Alarms & Limits

John J. Walsh

April 2, 1990

1 Introduction

The Alarms and Limits system (AL) was developed by the Penn group and others to monitor and control high and low voltages for the CDF detector. It is important to note the difference between the terms "monitoring" and "control." Monitoring entails reading out voltages (or some other values) and comparing then to known tolerances. Controlling means modifying high voltage values on the detector. Alarms and Limits is a Camac based system that monitors and controls a wide variety of hardware and a large number of channels (~10k). The computing power (such as it is) is provided by a VAX 750 (B0SCCC) and about five LSI-11 computers which reside in Camac.

Figure 1 shows a schematic representation of the AL system. At the lowest level are the quantities on the detector (or in the counting room) that must be monitored or controlled. These quantities (voltages, usually, but sometimes temperature, etc.) are read out via CAMAC. Each channel has a unique CAMAC address, and these addresses serve as channel ID's. Some portion of the CAMAC crates involved contain an LSI-11 computer, which runs the code necessary to read the channels associated with that particular crate. The LSI also checks for out of tolerance conditions on the channels and stores the data until it is time to upload the values to the VAX. If a channel is found to be out of tolerance by the LSI, it generates an interrupt on the VAX and uploads the alarm information. The VAX receives this information and updates the graphics display screen in the control room.

The VAX polls the CAMAC crates in a synchronous loop to obtain the latest data. If an LSI is present in a particular crate, the VAX uploads the most recently read data using a block CAMAC transfer. The data is stored on the VAX in the Current Values Global Section[1] (CV). If an LSI is not present in the crate, then the VAX must perform the CAMAC operations necessary to read out the channels. The the VAX also checks for an alarm condition and stores the data away in the CV.

There are several programs that run on the VAX performing these operations. VAX_SCANNER is the program that polls the CAMAC crates for data. LAM_MONITOR is the VAX program that receives interrupt messages from the LSI's. MONITOR receives alarm information (from both VAX_SCANNER and LAM_MONITOR) and updates the AL display terminal in the B0 control room. There is also a program called HISTORY, which keeps history records for specified channels. These four programs run as batch jobs

[1] A global section is a common block that can be accessed by more than one process.

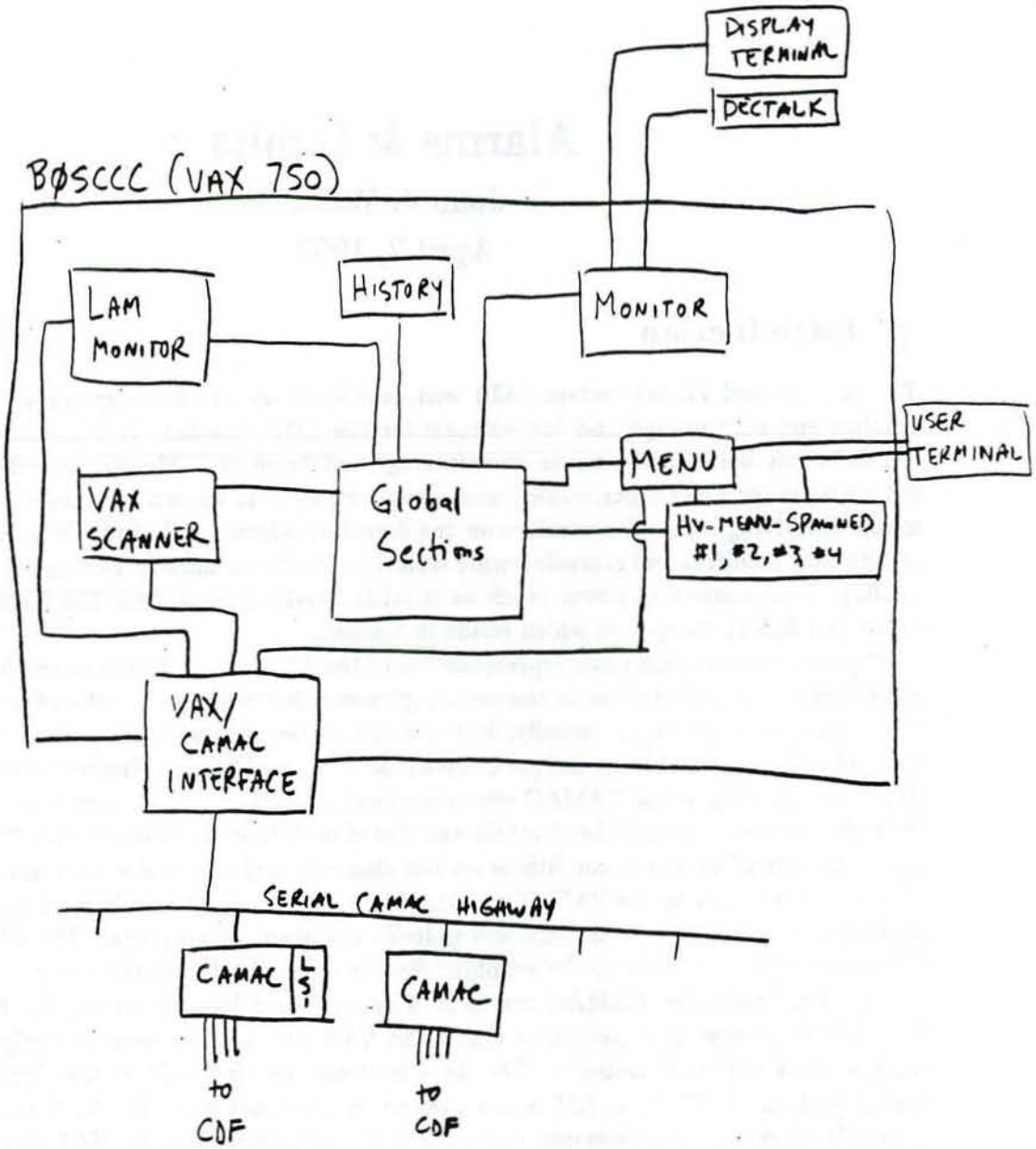

Figure 1: Schematic representation of Alarms & Limits.

on the B0SCCC machine. Additionally, there is the user interface program called MENU, which allows the user to perform control functions and also to view a wide variety of status conditions. MENU is an interactive program.

