STANFORD SYNCHROTRON RADIATION LABORATORY

ENGINEERING NOTE

CODE	SERIAL	PAGE
1550	M227	1 OF 32

AUTHOR: HOWRY/WILLIAMSON | DEPARTMENT: SSRL | LOCATION: | DATE: 5-4-94

PROGRAM – PROJECT – JOB: SSRL - STORAGE RING; SPEAR ELECTRICAL SYSTEM

COMPUTER CONTROL SYSTEMS

TITLE: SPEAR REAL TIME DATABASE SOFTWARE PACKAGE

7600-54250

RL-3220 (Rev. 5/75)

CODE 1550
5-4-94

SPEAR Real Time Database Software Package

S. Howry and A.Williamson

§ *Table of Contents*

1. Introduction

The package permits the system or applications programmer to:

- Create, install, and modify a database of any number of signals.
- Display, control and alter selected attributes of any subcollection of signals, in realtime, directly from a terminal.
- Write application software that can acquire, control and alter specific sets of attributes of subcollection(s) of signals; and to program sequences of such operations.
- Write applications software processes that can run concurrently under vax/vms and execute any or all of the above features.

This realtime database package creates an abstract acquistion/control environment that is fully independent of the nature of the hardware i/o interface actually installed. It is written for a DEC vax/vms operating system, in Fortran77. It is well suited to a local area network(LAN) for example, because the dynamic portion of the database can be broadcast as a single block by the one node that owns the hardware interface to the data. In this one node, all the data is continuously acquired by a process of this package and placed into this block. An application process in any node accesses only the constantly refreshed copy of this block in its node to acquire monitored values. Furthermore, since all processes interact with the database in this way, they can be moved from one node to another to distribute the load.

The sections of this document have enough information for the in house software developer to write applications software, and to make extensions to this RTDB package.

2. Functional Overview

The starting point is CALSIG, the database design file, sometimes called the database schema. This file contains all the functional concepts, enumerated in a tree-structured fashion. The ends, or 'leafs' of the tree are called signals, and represent either actual hardware signals, or an array of data generated by software. Each such end is assigned one database record. In this way functional concepts are associated with collections of database records. For example, a set of records associated with the functional concept 'trim supplies' could be specified as:

- T = trim supplies
- Each T has 2 orientations, horz,vert (TQ)
- Each TQ has:
- Each TQ has 12 supplies (TQC)
- Each TQC has:
    - class AM1 readback
    - class AC1 setpoint
    - class DI1 status
    - class DO1 on/off control
    - class XX1 strength to current calibration polynomial;(generated by a lab calibration process).

So the readback of the fourth horizontal trim has the unique signal name TQ1C4/AM1; the subcollection TQ1/AC1 represents setpoints of all of the horizontal trim supplies, and the entire set of 2*12*1=24 setpoints has the generic form TQC/AC1. Each record in this database contains 256 bytes. A header takes up the first 40 bytes, and the remaining 216 bytes are for 'attribute fields', whose 2 letter names, sizes, and locations are specified at the very end of the schema file. For hardware signals, attribute fields contain routing addresses,sizes,types, display names, pedestal constants,etc. For software signals, the record contains an array of generated data. These attribute records are discussed in more detail in section 3 below.

The CALSIG file is intended to serve both as a complete I and C signal documention (at least from the functional point of view), and also as input file for the database generation program DBGEN. For documentation reasons, there are many comment lines in CALSIG, which are ignored by the generation program. Process DBGEN expands the tree-structured description to build the internal database structures. It assigns to each signal a unique integer called a signal ID, which is used as a key to acquire data about the signal. The static part of the database makes up the global shared section DBTREE.GBL. The rest of the shared global section contains the 'dynamic', or continuously refreshed, portion of the database. It also contains the generated pointers that define the mapping between signal ID's and the device dependent(or at least network dependent) structure shared global section DBAS.GBL.

The program DBGEN fills the 'individual signal name', or SN attribute, of each record as it builds it. In the above example 'TQ1C4/AM1.' is the (unique) signal name for the fourth horz trim readback.

For the very first execution of DBGEN at an installation, all of the other attributes in each record are initialized to "E" bytes (hex 45) by program DBGEN. Often, especially in the early phases of a project, DBGEN is run in order to effect a change in the structure of the database (i.e.- an addition or deletion is made to the schema file CALSIG. In this case all of the attribute records are saved on tape before DBGEN starts its generation, and are copied back into corresponding records of the just-construcred structured database shell. Records corresponding to deleted signals are flagged as such at the terminal, and are discarded. Records corresponding to new signals are initialized to "E" bytes (hex 45).

Once the database structure is in place and some (but not necessarily all) attribute records are loaded with valid data the i/o process XCAMAC can be started. On startup, this process first calls function PBZ.FOR to generate the mapping pointers into and out of the dynamic arrays. It uses attribute data in the static arrays to do this. XCAMAC then executes a loop at about 64Hz(this rate depends in the vax/vms process swap time). Each time through this loop it does:

- finishes up the previous camac read/write channel operation
- initiates a new camac read/write channel operation

Database attribute record information can be represented in a text format of an editable vax/vms file. By convention, such files have an extension .ISD. These files can of course be created directly from terminals. They also are created by extracting any subset of attributes from any specified subset of database signals using process DBGISD. The format and examples are given in section 6 below. These files can be modified, searched, sorted, collated, etc., by any text editor or other program, and then re-inserted back into the 'static' database by using process DBFILL. DBFILL is the inverse operation of DBGISD. An audit trail is maintained (in file dbupdate.aud) is maintained to monitor data entered by DBFILL.

The process DBEXAM displays all of the attributes of a specified single record, and allows the developer at a terminal to alter them. This program is typically used to edit a small set of signals. Larger subcollections are more conveniently handled with .ISD files and programs DBGISD and DBFILL. The audit trail (in file dbupdate.aud) is updated if DBEXAM has changed data in the database.

All applicatons processes (such as DBGISD,DBEXAM as described above) will be called AP's in the discussions below. AP's interact with the database through a small set of subroutines that are linked from a common library into the executable image of each user process. Each user process can get static data by invoking subroutines such as COLLECT, which finds the information in the associated attribute records. In a similar way, each user process can get dynamic data through subroutines such as READDB, which just reads the common global section that is continuously refreshed by co-executing process XCAMAC. Each user process can output digital/analog data through SETDB and SETDB1. These put specified values into the dynamic arrays and then tell co-executing process XCAMAC to transmit the appropriate mapped data.

A full description of each subroutine callable from an AP is given in section 7 below. In general, AP's will call subroutine GETID only on starting up to get lists of database signal ID's, and then use these lists in subsequent calls to subroutines such as READDB, READSU, COLLECT, SETDB1, etc., to interact with the database. Processes DBEXAM, and DBGISD are examples of AP's.

