arguments
IN:
  LVL - (i*2) - Not used here!
OUT:
  NONE

Implicit Arguments (through common)
  NONE

HOW ROUTINE IS TO BE CALLED
  CALL HELP(LVL)

WHAT ROUTINES CALL IT?
  SPREDSHT
  XQTCMD

WHAT ROUTINES IT CALLS?
  UVT100
INDEX routine

INTEGER FUNCTION INDEX

PURPOSE: Finds the location of a character within a string.
Returns the location of character or the end of the string.

DATA INTERFACE I/O

Explicit Arguments
IN:
   STR(1) - (log*1) String to be searched
   C -(log*1) - Character to be searched for.
OUT:
   Returns location of character or end of string.

Implicit Arguments (through common)
NONE

HOW ROUTINE IS TO BE CALLED
XXX = INDEX(STR,C)

WHAT ROUTINES CALL IT?
CMND DOENTR
DOIF DOSTMT
XQTCMD SED

WHAT ROUTINES IT CALLS?
NONE
**Subroutine INPOST**

**PURPOSE:** Converts input string (infix notation) to postfix for later evaluation by POSTVL.

**DATA INTERFACE I/O**

**Explicit Arguments**

- **IN:**
  - NONE

- **OUT:**
  - RETCD - 1 = Ok, 2 = Error

**Implicit Arguments (through common)**

- COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED
- COMMON /V/ TYPE, AVBLS, VBLS, VLEN
- COMMON STACK/ STACK1, STACK2, ST1PT, ST2PT, ST1TYP, ST2TYP
- COMMON /ERROR/ LASTOP
  - **IN:**
    - ST1LIM, ST2LIM - (i*2) - Holds limit of respective stack.
    - VLEN(9) - (i*2) - Holds the number of bytes used by each data type
  - **OUT:**
    - LASTOP (i*2) - Holds the type of 1st element obtained on LINE(80).
      - Set at 0 at beginning of expression. Used by NEXTEL to identify unary operators.

- **IN/OUT:**
  - NONBLK - (i*2) - Pointer in LINE(80). NEXTEL starts scan at NONBLK+1
  - STACK1(100,40) - (LOG*1) - Where string is stored in postfix
  - ST1PT, ST2PT, (i*2) - Stack pointer for respective stacks.
  - ST2TYP(40) - (i*2) - Type of each element in stack 2
  - ST1TYP(40) - (i*2) - Type of each element in stack 1

- NOT USED:
  - LEVEL, LINE, LEND, VIEWSW, BASED, TYPE, AVBLS, STACK2

**HOW ROUTINE IS TO BE CALLED**

CALL INPOST(RETCD)
WHAT ROUTINES CALL IT?
CALC

WHAT ROUTINES IT CALLS?
ERRMSG
NEXTEL  (Gets the next element from LINE(80))
IN2AS routine

Subroutine IN2AS

PURPOSE: Convert row to letters. Assumes Col=2 or more, Row 1= A-Z

DATA INTERFACE I/O

Explicit Arguments
IN:
   ROW (i*2) - Row number to be converted
OUT:
   CHRS(4) - (LOG*1) - Character representation for number

Implicit Arguments (through common)
NONE

HOW ROUTINE IS TO BE CALLED
CALL IN2AS(ROW,CHRS)

WHAT ROUTINES CALL IT?
DSPSHT
XQTCMD
RELVBBL

WHAT ROUTINES IT CALLS?
NONE
INTEGER FUNCTION ISGN

PURPOSE: Returns zero if argument is zero, one if argument is positive, and minus one if argument is negative.

DATA INTERFACE I/O

Explicit Arguments

IN:
  IARG - (i*2) - Argument to be checked

OUT:
  Returns 0, 1, -1 depending on sign of IARG

Implicit Arguments (through common)

NONE

HOW ROUTINE TO BE CALLED

III = ISGN(IARG)

WHAT ROUTINES CALL IT?

XQTCMD

WHAT ROUTINES IT CALLS?

NONE
Subroutine LIST

PURPOSE: Lists the legal calc commands and gives a brief description of their functions.

DATA INTERFACE I/O
Explicit Arguments
NONE

Implicit Arguments (through common)
NONE

HOW ROUTINE TO BE CALLED
CALL LIST

WHAT ROUTINES CALL IT?
CALC

WHAT ROUTINES IT CALLS?
NONE
Subroutine MOUT

PURPOSE: This routine outputs value of a multiple precision variable.
INDEX points to an element in VBLS to be output

DATA INTERFACE I/O

Explicit Arguments
IN:
INDEX - (i*2) - Pointer to variable to be output

OUT:
RETCID - (i*2) - 1-ok, 2-error

Implicit Arguments (through common)
COMMON /V/ TYPE, AVBLS, VBLS, VLEN
IN:
AVBLS(100,27) - (log*1) - Holds variables.
TYPE(RRW,RCL) - (i*21) - Holds type of each variable.

OUT:
NONE

NOT USED:
VBLS, VLEN

HOW ROUTINE IS TO BE CALLED
CALL MOUT(INDEX,RETCID)

WHAT ROUTINES CALL IT?
VAROUT

WHAT ROUTINES IT CALLS?
ERRMSG PRTCON
Subroutine MTHINI

PURPOSE: Initialization routine.

DATA INTERFACE I/O

Explicit Arguments
IN:
  INDEXF - (i*2) - Flag determining how to initialize AC

OUT:
  SS,CTR,ACX - (REAL) - Initialized to 0.
  AC - (real) - initialized to 1.E20 if INDEXF is 1, and -1.E20 if INDEXF is 2

Implicit Arguments (through common)
  NONE

HOW ROUTINE IS TO BE CALLED
  CALL MTHINI(INDEXF,AC,SS,CTR,ACX)

WHAT ROUTINES CALL IT?
  DOMFCN

WHAT ROUTINES IT CALLS?
  NONE
Subroutine MULADD

PURPOSE: Multiple precision addition and subtraction routine.

DATA INTERFACE I/O

Explicit Arguments

IN:
- ENTRY - (i*2) Coded specification of Base and operation (add or subtraction)
- PT1 -(i*2) - Pointer to operand 1 (in stack 1)
- PT2 -(i*2) - Pointer to operand 2 (in stack 2)

OUT:
- RETCD - 1=OK, 2=ERROR

Implicit Arguments (through common)

COMMON /STACK/ STACK1, STACK2, ST1PT, ST2PT, ST1TYP, ST2TYP, ST1LIM, ST2LIM

IN/OUT:
- STACK1(100,40) - (log*1) - Contains operand 1 and is where result is stored.
- STACK2(100,40) - (log*1) - Contains operands (is not cleaned up by the operation.)

NOT USED:
- ST1PT, ST2PT, ST1TYP, ST2TYP, ST1LIM, ST2LIM

HOW ROUTINE IS TO BE CALLED

CALL MULADD(PT1, PT2, RETCD, ENTRY)

WHAT ROUTINES CALL IT?

CALBIN

WHAT ROUTINES IT CALLS?

ERRMSG
Subroutine MULCON

**PURPOSE:** Converts multiple precision number in `STACK(INDEX)` from base "OLD" to base "NEW".

**DATA INTERFACE I/O**

**Explicit Arguments**

**IN:**
- `INDEX` - (i*2) - Pointer to number (in stack) to be converted.
- `NEWA` - (i*2) - New base code
- `OLDA` - (i*2) - Old base code.
  
  base code:: ( 5=(base8), 6=(base 10), 7=(base 16)

**OUT:**
- `RETCDE` = 1=ok, 2=error

**IN/OUT:**
- `STACK(INDEX)` - (log*1) - Holds the multiple precision number to be converted. Dimensioned (100,40), Where result is put.

**Implicit Arguments (through common)**

NONE

**HOW ROUTINE IS TO BE CALLED**

CALL MULCON(STACK, INDEX, OLDA, NEWA, RETCD)

**WHAT ROUTINES CALL IT?**

CONTYP

**WHAT ROUTINES IT CALLS?**

NONE
Subroutine MULDIV

PURPOSE: Performs multiple precision division.

DATA INTERFACE I/O

Explicit Arguments

IN:
- PT1 - (i*2) - Pointer to STACK1 element (dividend)
- PT2 - (i*2) - Pointer to STACK2 element (divisor)
- BASE - (i*2) - Holds base: 8, 10, 16

OUT: RETCD - (i*2) - 1=OK, 2=ERROR

Implicit Arguments (through common)

COMMON /STACK/ STACK1, STACK2, ST1PT, ST2PT, ST1TYP, ST2TYP, ST1LIM, ST2LIM

IN:
- STACK2(100,40) - (log*1) - Holds divisor

IN/OUT:
- STACK1 (100,40) - (log*1) - Holds dividend and result

NOT USED:
- ST1PT, ST2PT, ST1TYP, ST2TYP, ST1LIM, ST2LIM

HOW ROUTINE IS TO BE CALLED

CALL MULDIV(PT1, PT2, RETCD, ENTRY)

WHAT ROUTINES CALL IT?

CALBIN

WHAT ROUTINES IT CALLS?

ERRMSG
Subroutine MULMUL

**PURPOSE**: Performs multiple precision multiplication.

**DATA INTERFACE I/O**

**Explicit Arguments**

**IN**:
- PT1 - (i*2) - Pointer to STACK1 element (operand 1)
- PT2 - (i*2) - Pointer to STACK2 element (operand 2)
- ENTRY - (i*2) - Specifies base in argument of subtraction.

**OUT**:  
- RETCD - (i*2) - 1=OK, 2=ERROR(overflow)

**Implicit Arguments (through common)**

**COMMON /STACK/ STACK1, STACK2, ST1PT, ST2PT, ST1TYP, ST2TYP, ST1LIM, ST2LIM**

**IN**:  
- STACK2(100,40) - (log*1) - Holds operand 2

**IN/OUT**:  
- STACK1 (100,40) - (log*1) - Holds operand 1 and result

**NOT USED**:  
- ST1PT, ST2PT, ST1TYP, ST2TYP, ST1LIM, ST2LIM

**HOW ROUTINE IS TO BE CALLED**

**CALL MULMUL(PT1, PT2, RETCD, ENTRY)**

**WHAT ROUTINES CALL IT?**

**CALBIN**

**WHAT ROUTINES IT CALLS?**

**ERRMSG**
Subroutine NEXTEL

**PURPOSE:** Scans LINE(80) from NONBLK+1 and returns the next element. This element could be a constant, value of a variable, a binary function code, or a unary function code. Upon return, NONBLK points to last character of next element.

