Online Database: Design and Optimization

Robert B. Garvey
Witan Inc.
Kansas City, Missouri

CONTENTS

FOUNDATION

A system language, GOALS, will be introduced to render systems and components.

A general set of principles will be presented incorporating the components and structures inherent in a structured system. The use of these components in the system life cycle and as a documentation system will evolve.

A general system architecture will be presented and an approach to interactivity will be discussed.

The detailed use of callable programs in the 3000 environment will be discussed with emphasis on improvement of system performance.

I am going to assume that you are first time users of a 3000 that you want to write online database systems, that you do not have some of the more typical real world problems like a conversion from another machine and that you are going to use VPLUS and IMAGE. I don't care what language you use unless it is RPG in which case much of what I say will not be true.

GOALS: A System Language

GOALS was designed to meet the following criteria:

- Provide good documentation throughout the lifecycle
- Ease maintainence
- Expedite development
- Provide users early understanding of System functions and restraints
- Improve project management and reporting
- Reduce resources required
- Optimize System performance and quality

Many of the above criteria can be achieved through reasonable structuring of the system . However many of the structuring techniques that are now popular are simply more trouble than they are worth. Yourdon, Jackson and certainly IBM's HIPO involve more work involve more work in their maintainence than rewards merit. Warnier comes closest to being worthwhile but cannot be reasonably maintained in machine sensible form.

GOALS will be described as a methodology only because it does what the popular "Methodologies" tout, and much more. We do not feel that any of the methodologies should be considered ends in themselves and more sacred than the system at hand. Once the principles are learned and applied the implications should be obvious and the apparent need for a methodology forgotten.

Documentation

General Statement

The purpose of documentation is to assist in the analysis, design, program design, maintenence and operation of a system. To those ends software documentation must be flexible, easily modifiable, current and easy to read. Witan has developed a system of documentation called GOALS which uses simple text files associated through control numbers to meet the criteria listed above. The following sections describe

the general features of the structural notation used in GOALS and the General system structure used in system projects.

GOALS is used throughout the life of a project. It is used:

1. To state requirements
2. Render flow and components in the analysis phase
3 To develop, test and render a general design
4. As a pseudo code or structured English for detail design
5. As a high level programming language
6. As a project network descriptor.

GOALS: Structural Notation

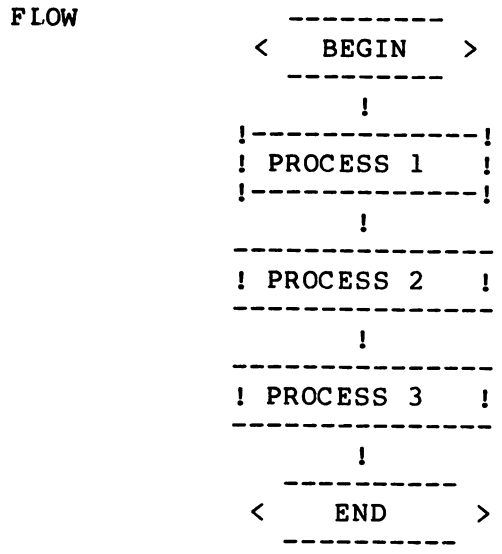
Formal structuring permits three primitive operations: Sequence, Repetition and Alternation. Structural Notation was developed to meet the criteria of formal systems in a generalized way and was guided by the assumption that systems must be rendered in a machine sensible form. GOALS relies upon text sequences and key words as its basis. Structural Notation is the basis of the syntax of GOALS.

Following are the representations of the primitive structures using flowcharts and GOALS. The word PROCESS is used to represent a step, a process or an item depending on the use of the notation at the time.