3. Database Signal Classes, Names, ID's, and Attributes

3.1 Definition of Signal Class and Signal Name

One major purpose of this database is to allow programs to refer to signals symbolically. For hardware signals, this means that signal groups can be easily defined; and code can be developed independent from consideration of network or i/o formats. For software signals, structures of arrays normally packed into fortran COMMON blocks can instead be defined as part of a system wide data structure. Congruent copies of any portion of this structure can be naturally declared in the database schema file.

Each individual signal belongs to one of the following classes:

- DM single-bit input "digital monitor"
- AM up to 16-bit ADC input "analog monitor(readbk)"
- DC single-bit output "digital control"
- AC up to 16-bit rampable DAC output "analog control(setpt)"
- DV real*4 adc thru HP digital voltmeter "dvm"
- DI up to 16-bit input "digital input"
- DO up to 16-bit output "digital output"
- XX 216 byte array of derived data "software signal"

Each such signal is given an alphanumeric Signal Name, and a numeric Signal ID. The signal Name for any single signal has the following structure:

$$< fcnletter >< node > [< subletter >< number >]/ < class >< number >$$

The slash (/) must always be present. $< fcnletter >$ is a single letter denoting function of the signals to be described, for example,

$T = trims,$
$V = vacuum,$
$P = protection,$
$R = rf$

$< node >$ is the number of the Local Area Network node where the signal is read. $< subletter >$ is a single letter. Each $< number >$ is a decimal integer. $< class >$ is one of the two-letter signal class names defined above. A signal name thus always consists of alternating letters and numbers. There can be at most 7 subletters before the slash. The letters alone, with the numbers omitted, define what is known as the full generic name for the specified function and class. Values for the letters and limits on the numbers are defined by the CALSIG file, which is described in the next section below.

The display names are independently assigned, and reflect, for example, magnet name, power supply location, device type, or other information familiar to the people in accelerator operations. Usually, all signals belonging to a given unit (magnet, power supply, etc.) will be given the same display name.

3.2 Specification of Subcollections Using Generic Forms

Collections of database signals can be represented in a compact notation by a text string of 'generic forms', separated by commas and terminated by a period. A generic form is a signal name with some or all of the numbers (not letters) either omitted or written like the following:

- (1:6) colon means 1 through 6
- (2;5;8) semicolon means 2,5, and 8 only
- (1:6;8) means 1,2,3,4,5,6,8
- (6:1) means 6,5,4,3,2,1

Entirely omitting a number means to use all possible values for that number. For example, if there is an excerpt

- R = radiofrequency system
- Each R has 4 stations (RS)
- Each RS has:
    - class DI
- Each RS has 2 cavities (RSC)
- Each RSC has:
    - class DM

- class DM
- class DM
- class DM
- class AM

in a CALSIG file for a hypothetical database, then the string

- RS(2;4)/DI, RSC/DM(1:3)

denotes $2*1+4*2*3=26$ signals, namely the DI for stations 2 and 4, and the first three DM's for each cavity at each station in the rf system.

3.3 Signal ID's and the associated database records

The Signal ID numbers are derived integers, assigned sequentially as follows: Signals for each node are first grouped together, and then within each node, the signals are grouped by class. Within that grouping, the signals are sorted by the order of their appearence in CALSIG, the schema file. The record of a signal is located by using the Signal ID as a (biased) subscript.

3.4 Signal Attributes and the fields of the database records

3.4.1 Attribute records for Hardware Signals

Each hardware signal belongs to one of the following classes:

- **DM** single bit input digital monitor
- **AM** up to 16 bit ADC input analog monitor(readbk)
- **DC** single bit output digital control
- **AC** up to 16 bit rampable DAC output analog control(setpt)
- **DV** real*4 adc thru HP digital voltmeter dvm
- **DI** up to 16 bit input digital input
- **DO** up to 16 bit output digital output

Each signal in the database has a 128-word (256-byte) record in the shared global section named DBTREE.GBL of paged memory. The record for a given signal is located essentially by using the Signal ID as a subscript. A description of the attributes, namely the information stored in the records is given below.

Each hardware signal in the database has its associated record split into attribute fields. The size and location of each field is specified at the very end of the schema file CALSIG. The meaning of each field is given here:

Camac Routing information INTEGER*2 fields (each of these is 2 bytes):

- **CO** Local Area Network node number (filled only by DBGEN)

- **BR** camac bramch number
- **CR** camac crate number
- **MN** camac module number
- **SA** camac subaddress
- **FC** camac function code

- **MT** module type. The possible module types are enumerated in file [spear.doc]mt.doc, and are listed below:

  - MT = 500 SAM(Smart Analog Multiplexor) for /AM'S
  - MT = 510 P.S.C. controllers for /AC'S
  - MT = 511 special for qddc (removed sept 1984)
  - MT = 520 TRANSIAC's trim supply cntllrs for /AC'S
  - MT = 5 type 208-605 pep style /AC modules for rf
  - MT = 8 type ???-??? pep style /AM modules for rf
  - MT = 609 pep style simple /DM,/DI modules for rf
  - MT = 540 IDIM(Isolated Dig Input Module) for TC/DM's

  - MT = 599 TRANSIAC model 2008 transient digitizer
  - MT =??? Model 145−??? LDDU (Longpulse Digital Delay Unit)

  - MT = 500 type 123-603 SAM(smart analog mx) for /am's
  - MT = 510 type 135-563 P.S.C. controllers for /ac's
  - MT = 520 TRANSIAC model 3016 trim supply, for /ac's
  - MT = 540 type 135-562 IDIM(isolated dig input) /dm's
  - MT = 550 type ???-??? IDOM(isolated dig output)
  - MT = 560 type 233-313 R10T (single-pole 10throw) sw

  - MT = type 123-669 attenuation, 10db, for bpms
  - MT = type 123-872 attenuation, 1db, for bpms
  - < *noaddr* > type 445-064 bpm detector

  - MT = 3388 KINETIC SYSTEMS gpib interface, for dvm
  - MT = 3530 KINETIC SYSTEMS relay mux, for old dvm
  - < *noaddr* > type 445-316 ext trigger for dvm
  - < *noaddr* > JOERGER model OR, output reg, for old dvm?

  - MT = JOERGER model SMC-L, motor cntlr, for slits
  - MT = type ?? 'flip coil input' module

  - MT = 2551 LECROY scalors, for lumnosity
  - MT = 2132 LECROY 'HV(4032a)-TO-CAMAC' for lumnosity
  - MT = 2008 TRANSIAC transient digitizer

- MT = 601 type 208-601 pep-style/di, for vacuum etc
- MT = 603 type 208-603 pep-style/di/do, for RGA etc
- MT = 604 type 208-604 pep-style/di/do, for RGA
- MT = 605 type 208-605 pep style/ac, for rf
- MT = 609 type 208-609 pep-style/dm/di, for rf

- $< noaddr >$ type 208-632 vcc branch simulator
- $< noaddr >$ TRANSIAC type 446-626 vcc branch driverII
- $< noaddr >$ type 135-337 camac branch receiver
- $< noaddr >$ type 135-315 camac crate controller
- $< noaddr >$ type 135-279 camac crate cntlr, for psc's ??
- $< noaddr >$ type 123-589 camac crate verifier

- **BN** bit number within the bit field defined by the device AD that contains the lsb(least significant bit) of the dynamic data of this signal.