**DATA INTERFACE I/O**

### Explicit Arguments

**IN:** NONE  
**OUT:**  
- **RETCID**  
  1 = If operand (value in RETVAL(100))  
  3 = No more elements  
  2 = If operator (value in RETTYP)  
  4 = If error  
- **RETVL** - holds value of operand found (either constant or if a variable (A-Z,x, the value of that variable)  
- **RETTYP** - (i*2) - The type code

### Implicit Arguments (through common)

- **COMMON /V/ TYPE, AVBLS, VBLS, VLEN**  
- **COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED**  
- **COMMON /DIGV/DIGITS**  
- **COMMON /CONS/ ALPHA, COMMA, BLANK, RPAR, LPAR, EQ**  
- **COMMON /ERROR/ LASTOP**

**IN:**  
- **AVBLS(100,27)** - (i*2) - Holds value of variables  
- **VBLS(8,RRW,RCL)** - (i*2) - Holds value of variables  
- **LEND** - (i*2) - Length of LINE(80)  
- **LINE(80)** - (log*l) - Command line  
- **TYPE(RRW,RCL)** - (i*2) - Type code for each variable  
- **VLEN(9)** - (i*2) - Gives length in bytes for each data type.

**IN/OUT:**  
- **LASTOP** - (i*2) - Used to hold value of last operator so that unary operators can be identified in cases like A*-B AND A/(-3).  
- **NONBLK** - (i*2) - Indicates where scan is to begin.

**NOT USED:**  
- **LEVEL, VIEWSW, BASED**
CONSTANTS USED:
   ALPHA, EQ
CONSTANTS NOT USED:
   BLANK, COMMA, LPAR, RPAR

HOW ROUTINE IS TO BE CALLED
   CALL NEXTEL( RETVAL, RETTYP, RETCD)

WHAT ROUTINES CALL IT?
   INPOST

WHAT ROUTINES IT CALLS?
   ERRMSG FLIP GETNNB RVBOO VARSCN DFLOAT
PCHELP file

PURPOSE: It is an include file in spredsht or HELP, it prints out help information.

DATA INTERFACE I/O

Explicit Arguments
NONE

Implicit Arguments (through common)
NONE

HOW ROUTINE IS TO BE CALLED
Included in HELP

WHAT ROUTINES CALL IT?
NONE

WHAT ROUTINES IT CALLS?
UVT100
POSTVL routine

Subroutine POSTVL

PURPOSE: Converts postfix expressions in STACK1 to a value.

DATA INTERFACE I/O

Explicit Arguments

IN:
NONE

OUT:
RETCD - (i*2) - 1 = OK, 2 = ERROR

Implicit Arguments (through common)

COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED
COMMON /V/ TYPE, AVBLS, VBLS, VLEN
COMMON /STACK/ STACK1, STACK2, ST1PT, ST2PT, ST1TYP, ST2TYP,
ST1LIM, ST2LIM

IN:
ST1TYP(40) - (i*2) - Vector types for each element of STACK1
ST2LIM - (i*2) - STACK2 limit
STACK1 (100,40) - (log*1) - Holds the original postfix expression.
VIEWSW - (i*2) - View switch - 1 = off, 2 = display commands,
3 = display value of expressions, 4 = display all
VLEN(9) - (i*2) - Length of variable

OUT:
AVBLS(100,27) (log*1) - Holds values of variables
VBLS(8,RRW,RCL) - (log*1) - Holds values of complexly-named
variables. 1st 27 elements are place holders for AVBLS; routines
that generate dimensions ID1,ID2 for VBLS return dimensions
1-27, for A-Z$. These result in AVBLS array being used. VBLS
array (max length 8 bytes/variable) is used for other variables
whose names are ,ALPHA>{<ALPHA>{<NUM>{<num>} (with option
for any reasonable # of ALPHAS and Numerics but clamped at
RRW,RCL values to work correctly.)
ST2TYP(40) - (i*2) - Vector types for each element of STACK2
STACK2 (100,40) - (log*1) - Used to evaluate expression in STACK1
TYPE(RRW,RCL) - (i*2) - Holds data type for each of the variables
POSTVL routine

IN/OUT:
  ST1PT, ST2PT - (i*2) - STACK pointers

NOT USED:
  LEVEL, LINE, NONBLK, LEND, BASED, ST1LIM

HOW ROUTINE IS TO BE CALLED
  CALL POSTVL(RETCXD)

WHAT ROUTINES CALL IT?
  CALC

WHAT ROUTINES IT CALLS?
  CALBIN  CALUN  ERRMSG  VAROUT
FUNCTION PRTCON

PURPOSE: Converts 0 to appropriate number for print with vector digits.

DATA INTERFACE I/O

Explicit Arguments

IN:
  L1 - (LOG*1) - Digits in.
  IBASE -(i*2) - base of number to be converted, 10,8,16.

OUT:
  Returns appropriate number for printing.

Implicit Arguments (through common)

COMMON /DIGV/DIGITS

CONSTANTS USED:
  DIGITS - Accessed to find appropriate value.

HOW ROUTINE IS TO BE CALLED

CALL PRTCON(L1,IBASE)

WHAT ROUTINES CALL IT?

MOUT

WHAT ROUTINES IT CALLS?

NONE
Subroutine RECALC

PURPOSE: Recalculate command, recomputes all elements of the spreadsheet where valid.

DATA INTERFACE I/O

Explicit Arguments
NONE

Implicit Arguments (through common)
COMMON /DCTL/ PROW, PCOL, DROW, DCOL, DRWV, DCLV
COMMON /FVLDC/ FVLD
COMMON /V/ TYPE, AVBLS, VBLS, VLEN
COMMON /FFGG/ FORMFG, RCFGX, PZAP, RCONE
COMMON /D2R/ NCDSP, NRDSP
COMMON /DOT/ KDRW, KDCL

IN:
   NCDSP(DRW,DCL) - (INT) - Physical coordinates of displayed sheets
   NRDSP(DRW,DCL) - (INT) - Physical coordinates of displayed sheets
   PROW,PCOL - (i*2) - Physical row and column

OUT:
   KDCL, KDRW - (i*2) - Current cell coordinate index into AVBLS

IN/OUT:
   DROW, DCOL - (i*2) - Current cell coordinate index into display sheet
   FVLD(RRW, RCL) - (log*1) - FVLD Flag,
      0 = No formula,
      -1 = Display formula itself, not value
      1 = Valid formula there to evaluate, Initially all 0's
      -2,-3 = Display formula
      2 = Numeric, compute once then set FVLD to 2
      3 = Numeric constant, already computed...Do not recompute.
   RCONE - (i*2) - Recalc flag

NOT USED:
   DRWV, DCLV, TYPE, AVBLS, VBLS, VLEN, FORMFGM RCFGX, PZAP

HOW ROUTINE IS TO BE CALLED
CALL RECALC
WHAT ROUTINES CALL IT?
SPREADSHEET

WHAT ROUTINES IT CALLS?
DOENTR

RECALC routine
Subroutine RELVBL

PURPOSE: Relocate variable below/right of JRTR, JRTC into LNOUT from LNIN.

DATA INTERFACE I/O

Explicit Arguments

IN:
- JRTR, JRTC - (int) - Coordinates of upper left hand corner of part of spreadsheet which needs to be relocated
- LNIN (128) - (LOG*1) - Line in
- INRW, INCL - (int) - Source
- JOUTR, JOUTC - (int) - Destination

OUT:
- LNOUT(128) - (LOG*1) - Line out

Implicit Arguments (through common)

NONE

HOW ROUTINE IS TO BE CALLED

RELVBL(LIN, LNOUT, INRW, INCL, JOUTR, JOUTC, JRTR, JRTC)

WHAT ROUTINES CALL IT?

XQTCMD

WHAT ROUTINES IT CALLS?

IN2AS MAXO
MINO VARSCN
Subroutine RVBOO

PURPOSE: This routine copies ID1, ID2 into RETV array to avoid some byte integer conversion problems. This packing is used to access variable location later.

DATA INTERFACE I/O
Explicit Arguments
IN:
ID1,ID2 -(int) - In values
OUT:
RETV(2) - (i*2) - As described above.

Implicit Arguments (through common)
NONE

HOW ROUTINE TO BE CALLED
CALL RVBOO(RETV,ID1,ID2)

WHAT ROUTINES CALL IT?
NEXTEL

WHAT ROUTINES IT CALLS?
NONE
Subroutine SCMP

PURPOSE: This routine compares 2 arrays until either endstring is seen on one or mismatch is seen.

ICODE is flag which tells if strings are the same.
1 = same
2 = different

DATA INTERFACE I/O
Explicit Arguments
IN:
LINA,LINB - (log*1) - Strings to be compared
LENM (INTEGER) - Length of LINA
OUT:
ICODE - (INTEGER) - Flag if strings the same.
1 = same
2 = different

Implicit Arguments (through common)
NONE

HOW ROUTINE IS TO BE CALLED
CALL SCMP (LINA,LINB,LENM,ICODE)

WHAT ROUTINES CALL IT?
CMND

WHAT ROUTINES IT CALLS?
NONE
Subroutine SED

PURPOSE: String edit routine.

OPERATION:
Edit LIN to LWRK, with LENGTH variable holding input length in character. LCMD holds command line, which ultimately gets edited string copied back into it.

EDITS:
Character at IDELIM is delimiter. Replace string in 1st interval between delimiters with second.
However:
&1 to &4 gets contents (up to null) of ARGSTR(X,1) to (x,4)
&5 returns XAC value converted to decimal integer and printed.
&6 returns XAC value converted to ASCII code (1 byte) and inserted.
XAC enters with contents of accumulator Z (to avoid too much difficulty in using it owing to the ubiquity of use of %)
We enter just pointing at the command line after the enter and its space. Assume 1st character is our delimiter.