GOALS Primitive Structures

SEQUENCE

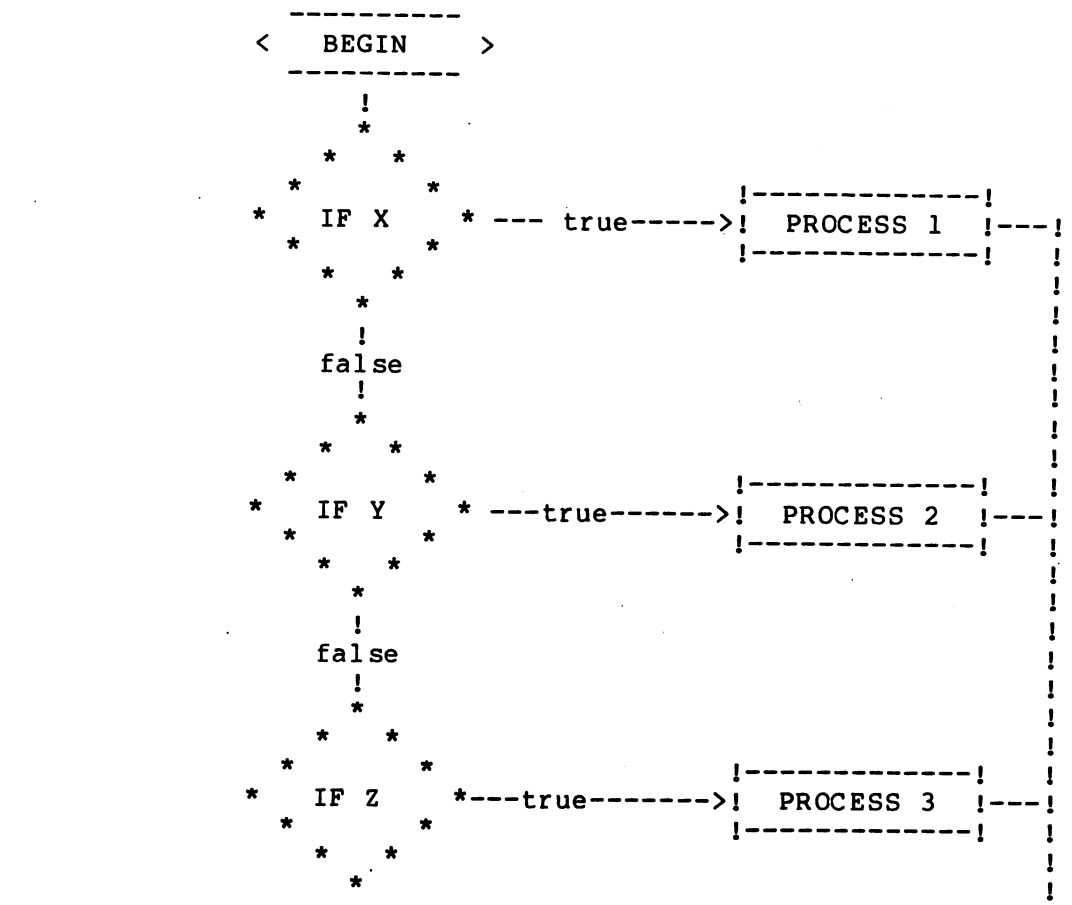
FLOW

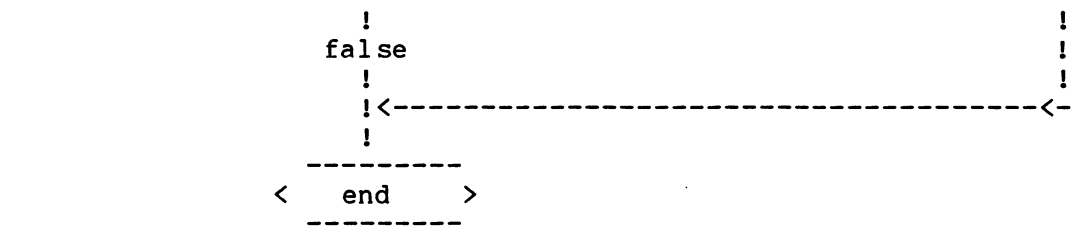



```
GOALS       1  PROCESS 1
            2  PROCESS 2
            3  PROCESS 3
```

ALTERNATION

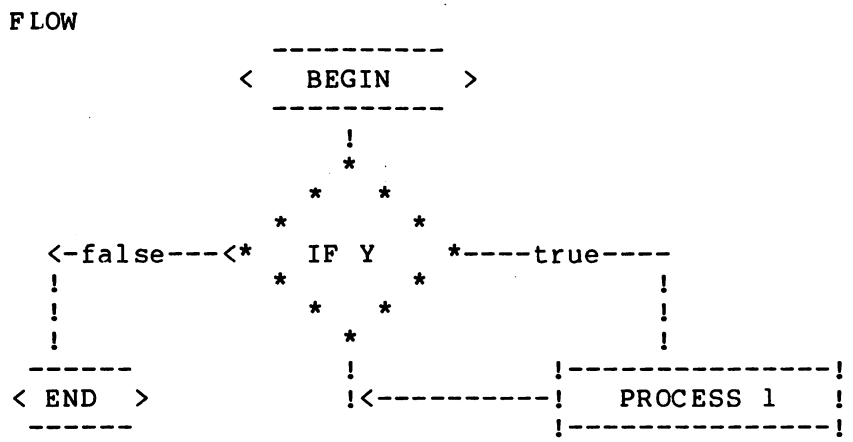
FLOW






```
GOALS       IF X IS TRUE
               PROCESS 1
            IF Y IS TRUE
               PROCESS 2
            IF Z IS TRUE
               PROCESS 3
            .
```

REPETITION




```
GOALS       WHILE  Y <<  IS TRUE  >>
               PROCESS 1
                  PROCESS 1A
                  PROCESS 1B
                  PROCESS 1C
            !
```

The exclamation point is used to signify control in the WHILE loop. If the condition is met the control passes to the statement following the (!) on the same level. If the condition is met the control passes to the first statement following the condition. Processes 1A through 1C were added to show a simple subsequence.

Data Structuring

GOALS is also used to represent data structure. As with control structure there are three general structures which can be represented.

Data items listed line after line represent sequence:

1. item-1
2. item-2
3. item-3

Subsequences are represented as sequences on a level below the item of which they are are a part.

1. item-1
   1A. item-1A
   1B. item-1B
   1C. item-1C
2. item-2
3. item-3

Repetition in data structuring can be represented by "(S)" at the end of the item name which is repeated, this can take the form an expression [i.e., (0>s<15)].

item-1(S)
  item-1

Example: a file of accounts

  Account File
Account(s)
  Account
  Account number

Name
Address(s)
  Address type (h=home, w=work)
  Street number
  Direction
  Street name
  Affix
Amount due
Order(s)
  Order number
  Item(s)
    Item

Alternation

Alternation is represented with the IF control word or with the notation (1,0).

IF segment descriptive code = 1
    material
IF segment descriptive code = 2
    supply

This convention is seldom used because the WHILE handles most situations for the case of data structuring.