- **FL** field width = number of bits in the dynamic data of this signal.

REAL*4 fields (each of these is 4bytes):

- **AK** linear term of the conversion between dac-adc integers and $< scientific - units >$ (AC, AM only)

- **OF** offset term of the conversion between dac-adc integers and $< scientific - units >$ (AC, AM only) note: the conversion formula is:
  $< scientific - units >=< dac/adcinteger > *AK + OF$

- **MI** minimum value in $< sci - units >$ (AC only)

- **MA** maximum value in $< sci - units >$ (AC only)

- **CK** constant (used on in vacuum related adc's, such as VG)

- **TO** tolerance value in $< sci - units >$ (AM only)

- **NF** 0. = in = active; 1. = out = not active (AC only). This is used to 'bugger out' a rampable control.

Ascii fields:

- **SN** (28 bytes) signal name (filled by DBGEN only) left-adjusted; terminated by period and trailing blanks

- **DN** (12 bytes) operational display name by AKPS (must not contain a comma)

- **SU** (12 bytes) text of $< sci - units >$ (e.g., AMPS for certain AC, AM signals; FAULT/ NOFAULT for certain DM, DC signals)

- **DP** (40 bytes) descriptive phrase (copied by DBGEN from CALSIG)

- **PL** (40 bytes) physical location (or other use, if desired by user)

- **AN** (12 bytes) auxiliary field, for unspecified use

Time field: This field is used to 'time stamp' the most recent change to any field in the record of the given signal.

- **TM** (8 bytes) vax/vms standard 64-bit binary integer 'quadword' giving the number of 100-nanosecond units elapsed since 0000 hours 17 Nov 1858. (see run time library routines SYS$GETTIM, SYS$ASCTIM)

Other INTEGER*2 fields (each of these is 2 bytes):

- **SC** signal class: 1=DM, 2=AM, 3=DC, 4=AC, 5=not used, 6=DI, 7=DO, 8=XX. (Filled only by DBGEN.)

- **RB** This is defined for class AC (setpoint) signals only; it is the ID of the class AM (readback) signal that corresponds to this setpoint signal. The correspondence is made by process DBGEN at database_build time, and is defined to be the signal with the same signalname (SN) as the setpoint signal except the text characters '/AC' are replaced with '/AM'.

3.4.2 Attribute records for Software Signals

Each software signal belongs to class**XX.**

A typical software, or class **XX** signal in the database has its associated record split into 3 attribute fields that make up a 40 byte header; and a single 216 byte body field which contains data produced and used by applications processes. The fields are:

Header fields:

- **SN** (28 bytes) signal name (filled by DBGEN only); this ASCII field is left-adjusted; terminated by period and trailing blanks

- **TM** (8 bytes) VAX/VMS standard 64-bit binary integer 'quadword'giving the number of 100-nanosecond units elapsed since 0000 hours 17 Nov 1858. (see run time library routines SYS$GETTIM, SYS$ASCTIM)

- **AT** (2 bytes) data type; this describes the data type of the body field. The types are described by 2 bit subfields within this word as follows:

XR*real* $*$ 4	00
XI*integer* $*$ 2	01
XD*integer* $*$ 4	10
XA*ascii* $*$ 2	11

The body field can be broken up into variable length subunits called 'arglists'. Arglists are described along with body fields below. Rarely it is be necessary to mix the data types among arglists. For example arglist 1 may be REAL*4, and ARGLIST 2 may be ascii*2, and this alternating pattern of arglists is continued indefinitely throughout the rest of the body field. In this case, the first 4 bits on the left(i.e.- the high order bits) of the **AT** word are 0011, and this 4bit pattern is repeated throughout the rest of the **AT** word. In this way the data type of each arglist is identified. In most cases, the body field is all real*4, which requires the **AT** word to be all 00 subfields, or = 0.

- **LK** (2 bytes) for future expansion, possibly permitting more general structures within a body field, or for an access lock mechanism.

Body field: The body field can be broken up into variable length subunits called 'arglists'. Arglists are specified as to data type by the **AT** attribute.

For type **XR** arglists, the format is:

$$ARGLXRj(0)2.*n$$
$$(1)real*4$$
$$(2)real*4$$
$$\ldots\ldots$$
$$(n)real*4$$

For type **XD** arglists, the format is:

$$ARGLXDj(0)2*n$$
$$(1)integer*4$$
$$(2)integer*4$$
$$\ldots\ldots$$
$$(n)integer*4$$

For type **XI** arglists, the format is:

$$ARGLXIj(0)n$$
$$(1)integer*2$$
$$(2)integer*2$$
$$\ldots\ldots$$
$$(n)integer*2$$

For type**XA** arglists, the format is:

$ARGLXAj(0)n$

$(1)character2//character1$

$(2)character4//character3$

...

$(n)characternn(orblank)//character(n-1)$

For types **XR, XD** , each value as well as the length specification word itself requires four bytes. For the other types, it is 2 bytes. The length n is the 'number of words to follow' in the arglist. So if the arglist is empty, $n = 0$. The number of the arglist is given by j in the examples above.

4. Database Schema Syntax and Semantics

The number of signals and the structure of the database is determined by a file known as CALSIG, which in effect enumerates all signal names in a compact tree-structure. One basic symmetry assumption, which simplifies the enumeration of the database, is shown by the following example: If a $< fcnletter >$, say R, existing in nodes 4, 8, and 12, has four "stations" (S's) in node 4, then R must also have 4 stations in each of nodes 8 and 12. In other words, the "tree structure" of a system must be identical for all nodes in which that system exists. Continuing the example, if each station S in turn has 2 cavities (C's), then the RSC's will be described in CALSIG as follows:

- R = radiofrequency system in nodes = 4,8,12
- Each R has 4 stations (RS)
- Each RS has 2 cavities (RSC)
- Each RSC has:
    - class DM
    - class DM
    - ...
    - class AM
    - etc.

In CALSIG, each line beginning with the word "class" defines a collection of signals. In the above example, the 1st "class DM" line defines the $(3 * 4 * 2 = 24)$ RSC/DM1's, the 2nd "class DM" line defines the 24 RSC/DM2's, etc.

CALSIG is intended to be free-field and rich in commentary, but it has an exact syntax by which DBGEN can extract the needed information.

CALSIG defines the "skeleton" of the database. To "flesh out" the database, all information such as display names, addresses, AK's and OF's, etc., must be entered with the help of DBFILL.