DATA INTERFACE I/O

Explicit Arguments
IN:
ARGSTR(52,4) - (LOG*1) - Argument string
LIN(1) -(LOG*1) - Part to be included
LENGTH - (INT) - Length of LIN
XAC - (real*8) - Passed, has value

OUT:
LWRK(1) -(LOG*1) - Where LIN is editted to

IN/OUT:
LCMD(1)-(LOG*1) - Command line

Implicit Arguments (through common)
NONE

HOW ROUTINE IS TO BE CALLED
SED(LCMD, LIN, LWRK, ARGSTR, XAC, LENGTH)
**WHAT ROUTINES CALL IT?**

XQTCMD

**WHAT ROUTINES IT CALLS?**

ABS       INDEX

SSCMP
Subroutine SLEND

PURPOSE: Sets value of LEND, pointer to last non-blank character in LINE(80).

DATA INTERFACE I/O

Explicit Arguments
IN:

NONE

OUT:

RETCD - (i*2) - 1 = NORMAL RETURN, 2 = ALL BLANKS

Implicit Arguments (through common)
COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED
COMMON /CONS/ ALPHA, COMMA, BLANK, RPAR, LPAR

IN:

LINE(80) - (LOG*1) - Holds command line

OUT:

LEND - (i*2) - Upon exit, points to the last non-blank character in LINE(80)

NOT USED:

LEVEL, NONBLK, VIEWSW, BASED

CONSTANTS USED:

BLANK

CONSTANTS NOT USED:

ALPHA, COMMA, RPAR, LPAR

HOW ROUTINE IS TO BE CALLED

CALL SLEND(RETCD)

WHAT ROUTINES CALL IT?

CALC

WHAT ROUTINES IT CALLS?

NONE
MAIN ROUTINE SPREDSHT

PURPOSE: MAIN PROGRAM, spreadsheet driver program

DATA INTERFACE I/O

Explicit Arguments
NONE

Implicit Arguments (through common)
COMMON /DCTL/ PROW, PCOL, DROW, DCOL, DRWV, DCLV
COMMON /D2R/ NRDSP, NCDSP
COMMON /IOLVL/ IOLVL
COMMON /V/ TYPE, AVBLS, VBLS, VLEN
COMMON /FVLDC/ FVLD
COMMON /OAR/ OSWIT, OCNTR, OARRY
COMMON /KLVL/ KLVL
COMMON /FFGG/ FORMFG, RCFGX, PZAP
COMMON /DEFVBX/ DVFMT
COMMON /NMSH/ NMSH
COMMON /INITD/ IBITMP
COMMON /DSPMN/ DVS, CWIDS

OUT:

AVBLS(100,27) -(LOG*1) - Initialized to zero
CWIDS(DRW) -(i*2) - Initialize cell width to 10
DCLV, DRWV - (i*2) - Set display limits to max columns and rows
DCOL, DROW - (i*2) - Display index
DVFMT(12) - (LOG*1) - Default format f9.2
DVS(DRW, DCL), (real*8) - Display virtual screen
FORMFG - (i*2) - Initialized to zero.
FVLD(RRW,RCL) -(LOG*1) - Initialized to zeros
IBITMP(BRRCL) -(LOG*1) - Initialized to zero
IOLVL - (i*2) - Initialized to five, logical unit where reads are done.
KLVL - (i*2) - Initialized to one
NCDSP,NRDSP(DRW,DCL) -(int) - Physical coordinates of display sheet
NMSH(80) - (LOG*1) - Numeric default format
OSWIT - (i*2) - Output switch, set to 20
SPREADSHT routine

PCOL, PROW - (i*2) - Initially set to 2 and 1 respectively
PZAP - (i*2) - Initially set to 0
RCFGX - (i*2) - Initially set to 0, FLAG
VBLS(8,RRW,RCL) (LOG*1) - Equivalenced to XXV, initialized to zeros

NOT USED:
    TYPE, VLEN, OCNTR, OARRY

WHAT ROUTINES IT CALLS?
    ASSIGN    CLOSE
    *DSPSHT    HELP
    *RECALC    TTYINI
    UV100      *XQTCMD

*MAIN IMPORTANT SUBPROGRAMS
Subroutine SSCMP

PURPOSE: String compare 2 arrays until either endstring is seen on one or mismatch is seen.

DATA INTERFACE I/O

Explicit Arguments

IN:
- LINA(1), LINB(1) - (LOG*1) - Strings to compare
- LENM - (int) - Length of one of lines

OUT:
- ICODE - (int) - 1 - if match, 0 - otherwise

Implicit Arguments (through common)
- NONE

HOW ROUTINE IS TO BE CALLED
- CALL SSCMP(LINA, LINB, LENM, ICODE)

WHAT ROUTINES CALL IT?
- SED

WHAT ROUTINES IT CALLS?
- NONE
Subroutine STRCMP

**PURPOSE:** STRCMP looks past blanks for the name help by NAME(1), ..., NAME(LENGTH), the return code RETCD indicates success or failure, 1 = Match, 2 = Failure. Upon exit, common variable NONBLK, if successful, points to one beyond the last character scanned for match, if failure, unchanged.

**DATA INTERFACE I/O**

**Explicit Arguments**

IN:
- LENGTH - (i*2) - Length of NAME
- NAME(LENGTH) - (LOG*1) - NAME to be compared

OUT:
- RETCD - (i*2) - 1 = Match, 2 = Failure

**Implicit Arguments (through common)**

COMMON LEVEL, LINE, NONBLK, LEND, VIEWSW, BASED
COMMON /CONS/ ALPHA, COMMA, BLANK, RPAR, LPAR, EQ

IN:
- LINE(80) - (LOG*1) - Command line

IN/OUT:
- NONBLK - (i*2) - Upon exit, common variable NONBLK, if successful, points to one beyond the last character scanned for match
  if failure, unchanged.

NOT USED:
- LEVEL, LEND, VIEWSW, BASED

CONSTANTS NOT USED:
- ALPHA, COMMA, BLANK, RPAR, LPAR, EQ

**HOW ROUTINE IS TO BE CALLED**

STRCMP( NAME, LENGTH, RETCD)

**WHAT ROUTINES CALL IT?**

CMND

**WHAT ROUTINES IT CALLS?**

GETNNB
**Subroutine TEST**

**PURPOSE:** Test the logical relation for If statements, Flag=1 if true, 0 else.

**DATA INTERFACE I/O**

**Explicit Arguments**

**IN:**
- LOGTYP - (i*2) - Logical relation type.
- VI, V2 - (i*2) - Variables being tested.

**OUT:**
- FLAG (i*2) - Return 1 if true and 0 otherwise.

**Implicit Arguments (through common)**

NONE

**HOW ROUTINE IS TO BE CALLED**

CALL TEST(LOGTYP, FLAG, VI, V2)

**WHAT ROUTINES CALL IT?**

DOIF

**WHAT ROUTINES IT CALLS?**

NONE
Subroutine TTYINI

PURPOSE: This routine is a place holder for a routine to perform terminal and/or system dependent initializations. The pure FORTRAN version just returns. Unit 5 is the terminal input here.

Typical action:
- Set the terminal not to wrap around
- Attach terminal to munge around the escape sequences to allow special function and/or error keys to work.

DATA INTERFACE I/O

Explicit Arguments
NONE

Implicit Arguments (through common)
NONE

HOW ROUTINE IS TO BE CALLED
CALL TTYINI

WHAT ROUTINES CALL IT?
SPREDSH

WHAT ROUTINES IT CALLS?
NONE
Subroutine USRFCT

PURPOSE: User function routine, generates parsing and execution or routine calls of form \texttt{U FNAME(ARGUMENTS)}
where \texttt{LINE(80)} contains command line and all arguments may be parsed.
Parsing is up to user. Note \texttt{VARSCN} may be called to parse variable names. Supplied version calls \texttt{IDATE} which returns system date in RSX or VMS as integer day, month, and year. This returns here in \texttt{ACT}, \texttt{U}, and \texttt{V})

DATA INTERFACE I/O

Explicit Arguments
IN:
\texttt{LINE(80)} - (log*1) - Contains command line

OUT:
\texttt{RETCQD} - (i*2) - 1=OK, 2 = ERROR

Implicit Arguments (through common)
COMMON /V/ TYPE, AVBLS, VBLS, VLEN

IN/OUT:
\texttt{AVBLS(100,27)} - (LOG*1) - Equivalenced and set
EQUIVALENCE (XAC, AVBLS(1,27))
EQUIVALENCE (TAC, AVBLS(1,20))
EQUIVALENCE (UAC, AVBLS(1,21))
EQUIVALENCE (VAC, AVBLS(1,22))
\texttt{VBLS(8,RRW,RCL)} - (LOG*1) - Equivalenced
EQUIVALENCE (VBLS(1,1,1), JVBLs(1,1,1))
EQUIVALENCE (VBLS(1,1,1), XVBlS(1,1,1))

HOW ROUTINE IS TO BE CALLED
USRFCT(LINE, RETCD)

WHAT ROUTINES CALL IT?
CMND

WHAT ROUTINES IT CALLS?
IDATE
Subroutine UVT100

PURPOSE: VT100 video display command program.

DATA INTERFACE I/O

Explicit Arguments

IN:
CMD - (i*2) - One of the command in the parameter list
N1, N2 - (i*2) - Additional optional parameters depending on CMD.

Implicit Arguments (through common)

NONE

HOW ROUTINE IS TO BE CALLED

UVT100(CMD, N1, N2)

WHAT ROUTINES CALL IT?

DSPSHT  HELP
PCHelp  SPREDSHT
XQTCMD  VARSCN

WHAT ROUTINES IT CALLS?