The other type of alternation is within a string of data items where the item can either exist or not exist. Another way of representing a non-required item.

1. item-1
2. item-2
3. item-3(1,0)

This says that items 1 and 2 must exits or are required and item 3 is optional.

Discussion

The highest level of repetition within a data structure is assumed to be the key to the file or at least the major sort sequence. If additional keys are required they can be represented with the word KEY [i.e., item-3 (KEY)] or an additional data structure can be presented to represent the structure represented when the KEY is used.

GOALS can be used to represent logical structures as well as the physical implementations. It is important that the required logical views of data be derrived and documented before any physical structures be planned. A goal in system design is to have a one to one relationship between the physical and the logical structures of the system. The coding complexity is reduced appreciably as wellas the maintainence activity. An additional byproduct is the ability to use Query or other general inquiry languages in a more straight forward fashion.

LEVELS: are represented graphically with the use of indentation. The first character in a line is considered to begin an "A" level subsequent levels are indented an additional three spaces each.

Successively lower levels (higher value characters and more deeply indented) represent subordinate processes. As will be seen in the general system structure the highest most levels are controled by increments of time; years, quarters, months, days, etc. while lower levels are controlled by events or conditions.

CONTROL NUMBERS: The control numbers used in GOALS are developed by alternating the use of numbers and letters to represent sucessively lower levels within the system. The system is similiar to English outlining except that only capital letters and numeric characters are used. For a given statement there is nothing to indicate its position in the hierarchy unless the entire control number is dipicted or the starting control number on the page is given. When GOALS statements are machine stored the entire control number is either stored or is assumed.

Principles

An Information system is distinguished from operating systems, command interperters, compilers and the like. An Information System is that set of communications, operations, files and outputs associated with a single conceptual "file."

I am not talking about a single program. Historically I am talking more about an application area.

Elements

Components

First an analogy: All purely mecanical devices are made up of elemental components; the incline plane, the wheel and axle, the lever and the chamber. The physics of these basic components and the materials from which they are constructed define the limits of their application. You may be saying, that list does not sound correct or "what about the screw." In listing elemental components certain definitions are inherent. I define the screw as a "rolled incline plane."

For information systems I assert that the list is: Communications, files, operations and outputs. The limits for such systems are defined by the ordering of the elements using the primitive structures (sequence, alternation and repetition).

As a note; to date the list of elemental components may have been input, process and output without regard to structure. This is more elemental considering all computer processes but is unbounded. This makes a general system design technique very difficult. Adding hierarchy to the above does not enhance these primitives to any great extent.

Relationships

With these boundries and definitions in hand, lets look at the relationships that develop.

There is generally a one to one relationship between file structure and operations structure, between communications structure and operations structure, between output structure and operations structure. In other words the operations or control structure mimics the other components of the system and each componet is related to the other in structure. Structure begins with the file structure.

Example; if you have a file of accounts and you want

to report them; the report program may need to be structured exactly the same as the file or database to report all the data in the file. Most often there is a one to one relationship between files and outputs. In the report example the report structure could be expected to look exactly like the file. If the report is to look different than the file there would be in intervening operation usually a sort or selection to convert an intermediate output to the final output.

The same is true of communications which on the data processing level are the transmissions to the uses, the screens and the messages. The structure of a communication is generally the same as the operation structure which is the same as the data structure and thus the communication structure is the same as the data structure. This substantiates the theory that systems can be completly described knowing only the data structure. True but limited. Knowing the structure of any part should in theory give you the whole.

If everything describable about a system can be described in simple structures (and thus in GOALS) and the components of a system include only communications, files, operations, and outputs and GOALS can be used in all system phases then we have a framework for a general system covering conception through maintainence.

Lets look at any application. Traditionally you would begin with a requirements statement and do an analysis of the existing system. Forget flowcharts, classic narratives, and other charting techniques. Think of progressively decomposing the system using simple english outlining starting with the functions. Functions fit into the operations structure discussed. You will note that as you get down a level or two you will encounter repetitive tasks dependant on conditions, add WHILE and IF to your outlines and keep describing. Remember that users can understand outlines and repetition and alternation are not difficult to understand.

Operations will include existing machine processes, manual proceedures, paper flows, sorting processes etc. As you are going through the operations keep a list of the files that are mentioned and note the file keys (and sorts) and any advantages or requests for multiple keys.

List any outputs or reports prepared by the organization or required in the future.

Communications will be minimal at this stage but note any memos that may go from one section to another of a "file" of notes used as crossreference or duplicate of any more perminent file.

Your documentation is now shaping up; your notebook and I assume that the whole world has change to 8½"S11", should be divided into communications, file, operations and outputs.

The starting point for design is the detailing of the files in your file list. You will want to reduce the files as much as possible to a single file. By way of naming conventions the "file" should have the same name as the system at hand.

You will notice that many of the manual files are really communications in that they are "views" of the file that are required in a particular subfunction.

The design of the conceptual file must be validated against the required operations. I am going to leave this hanging for a moment to discuss a General System Structure.

General System Structure

A General System Structure is presented on the following page in Goals.

This structure is not applicable in all systems but is used as a pattern for system discription, design and understanding.

The key elements of design of this structure are:

1. File unity; a system with this structure has only one conceptual file. It may have any number of datasets of or physical files but they must be formalized into one.
2. Journalizing or logging; all changes made to data items can be (and normally should be) logged.
3. Last action dating; incorporated as part of logging, permits an offline log.