4.1 Detailed Syntax of CALSIG

CALSIG is a file, copies of which are maintained in the root directory CALSIG is used for two main purposes: 1) as a repository (where programmers, operators, physicists, and anyone else can find it) for descriptive information about all signals in the database and all hardware that control programs must output data to or acquire data from; and 2) as an exact description (decoded by DBGEN) of all signal names in the database. That is, CALSIG is intended to be read by humans as well as by the computer. This document is intended only to describe the exact way in which the program DBGEN decodes CALSIG.

Each newly edited version of CALSIG should be named CALSIG.DAT. The version to be actually used as input to DBGEN should be renamed CALSIG.SIG, so that the source for the currently running database will always be the most recent version of CALSIG.SIG, regardless of any editing in progress. DBGEN always reads CALSIG.SIG.

The syntax description below will be based on the following example of an excerpt of CALSIG of a hypothetical database.

columns:

....v....1....v....2....v....3....v....4....v....5....v

- line 1: R = radiofrequency system
- line 2: Each R has 4 stations (RS)
- line 3: Each RS has:
    - line 4: class DM
- line 5: Each RS has 1 klystron with controller (RSK)
- line 6: Each RSK has:
    - line 7: class DM local panel control
    - line 8: class DM computer control
    - line 9: class DM klystron system on/off
- line 10: Each RS has 2 cavities (RSC)
- line 11: Each RSC has:
    - line 12: class DM cavity cooling fault /nofault
    - line 13: class AM forward power
    - line 14: class AM reflected power
    - line 15: class DC tuner loop auto/manual
    - line 16: class AC cavity tuning angle control +-5V = +-180deg
    - line 17: class AC spare

- line 18: class XX ARGLXR1(0) 2.*4 derived cavity values
    - (1) value1
    - (2) value2
    - (3) value3
    - (4) value4
    - ARGLXR2(0) 2.*3 more values
    - (1) value1
    - (2) value2
    - (3) value3

The first letter in any signal name denotes the function(V =vacuum, R =rf, P =protection, etc.). The functions are defined in CALSIG by those lines which have a letter in column 1, an "=" in column 3, and optionally another "=" followed by a string of LAN node integers.

Subsystems (branch nodes in the tree) are defined in CALSIG by lines which have "EACH " in columns 1-5, followed by a string of letters, followed by "HAS " (as opposed to "HAS:"), followed by a number (known as the multiplicity of that node), followed somewhere by parentheses enclosing the same string of letters plus one more letter (making the subsystem name). The string of letters (not counting the added letter) must have been previously defined.

Thus line 2 in the example above defines the RS subsystem (R has been previously defined in line 1). Line 5 defines the RSK subsystem, and line 10 defines the RSC subsystem (RS has been previously defined in line 2). A subsystem can have at most seven letters.

For each subsystem, the signals in that subsystem are defined by lines which have " CLASS" in columns 1-7 (blanks in columns 1-2), followed by a valid two-letter class designator (one of D-M,AM,DC,AC, DI,DO). Any sequence of " CLASS" lines must be preceded by a line which has "EACH " in columns 1-5, followed by a previously defined string of letters (subsystem name), followed by "HAS:" (with a colon). The rightmost number (subscript) in the signal name is determined merely by the sequence of these " CLASS" lines.

Thus line 7 in the example above defines the RSK/DM1's, line 8 defines the RSK/DM2's, line 16 defines the RSC/AC1's, etc. For further details, see the comments in programs dbgen.f.

5. Database Generation

5.1 General Discussion

The schema file CALSIG described in the previous section above is provided as the input to the database_generating program DBGEN. DBGEN decodes CALSIG on a line-by-line basis, processing only those lines containing certain keywords or characters and ignoring all other lines. DBGEN converts each line to upper case before processing it.

5.2 How to do a database generation

The insructions to to this are in [spear.doc]dbgen.doc, and are reproduced here:

Rebuilding the Database Tree Structure with DBGEN

The first step in rebuilding the database tree structure is to make all needed changes to the CALSIG source file. Each newly edited version should be named [SPEAR]CALSIG.DAT;n.

The version to be actually used for rebuilding the database tree should be copied into [SPEAR.NSYS]CALSIG.SIG, so that the source for the currently running database will always be the most recent version of CALSIG.SIG. To repeat: you should edit only the ".DAT" file, never the ".SIG" file. DBGEN always reads [SPEAR.NSYS]CALSIG.SIG. NOTE: THE ABOVE IMPLEMENTED AS OF 1-SEP-1984 AT SPEAR

Before rebuilding the database tree structure, you should notify all users that the database will be inaccessible for an hour or so. All programs using the database should be stopped.

- Make sure that the desired version of the CALSIG source file is in [SPEAR]CALSIG.SIG. Make sure that all Vax processes using the database have been stopped. Hang a tape, ring in.
- Sign on: SPEAR
- SPEAR> SET DEFAULT [SPEAR.NSYS]
- SPEAR> @DBSAVE
    - (respond Y)
- . Take the ring out of the tape. Note that the DBSAVE step should not be repeated after running DBGEN. Type:
- SPEAR> RUN DBGEN
- . If DBGEN ran successfully, then re-hang the tape written by DBSAVE, (if its still hung as result of DBSAVE step above, just push ON_LINE) and type:
- SPEAR> @DBRSTR
    - (respond Y
    - ALL)
- SPEAR> SET DEF [SPEAR.INIT]
- SPEAR> @STARTUP
- SPEAR> SET DEF [-.LATGEN]
- SPEAR> RUN LATGEN (for now, so ZFE11 is built)
- >G
- SPEAR> LOGOFF

[SPEAR.NSYS]DBRSTR.COM includes a command to delete all compiled Forth touch-panels, namely: SPEAR> DELETE [PANEL]*.PNC;*.

If desired you can print output file of DBGEN(about 2000 lines).

If new hardware (non-XX) signals have been added, then new ISD files (containing camac addressing information, display names, AK's and OF's, etc.) for these signals should be made, and these ISD files should be fed to DBFILL on the Vax.

If DBGEN fails, you can "backtrack" to the previously existing database by replacing the old CALSIG source file and re-running DBGEN and DBRSTR. Note that DBRSTR requires a successful DBGEN.

6. Database access from a terminal

There are several processes that interact with the database directly from a terminal. The home directory has been setup so that these processes can be invoked by just typing their names after logging onto the home directory. The processes are described below.

Apart from alterations to the database tree structure performed by DBGEN, database updates are roughly speaking of two kinds: 1) changes to address specifications, 2) changes to other fields (DN, AK, OF, etc.) in signal records. In both cases, either DBEXAM or DBFILL can be used.