GETADR  WTQIO

PARAMETERS

CUP = 1  CURSOR POSITION REPORT
CUU = 2  CURSOR UP
CUD = 3  CURSOR DOWN
CUR = 4  CURSOR FORWARD
CUB = 5  CURSOR BACK
DECDWL = 6  DOUBLE WIDTH LINE
DECDHL = 7  DOUBLE HEIGHT LINE
DECRC = 8  RESTORE CURSOR
DECS = 9  SAVE CURSOR
DECSWL = 10  SINGLE HEIGHT, SINGLE WIDTH LINE
ED = 11  ERASE IN DISPLAY
EL = 12  ERASE IN LINE
UVT100 routine

SGR = 13   SELECT GRAPHIC RENDITION
NEL = 14   NEXT LINE
SCS = 15   SELECT CHARACTER SET
SM = 16    SET MODE
RM = 17    RESET MODE
ANSI = 18  SET TERMINAL TO VT100 MODE(VT52)
Subroutine VAROUT

PURPOSE: Outputs the value of the variable pointed to by INDEX. Note that VAROUT is used to dump only values from AVBLS, not VBLS, (1X2=1 at all calls) thus don't bother to pick up any further information from VBLS here.

DATA INTERFACE I/O

Explicit Arguments

IN:
- INDEX -(i*2) - Points to the variable being output.
- 1X2 -(i*2) - Always 1 at calls

OUT:
- NONE

Implicit Arguments (through common)

COMMON /V/ TYPE, AVBLS, VBLS, VLEN
COMMON /OAR/ OSWIT, OCNTR, OARRY
COMMON /DIGS/ DIGITS
COMMON /CONS/ APLHA, COMMA, BLANK, RPAR, LPAR, EQ

IN:
- AVBLS(100,27)-(log*1) - Holds value of each variable
- OSWIT -(i*2) - OSWIT non-zero means output to OARRY. OSWIT=2 means no zeroing of OARRY; nothing much comes out.
- TYPE (RRW,RCL) -(i*2) - Type in this cell.

OUT:
- OARRY(100) -(LOG*1) - Output array
- OCNTR -(i*2) - Output counter

NOT USED:
- VBLS, VLEN

CONSTANTS USED:
- RPAR, DIGITS

CONSTANTS NOT USED:
- ALPHA, COMMA, BLANK, LPAR, EQ

HOW ROUTINE IS TO BE CALLED

VAROUT(INDEX,1X2)
WHAT ROUTINES CALL IT?
  CALC    POSTVL

WHAT ROUTINES IT CALLS
  ERRMSG    MOUT
  DABS      IABS
VARSCN routine

Subroutine VARSCN

PURPOSE: Scans for variable names of the form AAANNN where
AAA=letters between A and Z up to non-alpha, corresponding to row,
followed by numbers in range 0-9 making a decimal column number.
The letters are formed by
A-Z alone give row 1-26, col 1. % is row 27, col 1.
A1-Z1 give row 1-26, col 2
AA1-ZZ1 are rows 27-52, col 2

DATA INTERFACE I/O

Explicit Arguments

IN:
  LEND - (i*2) - Length of LINE
  LINE(LEND) - (LOG*1) - Line to be scanned
  IBGN - (i*2) - Where to begin scan

OUT:
  IVALID - (i*2) - 0 = NOT VALID, 1 = VALID
  ID1, ID2 - (int) - Cell of variable of form #nnn etc.
  LSTCHR - (int) - Last character processed

Implicit Arguments (through common)

COMMON /DCTL/ PROW, PCOL, DROW, DCOL, DRWV, DCLV
COMMON /D2R/ NRDSP, NCDSP

IN:
  DCOL, DROW -(int) - actual "cursor" location (current display location)
  NCDSP(DRW, DCL), NRDSP(DRW, DCL) - (int) - Real row, column of
display row ,col cells.
  PCOL, PROW - (int)- Physical row, column being handled.

NOT USED:
  DRWV, DCLV

HOW ROUTINE IS TO BE CALLED

VARSCN(LINE, IBGN, LEND, LSTCHR, ID1, ID2, IVALID)
VARSCN routine

WHAT ROUTINES CALL IT?
CMND       DECLR
DOMFCN      ERRCX
XQTCMD      NEXTEL
RELVBL

WHAT ROUTINES IT CALLS?
GN    UVT100    MIN0    MAX0
Subroutine XQTCMD

PURPOSE: This is the command processing routine.

DATA INTERFACE I/O
Explicit Arguments
IN:
   NONE
OUT:
   ICODE - (INT)
      1 = Command just moves on display, so no need to recalculate the entire sheet
      2 = Redraw whole screen
     -1 = Reinitialize display defaults
     -2 = New spreadsheet file setup
   Other = All ok

Implicit Arguments (through common)
COMMON /IOLVL/ IOLVL
COMMON /DCTL/ PROW, PCOL, DROW, DCOL, DRWV, DCLV
COMMON /V/ TYPE, AVBLS, VBLS, VLEN
COMMON /FVLDC/ FVLD
COMMON /OAR/ OSWIT, OCNTR, OARRY
COMMON /D2R/ NRDSP, NCDS
COMMON /KLVL/ KVLV
COMMON /FFGG/ FORMFG, RCFGX, PZAP
COMMON /NMSH/ NMSH
COMMON /ILN/ ILNFG, ILNCT, ILINE
COMMON /ICPOS/ IC1POS, IC2POS
COMMON /DEFVBX/ DEFVB
COMMON /DSPMN/ DVS, CWIDS
      IN:
         DEFVB(10,DRDW,DCL) - (LOG*1) - Default format
      OUT:
         ILNCT - (i*2) - Length of ILINE if it exists, it is set to zero here.
         ILNFG - (i*2) - Flags input from terminal for CALC.
XQTCMD routine

KLVL - (i*2) - Saves value of LEVEL for resetting when LEVEL is modified.
OSWIT - (i*2) - Set to zero before dropping in to CALC bare, set to 2 otherwise.
RCFGX - (i*2) - Flags whether to do recalculation.
RCONE - (i*2) - Forces recalc of all of sheet.

IN/OUT:

AVBLS(100,27) - Equivalenced as follows
  EQUIVALENCE (XAC, AVBLS(1,27)),(ZAC, AVBLS(1,26))
  EQUIVALENCE (XXAC, AVBLS(1,24)),(ZZAC, AVBLS(1,25))

DCOL, DROW - Current cell coordinates
DCLV, DRWV, - Number of max rows, cols actually on screen now.

DVS(DRW,DCL) - (REAL*8) - Equivalenced as follows
  EQUIVALENCE (LDVS(1,1,1), DVS(1,1))

FORMFG - (i*2) - Formula flag, if one then display formula in cells
FVLD(RRW,RCL) - (LOGICAL*1) - Flags contents of cells of sheet.

IC1POS, IC2POS - (i*2) - Physical coordinates of cell displayed.
IOLVL - (i*2) - Number of logical i/o unit, set 3 for indirect files and 5,6 for terminal i/o.

NCDSP(DRW,DCL) - (INTEGER) - Store physical coordinate for column of current cell.
NRDSP(DRW,DCL) - (INTEGER) - Store physical coordinate for row of current cell.

NMSH - (LOGICAL*1) - Numeric format of reading in cell.

PROW - (i*2) - Current physical row.
PCOL - (i*2) - Current physical column.
PZAP - (i*2) - Controls whether to redraw sceeen.

TYPE(RRW,RCL) - (LOGICAL*1) - Holds data type of that cell.
VBLS(8,RRW,RCL) - Equivalenced as follows
  EQUIVALENCE (ARGSTR(1,1), VBLS(1,1,1))
  EQUIVALENCE (XXVBLS(1,1), VBLS(1,1,1))
**XQTCMD routine**

**HOW ROUTINE IS TO BE CALLED**

CALL XQTCMD

**WHAT ROUTINES CALL IT?**

SPREADSHT

**WHAT ROUTINES IT CALLS?**

<table>
<thead>
<tr>
<th>ABS</th>
<th>ASSIGN</th>
<th>CALC</th>
<th>CLOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMDMUN</td>
<td>DSPSHT</td>
<td>EXIT</td>
<td>GN</td>
</tr>
<tr>
<td>HELP</td>
<td>IN2AS</td>
<td>INDEX</td>
<td>ISGN</td>
</tr>
<tr>
<td>MAXO</td>
<td>MINO</td>
<td>RELVBL</td>
<td>SED</td>
</tr>
<tr>
<td>UVT100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Subroutine ZERO

PURPOSE: Zeros out all variables except %.

DATA INTERFACE I/O

Explicit Arguments
NONE

Implicit Arguments (through common)
COMMON /V/ TYPE, AVBLS, VBLS, VLEN

IN:
NONE

OUT:
AVBLS (100,27) - (LOG*1) - Holds variables
VBLS(8,RRW,RCL) - (LOG*1) - Holds variables

IN/OUT:
TYPE(RRW,RCL) -(i*2) - Set to absolute value of what is was.

NOT USED:
VLEN

HOW ROUTINE IS TO BE CALLED
CALL ZERO

WHAT ROUTINES CALL IT?
CMND

WHAT ROUTINES IT CALLS?
IABS
ZNEG routine

INTEGER FUNCTION ZNEG

PURPOSE: Determines if variable pointed to by index is zero or negative or undefined as opposed to be defined and positive.
Returns 1 If true (zero or negative or undefined)
0 if false (positive)

DATA INTERFACE I/O

Explicit Arguments
IN:
INDEX- (i*2) - Pointer to variable being tested
OUT:
Returns value as described above.

Implicit Arguments (through common)
COMMON /V/ TYPE, AVBLS, VBLS, VLEN
IN:
AVBLS(100,27) - holds variable to be checked
TYPE(RRW,RCL) - (i*2) - tells type of variable to be checked.
NOT USED:
VBLS, VLEN

HOW FUNCTION IS TO BE CALLED
III = ZNEG (INDEX)

WHAT ROUTINES CALL IT?
AT CALC CMND

WHAT ROUTINES IT CALLS?
ERRMSG
APPENDIX C.