One detailed implication of this is need to have a date stamp in each detail set and a master date stamp in the master file.

Note: sleeper from the contributed library is a must. A standard job stream to prepare the system for shutdown and to bring it back up to production mode is also recommended. Allocation of application programs a desirable feature is the reason for this and also a good way to get sleeper going again.

General System Architecture

```
Begin system
WHILE NOT EOSystem
  WHILE NOT EOYear
     WHILE NOT EOQuarter
        WHILE NOT EOMonth
           WHILE NOT EODay
              WHILE ONLINE
                 Begin online
                 identify operator and security
```

```
                    Open system file
                    Open current files
                    WHILE  Communication
                       IF control transfer
                          transfer control
                       IF batch request
                          initiate request
                       IF update , add or delete
                          Begin
                          Memo to LOG
                          LOCK
                          Update ,add or delete
                          UNLOCK
                          End
                       IF inquiry
                          perform communication operation
                       .
                    !
                 !
              !
              End Online
              Begin daily batch
              Perform daily batch processing
              Run LOG analysis
              If end of week
                 Perform Monthly Processing
              .
              ROLL FILES
           !
           perform Monthly processing
        !
        Perform Quarterly processing
     !
     Perform end of year processing
!
Close system
End
```

A GENERAL DESIGN

With this Architecture and database design complete we have the basis for the development and implementation of any application.

Step 1 is inquiry into our file; if there is only one search criteria then we calculate into to file and return the master data or a summary. Once positioned in a master we can chain through our detail sets or follow appropriate programatic paths.

The master screen (a communication) should provide inquiry, update, and addition ability.

Each detail set should have a screen providing the same update add and inquiry ability. Our screens will be one for one with the detail sets. Think of a detail set as having a buffer that will correspond to communication (VPLUS) buffer. Moving data within one program is facilitated with this concept.

The list of detail sets becomes a list of programs which must be written to handle the retrieval, update, addition, deletion and editing of data for the detail set.

When this is complete you will have a functioning system; it will not function well. I have intentionally oversimplified. The office proceedures which may be in place or will evolve will dictate what combination of sets will appear on a screen but no effort was be lost in developing the barebones system according to this method. Each set (detail set) should have its own program to handle retrieval and update. When requirements demand inclusion the programs can usually be used with few changes. You can take this one step further to include a general scheme to handle multiple data sets on one screen.

The question then becomes; "How do I tie this all together?"

Interactivity and Control

Let's say that we have written a system composed of a series of programs that correspond to our data sets. The way in which we implement interactivity is through a control program called MENU. 4A Menus

A master data set will exist at the top of the conceptual file and the primary search path will be the file key. Other search paths will be provided through subsystems such as "Name Family" or through automatic masters. For all detail sets associated to the master there will be

a program to handle that data set. Your analysis will dictate all the processes that the operator may wish to perform.

As other requirements develop associating more than one data set the code can be combined and new screens developed.

The menu control program provides transfer of control. It can do this either "quietly" or "loud." Loud is the obvious implementation; the operator choses a data set from a menu screen, the control is transferred via a "call" to a dynamic subprogram the data set is accessed updated, etc. and control returns to the controlling menu. But let us give the operator the ability to "tell" the system where he wants to go next. If he does a common area flag can be set to say don't display the menu simply transfer control to some other subprogram. We call are common area for data SYSBLK and out flag(s) Q1, Q2, etc. (you are not limited to one level of menu).

A menu structure may look like this:

```
    MAIN MENU
WHILE NOT PARENT OR END OF SYSTEM
  IF LOUD
     GET MAIN MENU SCREEN
     SEND (SHOW) SCREEN
     WHILE  EDITS'FAILED
       EDIT FIELD
       IF EDITS FAIL
          SEND SCREEN
       .
     SET MODE TO QUIET
  IF QUIET
     IF NEXTPROCEEDURE=A
        CALL A
     IF NEXTPROCEEDURE=B
        CALL B
          ...
     IF NEXTPROCEEDURE =N
        CALL N
     ELSE
        CALL CONTROL'NUMBER'TABLE
     .
  !
```

Through this technique those programs which are not being used are not using memory resources. The CONTROL NUMBER TABLE refers to implementations which have levels of menus. If the control reference is not handled at that menu level control is appropriately passed to the proper level where a control program can handle it.

The quiet "CALL" technique can be used for any of the data set programs discussed by putting the quiet call structure "around" the program and requiring the passing of appropriate data into or from the communication buffer. If you need to pass data from one subprogram to another and you want to release the calling program stack space you can do so with extra data segments (DMOVIN, DMOVOUT) or message files or scroll files (logical device dependant files) that you set up in the application program i.e. BUILD SCROL033;rec=-80,16,f,ascii.

Pitch for the use of intrinsics; we have found that most 3000 users do not take advantage of some of the very rich intrinsics in MPE. They are simple calls, well documented and even those that require bit settings are fairly easy to implement in any language.

The COMMAND intrinsic, for example lets you issue MPE command line, commands programatically. We use this to create stream jobs then kick off the job from online programs. A report menu can be used this way.

Effect of called programs on the Stack

The effect of using properly implemented called programs is simple and dramatic. You reduce the amount of stack (that normally translates into main memory) that is required by each user of an application program. Jim Kramer HP SE Saint Louis (Quad Editor Fame) calls it timesharing the stack.