DBEXAM takes input from a terminal in a conversational style, and is suitable for making relatively few updates to the database, or just examining the stored field values for a small number of signals without necessarily changing them. DBFILL takes input from a file which has been previously prepared with the help of DBGISD and/or any text editor. By convention, such files are named < *name* >.ISD. DBFILL is not conversational, except for entering the names of the input < *name* >.ISD files, and the requested information for the audit file. To invoke any of these processes, just type the desired name from the terminal after logging into the home directory and receiving the vax/vms 'SPEAR>' prompt.

6.1 DBFILL

DBFILL reads the 'individual signal data'(.ISD) files and writes the data into database records as directed. By convention, these files are given vax/vms names ¡filename¿.ISD. Examples of such files are files named *.ISD. Lines in these files can be as long as 120 bytes. DBFILL converts each line to upper case before processing it. Lines that begin with an asterisk are considered to be comment lines and have no effect. DBFILL will list the first 10 comment lines from each ISD file. The syntax for each non-comment line of an >ISD file is one of two forms:

$< SignalName >, < AttributeName >=< value >, < AttributeName >=< value > , .., < attributename >=< value >, < attributename >=< value >, ...$

When a line begins with a comma, it is understood to be a continuation of the last preceding non-comment line. Any number of fields can be specified for a given signal; all other fields for that

signal will remain unchanged. There cannot be more than one ¡signal name¿ on a line. The < *signal – name* > may be preceded by blanks, but not by anything else. Commas must be used only to separate attribute assignment clauses as shown; no other commas may be used (except in comment lines). The second form is:

< *signal – name* >, *ZAP*

This command "ZAP's" the record; it resets the entire record of the specified signal to initialized bytes. Initialized bytes are 'E'(=45hex=69decimal).

For the ascii attributes SU, DN, DP, PL, AN the < *value* > must not contain a comma, and must not be enclosed in quotes. The < *value* > will be left-adjusted in the field, and the rest of the field will be filled with blanks. Embedded blanks within the < *value* > will be retained. If the < *value* > is longer than the field, the extra characters at the right end will be ignored.

After sensing a syntax error of any kind, DBFILL will try to resume correct processing of the input file as soon as possible (generally after the next comma).

DBFILL prompts (via vax/vms device for005) for the name of the .ISD file to be used. It then prompts for the user's name and the reason for making the updates. This additional information is for the database audit file DBUPDATE.AUD, which keeps track of all such database changes. DBUPDATE.AUD is a text file, readable by any text editor. DBFILL will then read the requested .ISD file via for001. Comments and error complaints are written to for007.dat.

6.2 DBGISD

DBGISD generates .ISD vax/vms text files from the running database. These .ISD files can in turn be edited and fed back to the database thru DBFILL. DBGISD can also produce listing (.LIS) files, with the data aligned in vertical columns and the field names absent. These are more convenient for people to read, but cannot be reloaded back into the database. DBGISD takes the information from the Vax global section /DBTREE/.

DBGISD first prompts for output format. The user should respond by typing either "ISD" or "LIS". This response also specifies the default output file name extension. A blank response is treated as "ISD".

DBGISD then prompts for sort key. The user should respond by typing "SN" (signal name), or "DN" (display name), or just ¡return¿ for the default (reloadable) format. A blank response is treated as "SN". If ISD output format was specified, then DBGISD prompts for attribute selection information. The user should respond by typing "ALL", or by typing on one line the names of the fields he wants, or by typing "NOT" followed on the same line by the names of the fields he wants to exclude. For example:

AD, BN, FL, MT

Another example:

NOT DP,PL

A blank response is treated as "ALL".

DBGISD finally prompts for signal selection information and output file names, such as in the following examples:

R/=R.ISD
I/=I30.ISD
BF/AM=BX.ISD
BT/AM=BX.ISD
< *blankline* >

The item on the left of the equal sign is a truncated generic form, that is, a signal name with some or all numbers omitted, and even possibly some of the rightmost letters or the class omitted. Thus, in the above example, BF/AM will include all BF/AM's and all AM's whose names begin with BF (BFJ's, etc.), but not those whose names are just B/AM. Note that for most other programs using generic signal names, the sub-branches are NOT included. To generate a file (say ALL.ISD) containing all sub-branches (that is, all signals) of the database, just specify:

/=ALL.ISD
< *blankline* >

The item on the right of the equal sign is an vax/vms output file name. In the above example, the BF's and the BT's will both be in file BX.ISD. If the ".ISD" is omitted, the file name extension will be either ".ISD" or ".LIS", depending on the output format selected. Control input is terminated by a blank line if from a terminal (or by end-of-file, if vax/vms device for005 was assigned to a file). Each output file will be sorted separately.

Control input (prompted if terminal) is from for005; Commentary and diagnostic output is to for007.dat.

6.3 DBEXAM

To make a small change to the database, use DBEXAM. DBEXAM first prompts for signal name or display name. DBEXAM will show the present values of the attribute record of the requested signal. The user then has the options of changing any of the attributes stored in the record for that signal, or printing the same information being shown on the terminal, or else going on to another signal. If changes are made, then the new values are re-shown, and the user has the same options again. In fact, the updated information (if any) for that signal is not stored back into the database until the user goes on to another signal (by responding with just c/r to the attribute=value prompt). At that point, DBEXAM will prompt for the user's name and the reason for making the updates. This information is appended to an audit_trail file named DBUPDATE.AUD. If the user changes his mind, and wants to leave the last signal unchanged, he can type CNTL_Y to exit from DBEXAM immediately and no changes to the database will be made. After a signal has been

updated, DBEXAM will prompt again for signal name or display name. A response of just c/r to this prompt will cause DBEXAM to exit.

6.4 DBLOOK

The process DBLOOK permits monitoring and setting of database dynamic values. These are the values that are continuously being read from remote cpu's. The user can select any subset of database signals by keying in the desired generic form(s). The generated signals are then displayed in a fixed format called a screen. Up to 20 screens are available, and it is easy to go from one screen to another. Once set up, the entire set of screens can be saved and retrieved from UNIX files, so that signal selection is not necessary each time DBLOOK is run. Upon invoking this process the following prompt is displayed to the user.

HOW TO USE DBLOOK:

Data base LOAD/SAVE: [CTL-B]
asks FILENAME
asks LOAD/SAVE?

Set_up screen with database elements: [CTL-D]
asks SCREEN NO.
displays line numbers
waits for user to type in generic forms for signal names.
accepts generic forms, converting them immediately to lists of
individual signals.

accepts CTL-D to go to different screen for set_up
accepts CTL-E to go to different screen for control and display
accepts CTL-B to LOAD/SAVE data base
accepts CTL-Y to immediately exit from DBLOOK

Display and control data on screen: [CTL-E]

asks SCREEN NO. displays template that includes areas for raw and scientific units continuously refreshes all screen data, including the time accepts the following key characters at any time during refresh:

- UP arrow move cursor UP
- DN arrow move cursor DOWN
- $CTL-A < digits|'.' >$ to change 'AC' setpoint at cursor
- $CTL-A < 0|1 >$ to change 'DC' setpoint at cursor
- $CTL-A < hex\_digits >$ to change 'DO' setpoint at cursor
- $CTL-AI < digits|'.' >$ to increment 'AC' setpoint at cursor

- $CTL-E$ goto new control screen for display
- $CTL-D$ goto new control screen for setup
- $CTL-B$ prepare to LOAD/SAVE the screens
- $CTL-Y$ immediate exit from DBLOOK

The following example illustrates the use of this program.