COMMON DATA DICTIONARY
COMMON DATA DICTIONARY

BLANK COMMON

BASED  - (i*2) - Default base when constants are entered.
    USED IN:
       BASCNG, BLKDO1, CMND, NEXTEL

LEND  - (i*2) - Points to the last non-blank character in LINE(80)
    USED IN:
       CALC, DECLR, ERRCX, NEXTEL, SLEND

LEVEL  - (i*2) - Holds the logical I/O unit where the next calc command lines come from.
    USED IN:
       AT, BLKDO1, CALC, CMND

LINE(80)  - (LOGICAL*1) - Holds command input line.
    USED IN:
       AT, BASCNG, CALC, DECLR, ERRCX, GETNNB, NEXTEL, SLEND, STRCMP

NONBLK  - (i*2) - Points to the last non-blank found in LINE(80).
    USED IN:
       AT, BASCNG, CALC, CMND, DECLR, ERRCX, GETNNB, INPOST, NEXTEL, STRCMP

VIEWSW  - (i*2) - View switch,
    0 = Output error messages
    1 = Output error messages and file command lines
    2 = Output error messages and value of expressions evaluated
    3 = Output everything
    USED IN:
       BLKDO1, CALC, CMND, DECLR, POSTVL
**CONS COMMON**

Constant commons for various characters and letters.

**ALPHA(27)** - (LOGICAL*1) - Capital letters of the alphabet plus % sign.
USED IN:
- AT, BLKDO1, CALC, DECLR, NEXTEL

**BLANK** - (LOGICAL*1) - Blank character, '

USED IN:
- AT, BLKDO1, CALC, DECLR, ERRCX, GETNNB, SLEND

**COMMA** - (LOGICAL*1) - Comma character, ',

USED IN:
- BLKDO1, DECLR

**EQ** - (LOGICAL*1) - Equal sign, '='

USED IN:
- BLKDO1, ERRCX, NEXTEL

**LPAR** - (LOGICAL*1) - Left parenthesis, ('`

USED IN:
- BLKDO1, ERRCX

**RPAR** - (LOGICAL*1) - Right parenthesis, ')

USED IN:
- BLKDO1, ERRCX, VAROUT

**D2R COMMON**

**NCDSP (DRW, DCL)** - (INTEGER) - Physical coordinates of displayed sheet.
USED IN:
- DSPSHT, RECALC, SPREDSHT, VARSCN, XQTCMD

**NRDSP (DRW, DCL)** - (INTEGER) - Physical coordinates of displayed sheet.
USED IN:
- DSPSHT, RECALC, SPREDSHT, VARSCN, XQTCMD
**DCTL COMMON**

**DCLV - (i*2)** - Display column used in DVS, number of maximum columns of display actually used for screen.

USED IN:
DSPSHT, SPREADSHT, XQTCMD

**DCOL - (i*2)** - Column of current cell on display sheet, active cell.

USED IN:
SPREADSHT, RECALC, VARSCN, XQTCMD

**DROW - (i*2)** - Row of current cell on display sheet, active cell.

USED IN:
SPREADSHT, RECALC, VARSCN, XQTCMD

**DRWV - (i*2)** - Display rows used in DVS, number of maximum rows of display actually used for screen

USED IN:
DSPSHT, SPREADSHT, XQTCMD

**PCOL - (i*2)** - Physical column used

USED IN:
CMND, SPREADSHT, RECALC, VARSCN, XQTCMD

**PROW - (i*2)** - Physical row used

USED IN:
CMND, SPREADSHT, RECALC, VARSCN, XQTCMD

**DECIDE COMMON**

**DTBL1(9.9.8) - (LOGICAL*1)** - Decision table for binary operations.

DTBL1(OPERAND2,OPERAND1,INDEX)

where: 
operator:
INDEX = 1 modify code for operand 1 */+-
2 modify code for operand 2 */+-
3 function value type */+-
4 operator class */+-
5 modify code for operand 1 **
6 modify code for operand 2 **
7 function value type **
8 operator class **

Where type codes (modify codes) are:

- 0 no change 5 convert to M10
- 1 convert to ASCII 6 convert to M8
- 2 convert to decimal 7 convert to M16
- 3 convert to hexadecimal 8 convert to octal
- 4 convert to integer 9 convert to real

For */+- function value type and operator class are presently identical.

For ** operator classes follow:

<table>
<thead>
<tr>
<th>CODE</th>
<th>OPERATOR</th>
<th>CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>REAL</td>
<td>** REAL</td>
</tr>
<tr>
<td>2</td>
<td>REAL</td>
<td>** INTEGER</td>
</tr>
<tr>
<td>3</td>
<td>INTEGER</td>
<td>** REAL</td>
</tr>
<tr>
<td>4</td>
<td>INTEGER</td>
<td>** REAL</td>
</tr>
<tr>
<td>5</td>
<td>M8</td>
<td>** INTEGER</td>
</tr>
<tr>
<td>6</td>
<td>M10</td>
<td>** INTEGER</td>
</tr>
<tr>
<td>7</td>
<td>M16</td>
<td>** INTEGER</td>
</tr>
</tbody>
</table>

USED IN:

BLKDO1, CALBIN

**DEFVBX COMMON**

DEFVB(12) - (LOGICAL*1) - Holds the default format for numerics, initially it will be F9.2. It shares the same common area as DVFMT and is surrounded with parenthesisses.

USED IN:

BLKDO1, SPREDSHT

DVFMT(10) - (LOGICAL*1) - Default format for numerics. Initially it is set to F9.2. It shares same common area as DEFVB and format is surrounded with parenthesisses. DVFMT is equivalenced with DEFFMT in XQTCMD in a way which eliminates the parenthesisses.

USED IN:

BLKDO1, SPREDSHT, XQTCMD
DIGV COMMON

DIGITS(16, 3) - (LOGICAL*1) - Constant common which holds decimal, octal, hexadecimal digits. The second subscript is
  1 for decimal
  2 for octal
  3 for hexadecimal

DOT COMMON

KDCL -(i*2) - Current working index into AVBLS, and TYPE which is used in RECALC and DOSTMT.
  USED IN:
    DOSTMT, RECALC

KDRW -(i*2) - Current working index into AVBLS, and TYPE which is used in RECALC and DOSTMT.
  USED IN:
    DOSTMT, RECALC

DSPCMN COMMON

CWIDS - (i*2) - CWIDS is the widths in characters of columns on display.
  USED IN:
    DSPSH, SPREADSHT, XQTCMD

DVS - (i*2) - Display array, which will keep a copy of variables displayed and formats used locally which display routine can use to see what actually needs to be refreshed on screen. DRWV, and DCLV are cols, and rows of display actually used for screen.
  USED IN:
    DSPSH, SPREADSHT, XQTCMD
**ERROR COMMON**

**LASTOP** - (i*2) - Holds the type of last element obtained on LINE(80). Set at 0 at beginning of expression. Used by NEXTEL to identify unary operators.

**USED IN:**
- BLKDO1, INPOST, NEXTEL

**FFGG COMMON**

**FORMFG** - (i*2) - Flags format of display globally.

**USED IN:**
- DSPSHT, SPREDSHT, EXQCMD

**PZAP** - (i*2) - Controls whether to redraw screen. If zero, normal. If 1 (non-zero) inhibits redisplay. V command resets to 0 and VM inhibits. (sets to 1)

**USED IN:**
- SPREDSHT, XQTCMD

**RCFGX** - (i*2) - Flags whether to do recalculation automatically. If 1, inhibits auto recalc (Use R command to do a calc.). RM turns RCFGX on.

**USED IN:**
- SPREDSHT, XQTCMD

**RCONE** - (i*2) - If set to one by R command, it forces recalculation of all of sheet. There is a comment in RECALC where RCONE is reset to zero, which states that force function works once only.

**USED IN:**
- RECALC, XQTCMD

**FVLDV COMMON**

**FVLD**(RRW, RCL) -(LOGICAL*1) - Flags contents of cells of sheet.

- 0 = No formula
- -1 = Display formula itself, not value
1 = Valid active formula there evaluate.
2 = Indicates already computed constants
Initially all 0's so initially ignore.

**ICPOS COMMON**

**ICPOS** - (i*2) - Allows UVT100 routine access to displayed numbers for uses such as setting colors.

**USED IN:**
- DSPSHT, XQTCMD

**IC2POS** - (i*2) - Allows UVT100 routine access to displayed numbers for uses such as setting colors.

**USED IN:**
- DSPSHT, XQTCMD

**ILN COMMON**

**ILINE**(106) - (LOGICAL*1) - ILINE is for programmable line input. (i.e. not from console). ILINE is present if ILNFG <> 0 and ILNCT has # bytes in it.

**USED IN:**
- CALC, DOSTMT

**ILNCT** - (i*2) - Length of ILINE if it exists.

**USED IN:**
- BLKDO1, CALC, DOSTMT, XQTCMD

**ILNFG** - (i*2) - Flags if the input line is to come from alternate source, not console. 1 indicates that line comes from alternate source, such as formula from sheet. 0 indicates input is from terminal. This would be used in CALC bare.

**USED IN:**
- BLKDO1, CALC, DOSTMT, XQTCMD
INITCOMMON

IBITMP(BRRCL) - (LOGICAL*1) - An initialized bitmap, used in DSPSHT to avoid going to disk for FVLD. Bitmap is zeroes in SPREDSHT main program (or at save command) and is set in DSPSHT only. Also only used in DSPSHT.

USED IN:
   DSPSHT, SPREDSHT

IOLVL COMMON

IOLVL - (i*2) - This is the LUN for XQTCMD to use (normally 3 for indirect files or 5 for terminal. We use 5,6 for terminal input, output normally.

USED IN:
   CMND, SPREDSHT, XQTCMD

ITERA COMMON

ITCNTV(6) - (i*2) - Indexed by level. 0 indicates that no iteration on the indirect command file is to take place. If positive, it holds the index into VBLS and represents the variable used to control iteration.

USED IN:
   AT, BLKDO1, CALC, CMND

KLVL COMMON

KLVL - (i*2) - For setting LEVEL, which tells where next calc command lines come from.

USED IN:
   CALC, SPREDSHT, XQTCMD
NMSH COMMON

**NMSH(80)** - (LOGICAL*1) - New default numeric format for a spreadsheet. May do other stuff.

**USED IN:**
- DSPSHT, SPREDSHT, XQTCMD

OAR COMMON

Common OAR switches output off is OSWIT =2.

**OARRY(100)** - (LOGICAL*1) - Holds output variable is OSWIT is nonzero.

**USED IN:**
- VAROUT

**QCNTR** - (i*2) - Holds bytes valid in OARRY (up to 100, no more)

**USED IN:**
- BLKDO1, VAROUT

**OSWIT** - (i*2) - Initially set to zero. If OSWIT is nonzero, output is switched off, and output goes to OARRY. OSWIT=2, means no zeroing of OARRY, nothing much comes out.