Usually the outerblock program carves out the required amount of data area to be shared by all subprograms in the "system"; this would normally include a database area, a VPLUS area and an area for the system at hand. MPE then carves out some data area for Image and VPLUS. Using a simple menu concept as discussed, as each program is called it will require its own data area and thus addition stack on top of the common (Q relative) data area, when the program returns to the menu this stack space will be unused but as soon as the next program is called this same space will be used by that program for its space.

COBOL sections do not do the same thing. They create data areas for all declared data in the data division. Sectioning permits smaller code segmentation but this is a shared resource on the 3000 anyway. Note that with stack sharing per user that the reduction in memory requirements is greatly enhanced over code optimization.

You will also find that editing code is much easier with smaller source files, that compilation is faster and more concise code is written.

SL's and USL's

SL's

- Modules, entry points and called Programs require 1 CST entries if they are not already referenced in a running process.
- Code is sharable by all programs. The PUB.SYS SL is avalable to all programs. Account and group SL's are available to programs being run out of that Account or group.
- You may exclusive access to the SL to make an entry in it.
- When SL entries are made you do not need to prepare the SL. It is available after you have exited the segmenter.

USL's

- Programs compiled into a USL must be prepared before they are runnable.
- Many programs may be compiled into the same USL. When a program is run the system will look to the USL for resolution of called programs, it then looks to the PUB.SYS SL unless a library is specified in the RUN. (RUN prog;LIB=G)

- All USL resolved entries create XCST entries except the outer block.

CST's and XCST's

- There are 192 CST entries available to user processes
- There are 1028 XCST entries available to user processes.

COMPILE INTO A USL COBOL/3000 Example

```
!JOB JOBNAME,username/userpass.accountname/accountpass;OUTCLASS=
!COBOL progname,$NEWPASS,$NULL
!SEGMENTER
USL $OLDPASS                    **  only needed  **
NEWSEG progname,progname'       **      for      **
PURGERBM SEGMENT,progname'      **  COBOL/3000   **
USL yourusl
PURGERBM SEGMENT,progname
AUXUSL $OLDPASS
COPY SEGMENT,progname
EXIT
!TELL user.acct; yourprog ---> yourusl
!EOJ
```

PREP OF USL

```
!JOB DyourUSL,user/userpass.account/accountpass;PRI=ES;OUTCLASS=
!PURGE yourrun
!CONTINUE
!BUILD yourrun;DISC=2500,1,1;CODE=PROG
!SEGMENTER
USL yourusl
PREPARE yourrun;MAXDATA=16000;CAP=MR,DS
EXIT
!TELL user.acct;  yourrun ---> yourrun
!EOJ
```

CALLABLES INTO SL's

```
!JOB D!SL,user/userpass.account/accountpass;OUTCLASS=,1
!COBOL yourprog,$OLDPASS,$NULL
!SEGMENTER
AUXUSL $OLDPASS
SL SL
ADDSL yourprog
EXIT
!TELL user.acct;  yourrun ---> yourrun
!EOJ
```

MENU

```
REPEAT until parent or end of system
   IF loud
      get menu screen
      show screen
      REPEAT until edits pass
         edit fields
         IF edit fail
             send screen
      !  .
      set mode to quiet
```

```
.
IF quiet
   IF nextprocedure = "O"
       CALL "O" USING ., ., .
   IF nextprocedure = "I"
       CALL "I" USING ., ., .
        .
        .
        .
   IF nextprocedure = "n"
       CALL "n" using ., ., .
   ELSE
       CALL "CONTROLNUMBERTABLE" using nextprocedure
   .
!  .
```

Goals-SPL Standards

Section	Title
1	General
2	Procedures and Declarations
3	Moves
4	IF Control
5	REPEAT Control
6	Witan include files
7	Coding rules

GOALS-SPL STANDARDS

General

Indentation of three spaces indicates the beginning of a new level. If the next line is indented six spaces it indicates a continuation of the previous line.

Assignment is done with the ":=."

Comparison is done with the "=".

The astrisk is used to indicate that the address required in a statement has already been loaded on the stack. This has general applicability but we will limit its use to moves where the previous move has used the stack decrement option leaving the ending address on TOS. In a MOVE WHILE there is a stack decrement feature, a ",1" following the A, AN or N indicates that the final destination address is left on TOS.

The asterisk in parenthesis (*) indicates a backreference to another data item causing a redefinition of the area in the data stack. This back reference does allocate one word of the stack as a pointer.

Parameters should always be on word boundries thus BYTE ARRAYS should not be used as parameters.

Procedures and Declarations

Procedures parameters should all be called by reference not by value.

The form for an outer block program is:

```
$CONTROL USLINIT [ ERRORS=5, LIST, ... ]
BEGIN   << SOURCE >>
  [global data declarations]
  [procedures/intrinsics]
  [global-subroutines]
  [main-body]
END. <<  SOURCE  >>
```

The form for a subprogram is:

```
$CONTROL SUBPROGRAM [ERRORS=5, LIST, ... ]
BEGIN  << SOURCE >>
  [compile time constructs]
  [procedures/intrinsics]
END.   << SOURCE >>
```