- SPEAR> DBLOOK
- [$CTL-D$]
- SCREEN#=1
- TC/AM. (generates readback signals for all trims)
- [$CTL-E$]
- SCREEN#=1(displays readback values for all trims)
- [$CTL-B$]
- SCREEN#=2
- TC/AC. (generates setpoint signals for all trims)
- [$CTL-D$]
- SCREEN#=3
- TC/DM. (generates status signals for all trims)
- [$CTL-E$]
- SCREEN#=3 (switch between screens to observe values)
- [$CTL-E$]
- SCREEN#=1
- [$CTL-E$]
- SCREEN#=2
-
- DNarrow (move cursor down to 3rd trim)
- DNarrow
- DNarrow
- [$CTL-A$] 3.22 (change this setpoint to 3.22amps)
- [$CTL-B$] (save all screens into a UNIX file)
-
- FILENAME=TRIMS.DBLOOK
- SAVE/RESTORE? SAVE
- [$CTL-Y$] (exit DBLOOK)
- . . .
-

- SPEAR> DBLOOK
- [$CTL-B$] (later, retrieve all screens from file)
- FILENAME=TRIMS.DBLOOK
- SAVE/ RESTORE? RESTORE
- [$CTL-E$]
- SREEN#=2

- etc.

7. Access to the database through user-written software

Application programs can only access the SPEAR database on the SPEAR account of the SPEAR computer. If the computer has been turned off or has been rebouted, the STARTUP command must be given to activate the database:

SPEAR> STARTUP

The database subroutines that the programmer can invoke to manipulate database signals,classes, id's, attributes, etc are discussed in this section. Before calling any of this subroutines the programmer first call DBMAPXX. This maps the database to his program:

CALL DBMAPXX

The database access subroutines are as follows:

- Subroutine GETID Create lists of signal IDs from strings of generic forms (database names) for quick data base access.
- Subroutine COLLECT Get a signal attribute for each member of a list of signals IDs. Display names, scales, offsets, and units are some sample attributes.
- Subroutine DISTRIB Set a signal attribute for each member of a list of signals IDs.
- Subroutine READDB Get data out of Run Time Data Base(read values received from remote cpu's) from database IDs. Raw values.
- Subroutine READSU Get data out of Run Time Data Base(read values received from remote cpu's) from database IDs. Values scaled to appropriate units.
- Subroutine SETDB1 Put data into Run Time Data Base(send values to remote cpu's) from database IDs. Raw values.
- Subroutine SET1SU Put data into Run Time Data Base(send values to remote cpu's) from database IDs. Values scaled to appropriate units.

Software that calls these subroutines need to be linked the database routines:

```
SPEAR> LINK YOUR-PROGRAMS, -
[SPEAR.NSYS]PCDL/LIB, -
[SPEAR.COMMON]L/LIB
```

7.1 GETID - Get list of IDS from database.

DESCRIPTION:

This routine allows the user to create signal ID lists, which in turn permit access to both the atttributes and the run time values of collections of database signals using subroutines in the following sections below.

FORTRAN FORMAT:

CALL GETID (GENFORMS, NIDMAX, IDLIST, NID)

ARGUMENTS:

INPUT:

GENFORMS I*2 The text string of generic forms. This string expresses all of the names of the signals whose ID's are to be fetched.

NIDMAX I*2 The maximum number of ID's to be generated. This should always be less than or equal to the dimension of array IDLIST in the caller's program.

OUTPUT:

NID I*2 The number of ID's actually generated by GETID.

IDLIST I*2 The list of generated ID's. This is an array in the caller's program where the signal ID's will be placed by GETID.

EXAMPLES:

Example 1:

```
INTEGER*2 STEERING_CURRENT_IDS(10), NIDS
CALL GETID('SLF/AM1.', 10, STEERING_CURRENT_IDS, NIDS)
```

In this example, the programmer wishes to get the database IDs for the 10 beamline steering currents. SLF/AM1 is the readback database name for the steering currents. These database names are found in CALSIG.

Example 2:

```
CHARACTER*40 RFSUBSET/'RS(2;4)/DI, RSC/DM(1:3).'/
INTEGER*2 IDLIST(26)
INTEGER*2 NID
CALL GETID(%REF(RFSUBSET),26, IDLIST, NID)
```

Example 3:

```
INTEGER*2 IDLIST(26)
INTEGER*2 NID
```

CALL GETID('RS(2;4)/DI,RSC/DM(1:3).',26, IDLIST, NID)

The period '.' in the specification of GENFORMS is important! It indicates that the input list of database names is done.

C PROGRAMMING FORMAT:

call getid (genforms, &nidmax, idlist, &nid)

genforms char[]: the text string of generic forms. This string expresses all of the names of the signals whose ID's are to be fetched.

nidmax int: The maximum number of ID's to be generated. This should always be less than or equal to the dimension of array idlist in the caller's program.

nid int: The number of ID's actually generated by GETID.

idlist int[]: The list of generated ID's. This is an array in the caller's program where the signal ID's will be placed by GETID. The ID's are packed as fortran integer*2 variables.

Example 1:

```
char rfsubset[]="rs(2;4)/di, rsc/dm(1:3)." ;
... int idlist[26];
int nid,imax;
...
imax=26;
getid( rfsubset, &imax, idlist, &nid);
```

Example 2:

```
int idlist[26];
int nid,imax;
...
imax=26;
getid( "rs(2;4)/di,rsc/dm(1:3).", &imax, idlist, &nid);
```

7.2 COLLECT - Get signal attributes from the 'static' database.

DESCRIPTION: This routine fetches the specified attribute for each signal whose whose ID is in the array IDLIST.

FORTRAN FORMAT:

CALL COLLECT(NID, IDLIST, ATRNAME, ATRVALUES)

ARGUMENTS:

INPUT:

NID I*2 The number of signal ID's in array IDLIST.
IDLIST I*2 The list of signal ID's, as generated by GETID.
ATRNAME I*2 A single 2 byte ASCII attribute name.
Commonly used valid names are:

'SC' get signal class
'TM' get time of most recent change
'SN' get signal name
'SU' get scientific units
'DN' get operational display name
'DP' get descriptive phrase
'CO' local area network node number
'BR' camac branch number
'CR' camac crate number
'MN' camac module number
'MT' module type
'SA' camac subaddress
'FC' camac function code
'BN' bit number
'PL' get physical location
'AK' get scale
'OF' get offset
'MI' get mininum value
'MA' get maximum value
'CK' constant
'TO' tolerance value
'FL' field width
'RB' get ID of readback from ID of setpoint

OUTPUT:

ATRVALUES I*2 A list of attribute values, one for each of the signal ID's in the array IDLIST. A total of NID*L bytes will be returned, where L=the length of the attribute, which is given in section 3.4 above.