**USED IN:**
- BLKDO1, SPREDSHT, VAROUT, XQTCMD

STACK COMMON

**ST1LIM** - (i*2) - Holds size of STACK1 (always constant)

**USED IN:**
- BLKDO1, INPOST

**ST1PT** - (i*2) - Points to the top of STACK1 (changes as stack is used)

**USED IN:**
- CALBIN, CALUN, INPOST, POSTVL
ST1TYP(40) - (i*2) - Data type of each element in STACK1
Used in:
   CALBIN, CALUN, INPOST, POSTVL

ST2LIM - (i*2) - Holds size of STACK2 (always constant)
Used in:
   BLKDO1, CALBIN, INPOST, POSTVL

ST2PT - (i*2) - Points to the top of STACK2 (changes as stack is used)
Used in:
   CALBIN, CALUN, INPOST, POSTVL

ST2TYP(40) - (i*2) - Data type of each element in STACK2
Used in:
   CALBIN, CALUN, INPOST, POSTVL

STACK1(100.40) - (LOGICAL*1) - Utility stacks used when evaluating expressions. The first subscript controls indexing across the bytes of a single variable. The second subscript controls stack elements.
Used in:
   CALBIN, CALUN, INPOST, MULADD, MULDIV, MULMUL, POSTVL

STACK2(100.40) - (LOGICAL*1) - Utility stacks used when evaluating expressions. The first subscript controls indexing across the bytes of a single variable. The second subscript controls stack elements.
Used in:
   CALBIN, MULADD, MULDIV, MULMUL, POSTVL
V COMMON

AVBLS(100,27) - (LOGICAL *1) - Holds the values of the 27 legal variables. (accumulators). Initialized in BLKDO1 as follows.
AVBLS(1,27)/6/, AVBLS(2,27)/0/, AVBLS(3,27)/0/, AVBLS(4,27)/0/.
USED IN:
   BLKDO1, CALBIN, CMND, DOMFCN, DOSTMT, POSTVL, RECALC,
   SPREDSHT, VAROUT, XQTCMD, USRFCT, ZERO, ZNEG

TYPE(RRW.RCL) - (LOGICAL *1) - Holds the data types for each of the 27 variables. See CODES.FTN for the possible values.
USED IN:
   BLKDO1, CALBIN, CMND, DECLR, DOSTMT, DSPSHT, MOUT,
   NEXTEL, POSTVL, VAROUT, XQTCMD, ZERO, ZNEG

VBLS(8,RRW.RCL) - (LOGICAL*1) - Holds values of all variables. Constants are stored in VBLS according to their type. See BLKDO1 for further description of how they are stored.
USED IN:
   CALBIN, CMND, DOMFCN, DOSTMT, DSPSHT, SPREDSHT,
   XQTCMD, USRFCT, ZERO

VLEN - (i*2) - Indexed by data type, VLEN gives the number of bytes used by that data type. It is defined in BLKDO1 as follows:
   DATA VLEN /1,8,4,4,100,100,100,4,8/
USED IN:
   BLKDO1, CALBIN, INPOST, POSTVL, NEXTEL
GLOBAL PARAMETERS
Set in VVKLUGPRM

MXCOLS - Maximum number of columns, must be less than or equal to DCL.

MXROWS - Maximum number of rows, must be less than or equal to DRW.
This is for initial display.

MXCOLS - Maximum number of columns, must be less than or equal to DCL.

RRW - Number of real rows, physical rows of spreadsheet, must be 1
larger than 27 to handle 1st 27 variables in AVBLS

RCL - Number of real columns, physical columns of spreadsheet, must be at
least 2, the first are the accumulators and the second are display
coordinates.

DRW - Display maximum rows, must be less than or equal to RRW.

DCL - Display maximum columns (across top), must be less than or equal to
RCL.

RRCL - RRW times RCL

BRRCL - (RRCL+7)/8 - Enough bytes for a bitmap.
APPENDIX D.

DATA FLOW DIAGRAMS
CONCEPT DIAGRAM

USER

SCREEN

COMMANDS FILE

SPREADSHEET

FILE COMMAND LINE

FILE COMMAND LINE

UPDATE CELL CONTENTS

EXISTING CELL CONTENTS

EXISTING SPREADSHEET FILE

NEW SPREADSHEET FILE

SCREEN DISPLAY

SCREEN CODES

SCREEN CELL CONTENTS

CURRENT CELL CONTENTS

EXISTING CELL CONTENTS
Diagram 2. Calc

2.1 Check if Star Command

2.2 Star Command Processor CMND

2.3 Get Next Element

2.4 Put into Postfix Notation INPOST

2.5 Evaluate Postfix Expression POSTYL

CELL

EXPRESSION

LINE

ELEMENT

STACK YARS

CELL RESULTS

SHEET VALUES

SHEET FLAGS

ERROR

CMND

CMND
Diagram 3. Recalc

3.1 Determine if Formula to Evaluate

3.2 Check for Multiple Expressions

3.3 Check for Function or If or Cell Expression

3.4 Perform Mathematical Functions

3.5 Store Result

3.6 Process If Statement

EXISTING CELL CONTENTS

FLAG OF CONTENTS

CELL FORMULA

_SINGLE EXPRESSION

FUNCTION EXPRESSION

IF EXPRESSION

FUNCTION RESULT

CELL RESULTS

SHEET VALUES

DOENTR

EXPRESSION COMPLETE

DOENTR
APPENDIX E.

DATA DICTIONARY

FOR

DATA FLOW DIAGRAMS
Appendix E

DATA DICTIONARY

Key of symbols used

- IS EQUIVALENT TO
+ AND
() ITERATIONS OF
() OPTIONAL
| SEPARATOR
[...|...] EXCLUSIVE OR
[...+] INCLUSIVE OR

DESC = descriptions of data entry
esc = escape character, ASCII character 27
SYN = synonym of entry

CALC UPDATE CELL CONTENTS = CELL CONTENTS
  DESC = Updates of a cell contents as performed from CALC.

CELL CONTENTS = [CELL VALUE | CELL FORMULA] + DISPLAY FORMAT +
  CHECK FLAG + VALIDITY FLAG + OTHER FLAGS
  DESC = This is the record which stores a cell's contents. It is 128 bytes
  long, with 109 bytes for the formula or value, one byte for check, one byte
  for validity, eight bytes for display format, and the rest of bytes for flags.

CELL EXPRESSION = [LINE | STAR CMND]
  DESC = This is the expression that needs to be evaluated by CALC, it is
  either an expression to be evaluated or a star command to be executed by
  CMND.

CELL FORMULA = (SINGLE EXPRESSION)
  DESC = This is where the formula for a cell is stored, it may contain
  more than one expression to evaluate.

CELL RESULTS = (VALUE)
  DESC = This is the result of the evaluation of the CELL EXPRESSION.
**CELL VALUE** = VALUE  
**DESC** = The value associated with the cell.

**CELL VAR** = [A-Z, A1-Z1, AA1, AB1... | D#nn#mm | p#nn#mm]  
**DESC** = A-Z represent the 26 accumulators in the program  
A1-Z1 etc. represent the labels of cells.  
D#nn#mm represent locations relative to the current cell on the display spreadsheet. nn and mm represent the number of columns and rows respectively from the current cell. Negative values of nn indicate the number of columns to the left and positive integers columns to the right of the current cell on the display sheet. Negative values of mm indicate number of rows up with positive values indicating the number of rows below the current row on the display sheet.  
P#nn#mm represent locations relative to the current cell position on the physical spreadsheet. The meaning of nn and mm correspond with those of the D#nn#mm form except the current cell on the physical sheet is used instead of the display sheet.

**CHECK FLAG** = INTEGER  
**DESC** = This is a magic number to check that all is well and that one has the correct type of file. Byte 118 of record must contain 15 in this implementation.

**CMND COMMAND**  
**DESC** = These are the commands from the CALC portion of the system. They have direct control of the CALC program. See CMND SUBROUTINE COMMANDS table in Appendix ZZ for the list of commands and their syntax.

**COMMAND LINE** = [XQTCMD COMMAND | CMND COMMAND | EXE SCREEN CMD | EXPRESSION]  
**DESC** = These are the commands into the system.

**CURRENT CELL CONTENTS** = CELL CONTENTS  
**DESC** = This is how part of the spreadsheet can be saved for future use. The formulas can be saved or just the numerical values.

**DISPLAY FORMAT** = FORTRAN display formats  
**DESC** = This is where the display format for a cell is stored, it is a
regular FORTRAN format. The initial default format is F9.2

**DISPLAY SCREEN CMD** = SCREEN COMMANDS
   DESC = Modify display commands from display spreadsheet module.

**ELEMENT** = [RETURN VALUE + RETURN TYPE + RETURN CODE]
   DESC = This is an element of the expression with information to tell
      what kind of element it is.

**EVALUATED VALUE** = VALUE
   DESC = This is the result of evaluating the expression.

**EXE SCREEN CMD** = SCREEN COMMANDS
   DESC = Modify display commands from command processor.

**EXE UPDATE CELL CONTENTS** = CELL CONTENTS
   DESC = Updates of a cell contents as performed from command
      processor.

**EXISTING CELL CONTENTS** = CELL CONTENTS
   DESC = This is the input of cell contents to the program from an existing
      spreadsheet file.

**EXISTING SPREADSHEET FILE** = (CELL CONTENTS)
   DESC = This is a direct access file made up of records 128 bytes long.
      There is a record for each cell in the physical sheet. If the sheet was five
      rows by ten columns, this file would contain 50 records. This is both a
      source and sink of information from the system.

**EXPRESSION** = FORTRAN EXPRESSION with exceptions noted below.
   DESC = These are the expressions which can be evaluated by CALC.
      They have a syntax similar to FORTRAN syntax. A derivation from
      FORTRAN syntax is variable names. The names of the PORTACALC
      functions are reserved names. The valid variable names are explained in
      CELL VAR.