The form of a sample subprogram using the Witan INCLUDE files found in the appendix follows:

```
$CONTROL SUBPROGRAM,ERRORS=5,NOLIST,NOWARN,SEGMENT=SEGNAM
BEGIN << SOURCE >>
$INCLUDE INC1G.T

<< BEGIN EXTERNAL PROCEDURE DECLARATIONS >>
$INCLUDE STDINTR.T << STANDARD EXTERNAL PROCEDURE DECLARATIONS >
PROCEDURE BLANK(WINDOW,VI);
   VALUE VI;
   IA WINDOW;
   IN VI;
   OPTION EXTERNAL;
<< END EXTERNAL PROCEDURE DECLARATIONS >>

PROCEDURE SEGNAM(VBLK,SYSBLK,RTN'CDE);
   IA VBLK,SYSBLK;
   IN RTN'CDE;
BEGIN << SEGNAM >>

<< BEGIN DATA >>
$INCLUDE VBLK.T
$INCLUDE SYSBLK.T
IA IBLK(0:0);
$INCLUDE SUBGLOB.T     << USING SUBGLOB.T REQUIRES THAT VBLK,IBLK
                          SYSBLK HAVE BEEN INCLUDED IN THIS PROCE
                          EITHER AS PASSED PARAMETERS OR AS NULL
                          ARRAYS. >>
<< OTHER DATA LOCAL TO PROCEDURE >>
LG KEEP'GOIN;
IN VI;
IN MISC;
IA (0:9)TEN'WORDS;
<< END DATA >>

<< BEGIN SUBROUTINES >>
SUBROUTINE PUT'WINDOW;
   BEGIN << PUT'WINDOW >>
      V'PUT'PAUSE(VBLK,2);
      BLANK(WINDOW,30);
      WINDOW'LEN:=60;
      VPUTWINDOW(VBLK,WINDOW'LINE,WINDOW'LEN);
      VSHOWFORM(VBLK);
   END; << PUT'WINDOW >>
<< END SUBROUTINES >>

<<**********************************>>
BEGIN  << CODE >>

KEEP'GOIN:=TRUE;
WHILE        KEEP'GOIN DB
   KEEP'GOIN:=FALSE;
END'REP;

END; << CODE >>
END; << SEGNAM >>
END; << SOURCE >>
.
```

Moves

General Forms:

MOVE destination:=source, (length)[,stack decrement];

Literals:

Length need not be specified in the move of a literal If successive moves are anticipated to build a string or concatenate into a buffer then the stack decrement option of 2 can be used. Example:

```
MOVE OUTBUF:= "Hello",2
MOVE          *;=" Everyone";
```

Non-Literals:

SPL requires type compatibility in moves, therefore general buffers should be defined in words and in bytes. The word buffer name should end with "'W." The byte buffers will have the just name without an identifying sufix.

The length parameter in the move should specify a name equated to the length in bytes or words depending on the type of move. The equate will generally be generated by DBUF. Byte lengths will begin with "BL' ", word lengths with "WL'."

Example:

```
MOVE OUTBUF:=
   ACCOUNTNO,(BL'ACCOUNTNO);
```

Some moves may embed procedure to insure type compatibility and at the same time perform the appropriate conversion.

IF Control

The control structure for the IF will follow directly the structure enforced in GOALS. All IF's will be followed by a condition which may be compound and may extend to subsequent lines (note; continuation line disciple in general standards).

Following an "IF" condition a TB will be inserted, which is defined as a "THEN BEGIN." SPL does not require a BEGIN if the following statements are not compound, i.e., a lone statement. However, the "BEGIN" is required to bracket the sequence and to enforce the use of an "END" on the same level as the beginning "IF." If there are subsequent "IF's" on the same level (mutually exclusive IF's — programmer enforced) the IF should be converted to an IF'G which is defined in INC1G as an "END ELSE IF." This is not called a "IF" in GOALS. It is refered to as an "IF string" (mutually exclusive conditions).

Nested IF's:

If "IF's" are nested, the nested IF may begin any time after the "TB" of the preceding IF and will be indented to show its nesting. The rules for the nested IF are exactly the same as the IF; TB required.

ELSE

When the trailing ELSE is required in an IF string, the preceding end for the IF must not have a semicolon. The ELSE requires a BEGIN-END pair to enforce the terminating "END" at the end of the IF string.

Nested IF strings, where trailing elses come together may cause some confusion, but do not require any special rules.

Example:

```
IF --condition--          TB   << THEN BEGIN >>
  IF --condition--        TB
     --statm't--;
     --statm't--;
  IF'G --condition--        TB
     --statm't--;
     --statm't--;
  ELSE'G
     --statm't--
  END'IF;
ELSE'G
  --statm't--;
  --statm't--;
END'IF;
```

Repeat

General Form:

```
WHILE          --condition--          DB
   --statm't--------)
   --statm't---------
END'REP;
```

The REPEAT in the GOALS-SPL is used as documentation and is defined as a null statment. REPEAT must be followed by WHILE and a condition or compound condition. Following a WHILE condition a "DB" is required which is DEFINED in INC1G as a "DO BEGIN." As in the IF construct a "BEGIN" is required to enforce a terminating "END'REP."

```
BYTE POINTER
   BP  << USED FOR TEMPORARY POINTER, NOT SAVED >>
  ;
EQUATE
   RTN = 13  << CARRIAGE RETURN IN ASCCI >>
  ,ESC = 27  << ESCAPE CHARACTER IN ASCII >>
  ;
INTEGER I,J,K,LEN80,OLD'LANGUAGE;

  DA IBLK'D          (*) =  IBLK;
  BA IBLK'B          (*) =  IBLK;

  DA SYSBLK'D        (*) =  SYSBLK;
  BA SYSBLK'B        (*) =  SYSBLK;

  DA VBLK'D          (*) =  VBLK;
  BA VBLK'B          (*) =  VBLK;
DEFINE
   EL = END ELSE#
  ,END'IF = END#
  ,END'REP = END#
  ;
```