EXAMPLES:

Example 1:

```
CHARACTER*40 RFSUBSET/'RS(2;4)/DI, RSC/DM(1:3).'/
INTEGER*2 NID,IDLIST(26)
REAL*4 AK(26),OF(26)
CHARACTER*12 DISPLAY_NAMES(26)
CALL GETID( %REF(RFSUBSET), 26, IDLIST, NID)
CALL COLLECT( NID,IDLIST,'AK',AK)
CALL COLLECT( NID,IDLIST,'OF',OF)
CALL COLLECT( NID,IDLIST,'DN',%REF(DISPLAY_NAMES) )
```

C PROGRAMMING FORMAT:

collect(&nid, idlist, atrname, atrvalues)

nid int: The number of ID's (generated by getid) in array idlist.

idlist int[]: The list of generated ID's.

atrname char[2]: A 2byte ASCII attribute name.

atrvalues char[]: A list of attribute values, one for each of the signal ID's in the array IDLIST. A total oF nid*len bytes will be returned, where len=the length of the attribute, which is given in section 3.4 above.

Example 1:

```
char rfsubset[]="rs(2;4)/di, rsc/dm(1:3)." ;
...
int idlist[26];
int nid,imax;
float ak[26],of[26];
char display_names[26][12+2];
...
getid( rfsubset, &imax, idlist, &nid);
collect( &nid,idlist,"ak",ak)
collect( &nid,idlist,"of",of)
collect( &nid,idlist,"dn",display_names )
```

Note: The extra two bytes declared in each row of char[][] arrays is necessary so that function collect can insert the terminating null byte needed in 'C' codes.

7.3 DISTRIB- Set data in attribute records of software signals.

The COLLECT subroutine described above can be used to get data for software records, since this data is in the 216 byte(=108 word) attribute named XR. There is an inverse subroutine DISTRIB with an identical call sequence that can put generated data back into the XR attribute for software records. Caution should be exercised in the use of DISTRIB; It changes the database attribute records without notifying the audit trail file. DISTRIB does update the record's time stamp however. It is good practice to always use DISTRIB only after a call to COLLECT as shown in the examples below. If restricted in this way, it has the same effect as a typical database 'update' function, which is known to reduce the possibility of an accidental trashing of the database.

CALL COLLECT(NID, IDLIST, ATRNAME, DATAVALUES)

CALL DISTRIB(NID, IDLIST, ATRNAME, DATAVALUES)

COLLECT fetches the data array for each software signal whose whose ID is in the array IDLIST. DISTRIB copies the data array to the database records for each software signal whose ID is in IDLIST.

NID I*2 The number of software signal ID's in array IDLIST.

IDLIST I*2 The list of signal ID's, as generated by GETID.

ATRNAME I*2 A single 2byte ASCII attribute name; it should be one of the software data attributes: XR,XD,XI,XA.

DATAVALUES R*4 For COLLECT, this is the array that will receive the data; for DISTRIB, this is the array that holds the data to be copied to the database records of the signal ID's in the array IDLIST. A total of NID*L bytes will be transferred, where L=the length of the attribute, which is =216 for attributes XR,XD,XI,XA.

Example 1:

```
CHARACTER*40 SOFTWARESIGS/'ZB(2;4)/XX1,ZB1S(1:8)/XX1.'/
INTEGER*2 NID,IDLIST(10)
REAL*4 DATA_ARRAY(0:10*216/4-1)
CALL GETID(%REF(SOFTWARESIGS), 10, IDLIST, NID)
CALL COLLECT( NID,IDLIST,'XR',DATA_ARRAY) !get the data
CALL CALCULATE(NID,DATA_ARRAY) !modify the data
CALL DISTRIB( NID,IDLIST,'XR',DATA_ARRAY) !put the data
```

7.4 READDB - Get 'dynamic' raw data from input hardware.

FORTRAN FORMAT

CALL READDB(NID, IDLIST, VALUES) or

STATUS = READDB(NID, IDLIST, VALUES)

This routine reads raw values from the run-time database, which is being continuously refreshed from remote cpu's by the process XIO.

NID I*2 Number of signal ID's in the array IDLIST

IDLIST I*2 List of signal ID's

VALUES R*4 User's array where data will be stored. Values read in as integers will be converted to floating point. A single class DM status bit, for example will be returned as either 1. or 0.

STATUS I*2 0: No error -1:Error

C PROGRAMMING FORMAT

readdb(&nid, idlist, values) or

status freaddb(&nid, idlist, values)

This routine reads raw values from the run-time database, which is being continuously refreshed from remote cpu's.

nid int: The number of ID's (generated by getid) in array idlist.

idlist int[]: The list of generated ID's.

values float[]: User's array where data will be stored. Values read in as integers will be converted to SUNos fortran floating_point. A single class DM status bit, for example will be returned as either 1. or 0.

status long: this is returned as a 4byte integer;

1: ok, no error

0: error

Example 1:

```
char rfsubset2[]="rsc(1:4)/ac, rsc(1:4)/am." ;
...
int nid,idlist[8];
int imax,n;
float values[8], ak[8],of[8];
char display_names[8][12+2];
char units[8][12+2];
...
idmax8;
getid( rfsubset2, &imax, idlist, &nid);
collect( &nid,idlist,"ak",ak)
collect( &nid,idlist,"of",of)
collect( &nid,idlist,"dn",display_names )
```

collect(&nid,idlist,"su", units)

readdb(&nid, idlist, values)

for (i0; i<8; i++) values[n] āk[n]*values[n] + of[n] ;

7.5 READSU - Get 'dynamic' scaled data from input hardware.

CALL READSU(NID, IDLIST, VALUES) or
STATUS = READSU(NID, IDLIST, VALUES)

This routine reads converted values from the run-time database, being continuously refreshed from remote cpu's by the process XIO. XIO. The term 'converted' means that the class AC and class AM signal values are coverted to scientific units by the formula:

$$< converted - value >= AK * < raw - value > + OF$$

NID I*2 Number of signal ID's in the array IDLIST

IDLIST I*2 List of signal ID's

VALUES R*4 User's array where data will be stored. Values read in as integers will be converted to floating point. A single class DM status bit, for example will be returned as either 1. or 0.