**FILE COMMAND LINE** = COMMAND LINE
   DESC = These are commands coming from a file of commands.
FLAG OF CONTENTS = INTEGER
  DESC = This integer code flags what is in a cell, a formula, a value, or a constant which has been evaluated.

FORMULA = EXISTING CELL CONTENTS
  DESC = This is what the the recal module gets to evaluate from formula file. At this point we know that there is a formula in existing cell contents.

FUNCTION_ARGS = { VALUE }
  DESC = These are arguments for PORTACALC functions.

FUNCTION_EXPRESSION = FUNCTION_CODE + FUNCTION_ARGS
  DESC = This single cell expression includes a PORTACALC function and the arguments for that function.

FUNCTION_CODE = {1 | 2 | 3 | 4 | 5 | 6}
  DESC = Each integer represents a different mathematical function. In PORTACALC the number correspond as follows:
  1 = MAXIMUM VALUE
  2 = MINIMUM VALUE
  3 = AVERAGE VALUE
  4 = SUM
  5 = STANDARD DEVIATION

IF_EXPRESSION = LOGICAL_EXPR + EXPRESSION
  DESC = A FORMULA which has an IF function in it to perform.

IF_COMPLETED_EXPRESSION = EXPRESSION
  DESC = The remaining part of a FORMULA which is to be evaluated since the logical expression of the IF evaluated to true.

INTEGER = decimal number
  DESC = This can be displayed in various formats using different bases. The bases available could be base 10, 8, and 16.

KEYBOARD_COMMAND_LINE = COMMAND_LINE
  DESC = These are the command typed in from the terminal keyboard.

LOGICAL_EXPR = BOOLEAN_EXPRESSION
  DESC = This is logical or boolean FORTRAN expression used to determine if the then portion of the statement is to be performed.
LINE = EXPRESSION
   DESC = This is the line which is to be evaluated, one knows it is an expression to be evaluated.

MULTIPRECISION NUMBER = Multiple precision number
   DESC = This is for multiple precision, number can be base 8, 10, or 16.

NEW SPREADSHEET FILE = ( CELL CONTENTS )
   DESC = This is where one can store information about a sheet or part of a sheet which one is working on. One can store the CONTENTS of sheet or just the numerical values. This is a sink for information from the system.

REAL = floating point number

RETURN CODE = INTEGER
   DESC = Return code:
      1 If operand (value is to be returned in RETURN VALUE)
      2 If operator (value is to be returned in RETURN TYPE)
      3 If no more elements
      4 If error

RETURN TYPE = INTEGER
   DESC = This is the type code of the element found. See TYPE for listing of codes.

RETURN VALUE = VALUE
   DESC = This is the value of the variable.

SCREEN = a sink for information
   DESC = This is a terminal screen which displays information about the system. It can be one of many terminals including VT52 or VT100.

SCREEN CELL CONTENTS = CELL CONTENTS
   DESC = This is what is sent to screen for display.

SCREEN CODES = esc + letter
   DESC = This is terminal dependent, this is what is actually sent to terminal screen for manipulation functions. These functions include erasing all or parts of the screen, setting mode of screen, or movement of cursor on screen.
SCREEN COMMANDS = SCREEN PARAMETERS + (INTEGER + (INTEGER))

DESC = Screen commands are coded by SCREEN PARAMETERS and the
two optional integers are arguments needed for some of the commands.

SCREEN PARAMETERS = [1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14
| 15 | 16 | 17 | 18]

DESC = Coded parameters, each integer stands for a different command.
The list of commands is found in the UVT100 routine description in
Appendix B.

SCREEN DISPLAY = What is displayed on screen by the program.

SHEET FLAGS = (FLAG OF CONTENTS)

DESC = This is an array, one element for each cell on the sheet, which
flags what is in the cell. In PORTACALC, 0 = no formula, -1 = display
formula itself, not value, and 1 = valid active formula to evaluate.

SYN = FVLD

SHEET VALUES = VALUE + TYPE

DESC = This is equivalent to VBLS and TYPE variables in PORTACALC. It
is an array the size of the spreadsheet. More information on VBLS and
TYPE can be found in Appendix C, Common Data Dictionary.

SYN = VBLS and TYPE

SINGLE EXPRESSION = [ CELL EXPRESSION | FUNCTION EXPRESSION | IF
EXPRESSION ]

DESC = FORMULA clamped to one expression to evaluate.

STACK = (VALUE | OPERATOR)

DESC = This is where the value and operators of an expression are
stored in STACK form.

STACK LIMIT = INTEGER

DESC = Maximum height of STACK.
STACK POINTER = INTEGER
   DESC = Pointer to current top of STACK

STACK TYPE = INTEGER
   DESC = Type of each element on the STACK.

STACK VARS = STACK + STACK POINTER + STACK LIMIT + STACK TYPE
   DESC = This is where the expression to be evaluated is stored in postfix notation.

STAR CMND = CMND SUBROUTINE COMMANDS
   DESC = These are when the LINE contains an asterisk as it's first non-blank character. See CMND SUBROUTINE COMMANDS table in Appendix ZZ for a list of commands.

TYPE = INTEGER
   DESC = This is a flag which is used to identify what type of entity is stored at this location in the spreadsheet. In PORTACALC the codes used are as follows:
      0 = Undefined
      1 = ASCII
      2 = Decimal
      3 = Hexadecimal
      4 = Integer
      5 = Multiple Precision - base 10
      6 = Multiple Precision - base 8
      7 = Multiple Precision - base 16
      8 = Octal
      9 = Real

UPDATE CELL CONTENTS = [ CALC UPDATE CELL CONTENTS | EXE UPDATE CELL CONTENTS ]
   DESC = This is where a cell's contents are updated in the existing spreadsheet file.

USER = the user of the program
   DESC = A sink and source of information.

VALIDITY FLAG = INTEGER
   DESC = This is used to allow handling reads.

VALUE = ['INTEGER | REAL | OCTAL | HEX | M10 | M8 | M16]
   DESC = These are the possible types of values an expression can take on.
XQTCMD COMMAND

DESC = These are the manipulation commands of PORTACALC. See
XQTCMD SUBROUTINE COMMANDS table in Appendix ZZ for the list of
commands and their syntax.
APPENDIX F.

MINI-SPECIFICATIONS

OF THE PROCESSES

IN THE

DATA FLOW DIAGRAMS
Appendix F

Mini-specifications of the Processes in Data Flow Diagrams

Command Processor = Process #1
DESC = This process executes PORTACALC commands into the system. See XQTCMD SUBROUTINE COMMANDS in APPENDIX G for description of the command and their syntax.
SYN = XQTCMD

Check if asterisk command = Process #2.1
DESC = This process checks to see if REGULAR EXPRESSION contains an asterisk as its first non-blank character. If * is the first non-blank character then REGULAR EXPRESSION is sent out as a STAR COMMAND. These commands are detailed in CMND SUBROUTINE COMMANDS in Appendix G. If no * is found, then the line is to be evaluated, and REGULAR EXPRESSION is OUTPUT as LINE.

Star command processor = Process #2.2
DESC = This is the command processor for the CALC portion of the program. See CMND SUBROUTINE COMMANDS in Appendix G for description of command and their syntax.
SYN = CMND

Get next element = Process #2.3
DESC = This process gets a valid variable name or operator from LINE. If a variable name, then it accesses SHEET VALUES to get its value. The ELEMENT is then output.
SYN = NEXTEL
Translate from infix notation to postfix notation on STACK = Process #2.4
   DESC = This process accepts ELEMENTS, translates them onto STACK in postfix notation for later evaluation.
   SYN = INPOST

Evaluate postfix expression on STACK = Process #2.5
   DESC = This evaluates the postfix expression on STACK with the use of a second stack. VALUE is output.
   SYN = POSTVL

Determine if function to evaluate = PROCESS #3.1
   DESC = Checks FLAG OF CONTENTS for a given cell to see if there is a formula that needs to be evaluated. If so, read OLD CELL CONTENTS for that cell and pass it out as FORMULA.

Determine if multiple expression in formula = Process #3.2
   DESC = Checks FORMULA for a \ which indicates a multiple expression in FORMULA. If a \ is found, then send the first expression on as EXPRESSION, if \ is not found send on FORMULA as EXPRESSION.

Determine if EXPRESSION includes a function or is a regular expression = Process #3.3
   DESC = If EXPRESSION has a PORTACALC function in it, send the EXPRESSION as FUNCTION EXPRESSION to Process #3.4, If an IF is found, EXPRESSION as IF EXPRESSION to Process #3.6 and if no function name is found send EXPRESSION as REGULAR EXPRESSION to CALC for evaluation.
Perform mathematical function = Process #3.4
   DESC = This process accepts FUNCTION EXPRESSION, which contains
FUNCTION CODE and FUNCTION ARGS. The FUNCTION CODE is as follows:
   1 = Minimum function  2 = Maximum function
   3 = Average function   4 = Sum function
   5 = Standard deviation function
Then it performs the appropriate mathematical function.

Store results = Process #3.5
   DESC = Uses current cell coordinates to determine where to store result
in SHEET VALUES.

Do IF commands = Process #3.6
   DESC = Performs the necessary operations to handle the IF statement.
   If must check to see if logical expression is true. If true pass an expression
to Process 3.3 to continue evaluation.

Display spreadsheet = Process #4
   DESC = See routine description of DSPSHT in Appendix B for details.
   SYN = DSPSHT

Translates video display commands = Process #5
   DESC = See Routine description of UVT100 in Appendix B for details.
   SYN = UVT100
Appendix G
Command Descriptions

1. XOTCMD SUBROUTINE COMMANDS

Key for syntax of commands

* = NUMBER, can be positive or negative
r = row
c = column
var = variable label such as B4
' = the apostrophe acts as a delimiter
V1:V2 = a range of cells, must be in the same physical row or column
V3:V4 = a range of cells, must be in the same physical row or column

ADD/REMOVE ROW/COLUMNS

AA # r = Add/remove absolute rows, no relocation of variables
AA # c = Add/remove absolute columns, no relocation of variables
AR # r = Add/remove relocating rows, relocate all variables below
AR # c = Add/remove relocating columns, relocate all variables below
If # is positive, add row/columns, if # negative, remove row/columns.