INC1G.T

This INCLUDE is used for abbreviation of data types and some constructs for GOALS presentation SPL compilations.

```
DEFINE <<USED TO ABBREVIATE DATA TYPES>>
   IA = INTEGER ARRAY#
  ,IN = INTEGER#
  ,DI = DOUBLE #
  ,LA = LOGICAL ARRAY#
  ,DA = DOUBLE  ARRAY#
  ,BA = BYTE    ARRAY#
  ,RA = REAL    ARRAY#
  ,XA = LONG    ARRAY#
  ,LP = LOGICAL PROCEDURE#
  ,DB = DO BEGIN#
  ,TB = THEN BEGIN#
  ,LG = LOGICAL#
  ,REPEAT = #
  ,G'IF = END ELSE IF#
  ,G'ELSE = END ELSE BEGIN#
  ,IF'G = END ELSE IF#
  ,ELSE'G = END ELSE BEGIN#
  ;
```

IBLK.T

```
<< IA IBLK(0:42); >>
  DEFINE
     COND'WORD        = IBLK #,
     STAT2            = IBLK(1) #,
     STAT3'4          = IBLK'D(1) #,
     STAT5'6          = IBLK'D(2) #,
     STAT7'8          = IBLK'D(3) #,
     STAT9'10         = IBLK'D(4) #,
     BASE             = IBLK(10) #,
     MODE1            = IBLK(26) #,
     MODE2            = IBLK(27) #,
     MODE3            = IBLK(28) #,
     MODE4            = IBLK(29) #,
     MODE5            = IBLK(30) #,
     MODE6            = IBLK(31) #,
     MODE7            = IBLK(32) #,
     MODE8            = IBLK(33) #,
     ALL'ITEMS        = IBLK(34) #,
     PREV'LIST        = IBLK(35) #,
     NULL'LIST        = IBLK(36) #,
     DUM'ARG          = IBLK(37) #,
     NUM'BASE         = IBLK(38) #,
     IBLK'LEN         = 43 #
     ;
```

IBLKG.T

The following is initilization code to be included in the outer block program to set IBLK fields :

```
MODE1              := 1;
MODE2              := 2;
MODE3              := 3;
MODE4              := 4;
MODE5              := 5;
MODE6              := 6;
MODE7              := 7;
MOVE ALL'ITEMS     := "@;";
MOVE PREV'LIST     := "*;";
MOVE NULL'LIST     := "0;";
DUM'ARG            := 0;
```

VBLK.T

```
<<   THIS ASSUMES THAT VBLK IS DECLARED IA VBLK(0:51)          >
<<     VBLK IS MADE UP OF COMAREA AND THE OLD VBLK             >
<<  CALLS TO VIEW INTRINSICS WILL USE VBLK AS THE COMAREA PARM >

   <<SPL DECLARATIONS FOR COMAREA>>
   DEFINE
      COM'STATUS          = VBLK (0)   #,
      COM'LANGUAGE        = VBLK (1)   #,
      COM'COMAREALEN      = VBLK (2)   #,
      COM'USRBUFLEN       = VBLK (3)   #,
      COM'CMODE           = VBLK (4)   #,
      COM'LASTKEY         = VBLK (5)   #,
      COM'NUMERRS         = VBLK (6)   #,
      COM'WINDOWENH       = VBLK (7)   #,
      COM'LABELSOK        = VBLK (9)   #,
      COM'CFNAME          = VBLK'B (10*2)  #,
      COM'NFNAME          = VBLK'B (18*2)  #,
      COM'REPEATAPP       = VBLK (26) #,
      COM'REPEATOPT       = VBLK (26) #,
      COM'FREEZAPP        = VBLK (27) #,
      COM'CFNUMLINES      = VBLK (28) #,
      COM'DBUFLEN         = VBLK (29) #,
      COM'DELETEFLAG      = VBLK (32) #,
      COM'SHOWCONTROL     = VBLK (33) #,
      COM'PRINTFILNUM     = VBLK (35) #,
      COM'FILERRNUM       = VBLK (36) #,
      COM'ERRFILNUM       = VBLK (37) #,
      COM'FM'STORE'SIZE   = VBLK (39) #,
      COM'NUMRECS         = VBLK'D (21)    #,
      COM'RECNUM          = VBLK'D (22)    #,
      COM'TERMFILENUM     = VBLK (48) #,
      COM'TERMMODE        = VBLK (49) #,
      COM'TERMALLOC       = VBLK (50) #,
      COM'DATAOVERRUN     = VBLK (51) #,
      COM'READTIMEOUT     = VBLK (52) #,
      COM'OTHERDATAERR    = VBLK (53) #,
      COM'MAXRETRIES      = VBLK (54) #,
      COM'TERMCONTROLOPT= VBLK (55) #,
      COM'TERMOPTIONS     = VBLK (55) #,
      COM'ENVINFO         = VBLK (56) #,
      COM'TIMEOUT         = VBLK (57) #
   ;
   EQUATE
      COMAREALEN          = 60,
      COBOL'LANG          = 0,
      VBLKLEN             = 100,
      SPL'LANG            = 3,
      MAXWINDOWLEN        = 150,
      MAXMODELEN          = 8,
      NAMELEN             = 15,
      NORM                = 0,
      NOREPEAT            = 0,
```

```
      V'REPEAT          = 1,
      REPEATAPP         = 2,
      ENTERKEY          = 0,
      PARENTKEY         = 1,
      KEY2              = 2,
      KEY3              = 3,
      REFRESH           = 4,
      PREV              = 5,
      NEXTKEY           = 6,
      INQ'ENT           = 7,
      EXITKEY           = 8
   ;
   <<SPL DEFINITIONS FOR VBLK>>
   DEFINE
      WINDOW'LEN        = VBLK (COMAREALEN+0)     #,
      MODE'LEN          = VBLK (COMAREALEN+1)     #,
      WINDOW'LINE       = VBLK (COMAREALEN+2)     #,
      WINDOW'MODE       = VBLK (COMAREALEN+2)     #,
      WINDOW            = VBLK (COMAREALEN+MAXMODELEN)     #,
      WINDOW'LINE'B     = VBLK'B ((COMAREALEN+2)*2) #,
      WINDOW'MODE'B     = VBLK'B ((COMAREALEN+2)*2) #,
      WINDOW'B          = VBLK'B ((COMAREALEN+MAXMODELEN)*2) #
   ;
```