STATUS I*2 0: No error -1:Error

Example 1:

```
CHARACTER*40 RFSUBSET/'RSC(1:4)/AC,RSC(1:4)/AM.'/
INTEGER*2 NID,IDLIST(8)
REAL*4 AK(8),OF(8)
REAL*4 VALUES(8)
CHARACTER*12 UNITS(8) CALL GETID( %REF(RFSUBSET),8, IDLIST, NID)
CALL COLLECT( NID,IDLIST,'AK',AK)
CALL COLLECT( NID,IDLIST,'OF',OF)
CALL COLLECT( NID,IDLIST,'SU',%REF(UNITS) )

CALL READDB( NID, IDLIST, VALUES)
DO N=1,8
VALUES(N) = AK(N)*VALUES(N) + OF(N)
ENDDO
```

Example 2:

```
CHARACTER*40 RFSUBSET/'RSC(1:4)/AC,RSC(1:4)/AM.'/
INTEGER*2 NID,IDLIST(8)
```

```
REAL*4 VALUES(8)
CHARACTER*12 UNITS(8)
CALL GETID( %REF(RFSUBSET),8, IDLIST, NID)
CALL COLLECT( NID,IDLIST,'SU',%REF(UNITS) )
CALL READSU( NID, IDLIST, VALUES)
```

Both examples produce the same numbers in array VALUES. The conversion constants AK,OF, should be such that the units of the conversion are given by the text attribute SU.

Example 3:

```
CHARACTER*40 RFSUBSET/'RSC(1:4)/AC.'/
INTEGER*2 NID,IDLIST(4),IDLISTRB(4)
REAL*4 VALUES(4),VALUESRB(4)
CHARACTER*12 UNITS(4),UNITSRB(4)
CALL GETID( %REF(RFSUBSET),4, IDLIST, NID)
CALL COLLECT( NID,IDLIST,'RB',IDLISTRB ) !get rdbk ID's
CALL COLLECT( NID,IDLIST, 'SU',%REF(UNITS )) !get stpt units
CALL COLLECT( NID,IDLISTRB,'SU',%REF(UNITSRB)) !get rdbk units
CALL READSU( NID, IDLIST , VALUES ) !get setpts
CALL READSU( NID, IDLISTRB, VALUESRB) !get rdbks
```

This example also produces the same numbers as the other two examples above, but starts with only the class /AC (setpoint) database ID's.

7.6 SETDB1 - Send 'dynamic' raw data to output hardware.

```
CALL SETDB1( NID, IDLIST, VALUE ) or
STATUS = SETDB1( NID, IDLIST, VALUE )
```

This routine sends raw values to the run-time database.

NID I*2 Number of signal ID's in the array IDLIST

IDLIST I*2 List of signal ID's

VALUE R*4 Single setpoint to which all signals will be set. The value will be converted from floating point to an integer. A single class DC status bit, for example must be either 1. or 0.

STATUS I*2 0: No error -1:Error

C PROGRAMMING FORMAT

setdb(&nid, idlist, values) and
setdb1(&nid, idlist, &value) , or
status = setdb(&nid, idlist, values) and
status = setdb1(&nid, idlist, &value)
nid int: The number of ID's (generated by getid) in
array idlist.
idlist int[]: The list of generated ID's.
values float[]: User's array where data will be stored. Values
read in as integers will be converted to SUNos fortran
floating_point. A single class DM status bit, for
example will be returned as either 1. or 0.
value float: User's single value.
status long: this is returned as a 4byte integer;
1: ok, no error
0: error
Example 1:

```
char rfsubset3[]="rsc(1:4)/ac, rsc(1:4)/dc." ;
...
int nid,idlist[8];
int imax,n,nid2;
float values[8],value, ak[8],of[8];
char display_names[8][12+2];
char units[8][12+2];
...
idmax=8;
getid( rfsubset2, &imax, idlist, &nid);
collect( &nid,idlist,"ak",ak)
collect( &nid,idlist,"of",of)
collect( &nid,idlist,"dn",display_names )
collect( &nid,idlist,"su", units )
for (i=0; i<4; i++) values[n] = (values[n] - of[n])/ak[n] ;
setdb( &nid, &idlist[1], values) ;
nid2=4.;
value=1.;
setdb1( &nid2,&idlist[5], &value) ;
```

7.7 SET1SU - Send 'dynamic' scaled data to output hardware.

CALL SET1SU(NID, IDLIST, VALUE) or
STATUS = SET1SU(NID, IDLIST, VALUE)

This routine sends converted values to the run-time database, The term 'converted' means that the class AC signal value is coverted from scientific units by the formula:

$$< raw - value >= (< converted - value > - OF)/AK$$

NID I*2 Number of signal ID's in the array IDLIST

IDLIST I*2 List of signal ID's

VALUE R*4 User's setpoint value in converted units. This

value will be converted from floating point to an integer. A single class DC status bit, for example must be either 1. or 0. STATUS I*2 0: No error -1:Error

Example 1:

```
CHARACTER*40 RFSUBSET/'RSC(1:4)/AC,RSC(1:4)/DC.'/
INTEGER*2 NID,IDLIST(8)
CHARACTER*12 UNITS(8)
CALL GETID( %REF(RFSUBSET),8, IDLIST, NID)
CALL COLLECT( NID,IDLIST,'SU',%REF(UNITS) )
CALL SET1SU( 4, IDLIST(1), 10.3 ) !set thes to 10.3volts
CALL SET1SU( 4, IDLIST(5), 1. ) !turn these ON
```

8. Internals

8.1 Structures of the shared global memory sections

There are two main shared global memory sections in this system. DBTREE.GBL and DBAS.GBL. DBTREE.GBL contains the 'static' attribute data of all of the signals and has a relatively easy-to-describe internal structure. It is made up of a number of 256 byte (or 128 integer*2 word) records. The first few records contain the structure keys generated by process DBGEN. These keys can be printed from UNIX file FOR007.DAT after DBGEN is run. The rest of the records are for the attributes of all of the signals. The exact structure is given in the file [.NSYS]DBCOMT.FOR which is included in many of the database routines.

The global memory section DBAS.GBL has a more complex structure. It is given in the two files [.NSYS]DBCOMH.FOR, DBCOMH1.FOR. These two files are also included in several of the database routines.

The size of the global sections must be given at compile time. The size of each of the two sections depend on parameters declared in the source .FOR files mentioned above.

The independent parameters are to be found listed at the top of the .FOR file given in the table below. All other sizes are generated from these parameters.

Parameter Current value Filename Description

HIGIDMX 500 DBTREE.FOR upper limit for # database ID's

PAKMX 2000/4? DBCOMH.FOR max # of bytes in ARC token ring transfer.

To change any of these independent parameters:

- set default to [.NSYS]
- stop all processes using the Real Time Database System
- Edit the appropriate .FOR file
- If only DBCOMH.FOR was edited, then run comfile SPEAR> @DBCOMCLL.COM to recompile or assemble all of the affected source code, and to relink all the processes.
- If both .FOR files were edited, then run comfile SPEAR> @DBCOMTCLL.COM to recompile or assemble all of the affected source code, and to relink all the processes.