COPY CELL COMMANDS

CA v1:v2 v3:v4 = Copy all cell attributes from v1:v2 to v3:v4
CV v1:v2 v3:v4 = Copy numeric values only
CD v1:v2 v3:v4 = Copy display formats only
CF v1:v2 v3:v4 = Copy formulas only (no relocation)
CR v1:v2 v3:v4 = Copy all cell attributes as in CA but relocate
cell names to new location from old one.

If V2 is not present, just one cell gets copied.
DISPLAY FORMATTING COMMANDS

DF v1:v2 [fmt] = Display format, v1:v2 is the range to be changed, and fmt is the desired format to display.

DT v1:v2 type = Set range from v1 to v2 to designated data type.
   type = F means to set type to floating data type
   type = I means to set type to integer data type

DW N, M = Set column N on display to be M characters wide

DB n, m = Set number of columns displayed on screen to n and number of row on screen to m

DROP INTO CALC BARE COMMAND

K = Drop into CALC with direct control
   One can access cells of the spreadsheet from CALC in this mode.

ENTER COMMANDS

EN = Enter expression, may be numbers or text

ED 'STRING1'STRING2' = Replaces STRING2 for STRING1 in current cell formula

EXIT COMMANDS

X = Exit to operating system

XD = Exit to operating system after deking random access scratch file used to hold spreadsheet formulas/number/formats

GET COMMANDS

GD = Reload sheet down/right onto display coordinates

GP = Reload sheet down/right onto physical coordinates

G = Get input numbers off sequential file, use current origin

HELP COMMAND

H = Prints help information
MOTION DIRECTION

This command sets the default direction for automatic motion after E commands.

M1 = move up
M2 = move down
M3 = move left
M4 = move right
M5 = default, not to move

MOVE CURSOR

1 = Move up one cell
2 = Move down one cell
3 = Move left one cell
4 = Move right one cell
L var = Move to cell var

ORIGIN set of display sheet

OA = Moves origin to 1,1 on physical sheet, move absolute
OR = Moves origin to current display position, DROW, DCOL, move relative

RECOMPUTE COMMANDS

R = Recalculate all of sheet
RM = Recalc manually only when R command is given
RAF = Recalculate automatically

SAVE and PUT COMMANDS

PD = Write out formulas of current display sheet to a file
PP = Write out formulas of current physical sheet to a file
PDN = Write out numbers of current display sheet to a file
PPN = Write out numbers of current physical sheet to a file
S = Save work file, allow you to use a new one
VIEW SCREEN UPDATE COMMANDS

V = Redraw screen with normal formats
VF = Redraw screen showing all formulas
VM = Don’t redraw screen until a V or VF is given

WRITE COMMAND

W = Write (print) screen out to file (may be printer)

ZERO COMMANDS

ZA = Zero whole sheet
ZE v1:v2 = Zero variables in range of v1 to v2
2. CMND SUBROUTINE COMMANDS

These commands can occur three ways, in a formula in a cell from the spreadsheet, from a file of commands directed to CALC, or from keyboard in direct control of CALC, such with K command from PORTACALC.

Key

- var(s) = Zero or more variables separated by a comma
- V1 = Alone represents a variable name such as B5
- filename = This is the name of a file to be accessed from the program.

ACCESS FILES OUTSIDE CALC OR PORTACALC

*F LABLE = If the value in % is positive and nonzero the command rewinds the input file for the PORTACALC command and seeks a line beginning with the character *CLABLE. This allows CALCV commands inside a cell to direct outside command entry.

*J LABLE = Same as *F but used CALC command instead.

ACCESS USER FUNCTIONS

*U function args = Allows user to access user defined functions

BASE CHANGE COMMANDS

*B = Displays current default base
*B 8 = Changes default base to octal
*B 10 = Changes default base to 10
*B 16 = Changes default base to 16

COMMENTS

*C = Comment line, characters that follow are ignored by CALC

DECLARE ASCII VARIABLES

*ASCII var(s) = Declares variable(s) to be of type ASCII
DECLARE TYPE OF VARIABLES REAL AND DECIMAL

*REAL = Declares variables to be of type REAL*8. When these variables are output, FORTRAN's D format is used.
*DECIMAL = Declares variables to be of type REAL*8. When these variables are output, FORTRAN's F format is used.

DECLARE VARIABLES TO A BASE

*INTEGER var(s) = Declares variable(s) to be a base 10 integer
*HEX var(s) = Declares a variable(s) to be a base 16 integer
*OCTAL var(s) = Declares a variable(s) to be a base 8 integer
*M8 var(s) = Declares a variable(s) to be a multiple precision base 8 number
*M10 var(s) = Declares a variable(s) to be a multiple precision base 10 number
*M16 var(s) = Declares a variable(s) to be a multiple precision base 16 number

If var(s) is is zero variables, then these commands list all variables that have been declared to be of that type.

EXAMINE SEQUENTIAL FILES OUTSIDE PORTACALC

*QF filename ?key1? ?key2? <lm>
*QW filename ?key1 ?key2? <lm>

FORMULA FROM ANOTHER SAVED SPREADSHEET FILE

*XV filename V1 = Loads value from the V1 position of another NUMERICALLY saved spreadsheet file.
*XF filename V1 = Load formula from the V1 position of another saved spreadsheet file.

These commands can be used to link sheets together.

GET VALUE FROM SHEET

*G V1, V2 = (V1 and V2 are either cell or accumulator names) Gets cells value from spreadsheet where V1 and V2 are the column and row numbers respectively, or form accumulator variable V1
INDIRECT FILE OF COMMANDS
   *filename = CALC reads from this file and executes the commands found there

P COMMANDS
   This command resets the current cell coordinates from within a cell.
   *P = Cause CALC to prompt used for new physical column and row.
   *P V1 = This moves the current location to the named cell V1

READ NEXT COMMAND FROM TERMINAL
   *R = Allows for a single line to be read from the terminal.

SET PRINT OPTIONS
   Controls CALC's printing options, default is *V
   *V 0 = Print error messages
   *V 1 = Print error messages, command lines read from files
   *V 2 = Print error messages and value of expressions evaluated
   *V 3 = Print error messages, command lines read from a file, and value of expressions evaluated
   *V = Same as V3
   *N = Same as V1

TERMINATE CALC SESSION
   *E = Terminates CALC session unless it is used within a file of CALC commands. Then CALC closes the file and continues with the next command.
   *S = Same as *E

WRITE
   *W = Takes value at current cell and writes it out to the formula file at current position.

ZERO VARIABLES
   *Z = Zeroes all variables except $, the accumulator. The data types of variables remain unchanged.
APPENDIX H.

RXVP80™,
A SOFTWARE ANALYSIS,
DOCUMENTATION, AND
TEST SYSTEM
Appendix H

RXVP80™, A Software Analysis

Documentation, and Test System

This appendix describes RXVP80™ and its reports, which can assist in analysis and specification. RXVP80™ is a tool for analyzing source code written in FORTRAN, IFTRAN™, or V-IFTRAN™. It can perform the following operations:

- Syntax and structural analysis of source programs.
- Static analysis to detect inconsistencies in program structure or in the use of variables.
- Automated documentation.
- Instrumentation of source code.
- Analysis of testing coverage.
- Retesting guidance.

Thus RXVP80™ can be used in various parts of the software development cycle, such as analysis, documentation, and testing.

RXVP80™ generates a data base of information as each module is analyzed. A number of reports can be generated from inquiries into this data base. The rest of this appendix describes reports which were used in the reverse engineering effort on PORTACALC.
LIST REPORT

The LIST request gives a program listing which is numbered for ease of indexing. This listing also has all levels of nesting indented for ease of understanding. Both local and global reports can be generated. Local reports refer to the subroutine just entered, and global refers to the current complete data base.

INVOCATION REPORTS

The INVOCATION SPACE report, the INVOCATION BANDS report, and the INVOCATION SUMMARY report help denote the systems structure. The INVOCATION SPACE report shows all invocations of other subroutines from the given subroutine, along with all routines which call the given subroutine. The statement numbers of each invocation is given. The INVOCATION BANDS report shows the various levels of invocation of routines which call and are called by the given subroutine. These reports are of much assistance when developing a hierarchical structure of the program. The INVOCATION SUMMARY report lists each subroutine and then lists all routines which call and are called by that particular subroutine.
SYMBOL REPORT

The SYMBOLS report shows the name, type, scope, mode, and use of each symbol in the module. The use portion of the report tells if a variable is set, used, or equivalenced. This is helpful when trying to understand how the variables are used and where they are modified. The LOCAL CROSS REFERENCE report goes further by giving line numbers where each of the variables are used. These reports promote the understanding of the data structures of the program.

REPORTS DEALING WITH COMMON VARIABLES

More information of data flow and data usage needs to be known to gain a complete understanding of a system. Along these lines the interfaces between modules need to be understood. The COMMON MATRICES report and COMMONS CROSS REFERENCE report aid in this information gathering effort. The COMMON MATRICES report lists all common areas with their elements. The report then indicates the use of each element within each subroutine, i.e. it tells if variable is set, used, and/or equivalenced. This is done in a graphical way, so one can see which subroutines use which common variables. This depicts the common
environmental coupling. The COMMONS CROSS REFERENCE report lists all of the common variables with each of their elements and indicates the scope of each variable, along with how it is used, set, or equivalenced. Each line where the variable occurs is indicated. This is very essential information when constructing data usage and data flow information.

**INPUT/OUTPUT REPORTS**

The INPUT/OUTPUT STATEMENTS report lists all of the input and output statements in the program. This is very helpful in understanding the file usage along with the input/output form of the program.

**SUMMARY REPORTS**

The SUMMARY requests provides an analysis of program, giving percentages of program having various characteristics. It gives interface characteristics by showing number of common, entry, and exit variables. The SUMMARY report also gives the percentage and number statement of given classification and type. The amount of documentation is shown as well.
GENERAL INFORMATION

The program functions of the systems are not directly found using RXVP80™. Through the use of an analyzer program, like RXVP80™, one will get a much better understanding of how the program is put together, and how it works. The functions of the system, will be much more self-evident after using an analyzer program such as RXVP80™.
Bibliography