SYSBLK.T

```
<<IA SYSBLK(0:114)  SPACE ALLOCATED IN MAIN PROGRAM >>
DEFINE
    CNTRL'NUM       = SYSBLK #,
    LST'PROC        = SYSBLK(2) #,
    NXT'PROC        = SYSBLK(4) #,
    Q1              = SYSBLK(6) #,
    Q2              = SYSBLK(7) #,
    Q'NEXT          = SYSBLK(8) #,
    OPER'ID         = SYSBLK(9) #,
    SECU'TY         = SYSBLK(11) #,
    SSC             = SYSBLK(13) #,
    CNUM            = SYSBLK(16) #,
    L'FLNUM         = SYSBLK(21) #,
    M'FLNUM         = SYSBLK(22) #,
    FLAGS           = SYSBLK(23) #,
    DQSTAT'SB       = SYSBLK(28) #,
    GLSTAT'SB       = SYSBLK(43) #,
    TERMID          = SYSBLK(58) #,
    MSBLK'SB        = SYSBLK(63)#  << STARTING ON DOUBLE BOUNDRY
    ;
DEFINE
    CNTRL'NUM'B     = SYSBLK'B #,
    LST'PROC'B      = SYSBLK'B(2*2) #,
    NXT'PROC'B      = SYSBLK'B(2*4) #,
    OPER'ID'B       = SYSBLK'B(2*9) #,
    SECU'TY'B       = SYSBLK'B(2*11) #,
    SSC'B           = SYSBLK'B(2*13) #,
    CNUM'B          = SYSBLK'B(2*16) #,
    FLAGS'B         = SYSBLK'B(2*23) #
    ;
EQUATE
    CNTRL'NUM'BL    = 4,
    LST'PROC'BL     = 4,
    NXT'PROC'BL     = 4,
    OPER'ID'BL      = 4,
    SECU'TY'BL      = 4,
    SSC'BL          = 6,
```

```
CNUM'BL          = 10,
FLAGS'BL         = 10
;
```

Coding Rules

All agorithms should first be done in GOALS without concern for the SPL structure. SPL constructs will be used for individual statements and conditions but the control structure should be in GOALS.

This complete:

1. Replace all ELSE's with G'ELSEs or ELSE'Gs.
2. Locate all "IF's that are on the same level as a "running" IF. Replace each running IF with an IF'G or G'IF.
3. Replace all "."'s with an END'IF;
4. Insert a THEN BEGIN or "TB" following every IF condition.
5. Replace all "!" with an END'REP;.
6. Insert a "DB" or DO BEGIN after every REPEAT condition.

An Example using the rules on the preceeding page

```
WHILE          ------------
   IF ------------
      --------------
      --------------
      IF ------------
         --------------
      ELSE
         --------------
         --------------
      .
   IF ------------
      --------------
   .
   IF ------------
      --------------
   IF ------------
      --------------
   ELSE
      --------------
   .
!
```

```
                                          <<   SPL      RULE >>
WHILE          ------------          DB   << DO BEGIN     6  >>
   IF ------------                   TB   << THEN BEGIN   4  >>
      --------------
      --------------
      IF ------------                 TB   << THEN BEGIN   4  >>
         --------------
      ELSE'G                               << END ELSE BEGIN  1 >>
         --------------
         --------------
      END'IF;                              << END'IF        2  >>
 ***(20) ERROR ***
LINE
1490
 TRUNCATED BY  4 CHARACTER(S)
      IF'G ---------                   TB   << END ELSE 2 >>  << THEN B
         --------------
      END'IF;                              << END'IF        3  >>
      IF ------------                 TB   << THEN BEGIN   4  >>
         --------------
 ***(20) ERROR ***
LINE
1495
 TRUNCATED BY  4 CHARACTER(S)
```

```
   IF'G ---------                TB   << END ELSE 2 >>  << THEN B
     --------------
   ELSE'G                             << END ELSE BEGIN  1 >>
     --------------
   END'IF;                            << END'IF      3  >>
END'REP;                              << END'REP     5  >>
```

Note: work the top example yourself using the rules and see if it matches the completed program. Note the count of the begins and ends match for SPL. Do the algorithm correctly in GOALS and the SPL code will follow.