The next section will deal with what I call *micro AL*, that is the smallest pieces of AL. These include a description of the channels and their data structures, some of the routines that manipulate channels and some of the basic programming techniques and strategies. Then there will follow a section on *macro AL*, which deals with the bigger picture of how the AL programs interact with each other. There will also be sections on the LSI (including some gruesome details on VAX/LSI communications), the various high voltage control systems, etc.

2 *Micro* Alarms & Limits

There are about 10 thousand channels in AL, and that number is increasing with time. Each channel has a unique CAMAC address, and it is this address that identifies the channel. The CAMAC address consists of a crate, a slot, a subaddress, and a function code (CNAF). The function code may or may not have anything to do with the CAMAC function code that is used to read or write a channel (most often it does not). There are some channels that in fact are not CAMAC-based (e.g. Gas Gain channels, CTC channels), but they are assigned a fictitious CAMAC addressed so they fit in the the basic AL addressing scheme.

All the information for a particular channel is contained in a single record of the current values array, which makes up part of the CV (current values global section). This information is either permanent (e.g. CAMAC address) or temporary (e.g. most recent value). The permanent information is stored on disk in the file ALARMS$DATA:LIMITS.DAT[2]. When AL is started up, the CV is initialized by reading the LIMITS.DAT file and transfering the permanent data to the CV. The temporary data is filled in as the detector components are read out.

2.1 AL database

Database is the word that loosely refers to the list of channels in AL, and the accompanying information. This information is stored in two places: (i) in the permanent disk file LIMITS.DAT, and (ii) temporarily in the VAX memory (the CV).

The LIMITS file is an unformatted file which is read at AL initialization time. When new channels are added to the system, or old channels are removed, or a new configuration is desired (e.g. changing the ID of a CAMAC crate), the LIMITS file must be modified. A DCL command file called ALARMS$WORK:EDIT_LIMITS.COM is used to make these modifications. This command file formats the LIMITS file and invokes the LSE editor to allow the user to make the necessary changes. The user then types in the new records using the editor. Care must be taken to follow the prescribed format (details coming soon) and also *the channels must be maintained in CAMAC order.* If the LIMITS file contains records

[2]Logical names are obtained on the B0 cluster by typing `setup alarms all`.

out of order the initialization procedure will not work. A sorting program should probably be written for the LIMITS file, but to date this has not been done.

The following fragment is a section of the LIMITS.DAT file. This comes from the formatted file created by EDIT_LIMITS.COM.

```
    0     6     3     9     0 3001
      0.000       4.000       2.000       5.000     0
Current -15V
    0     6     3    10     0 3001
      0.000     100.000      10.000      20.000     0
Inlet Water Temp
```

These lines correspond to 2 records in the file. Each record requires 3 lines in the formatted file. I will describe the fields now. The first line contains 6 integers, the first 5 of which are:

```
    0     6     3     9     0
```

These five numbers are the camac address. Branch, crate, slot, subaddress and function code. This group of numbers must be unique for each channel, and must satisfy the following restrictions.

- Branch = 0 (always)
- Crate ranges from 1–62
- Slot ranges from 1–18
- Subaddress ranges from 0–8191 (this is not the normal range of CAMAC subaddresses)
- Function code ranges from 0–31 (Note: For PISA high voltage systems: subaddress = 256*box + channel, where box is the wedge number, and function = 0)

There is one more integer on the first line. It is in hex format:

```
3001
```

This is the ASR, Alarm Status Register whose bits signify various things. For the purposes of constructing the limit table:

- bits 12:15 - Device code, usually specifies the type of CAMAC module that is being read out.
- bit 0 - set = alarm monitoring disabled (Generally, when bringing up a new system, it's a good idea to set this bit. This prevents thousands of alarms when debugging the system.)
- other bits have other meanings not relevant for constructing the LIMITS.DAT file.

The next line contains 4 real numbers and 1 integer:

```
0.000        4.000        2.000        5.000      0
```

The 4 reals are:

- low tolerance
- high tolerance
- set value (nominal value)
- scale factor (the scale factor is only used in displaying)

The one integer on this line is not used. Its space can be used if needed. The next line is a character*30 description field `Current -15V`. It is only used to describe the channel for the display.

It is important to format these records in the LIMITS.DAT file correctly (actually in the temporary formatted version of LIMITS.DAT created by EDIT_LIMITS.COM). Since in this LaTeX output it is difficult to count spaces, I refer the reader to the file ALARMS$DOC:DATABASE.ALARMS, which contains sample records in the correct format.

The information stored in the LIMITS.DAT file is loaded into the CV when the AL system is initialized. Each record (i.e. channel) is contained in one element of the FORTRAN array CV_ARR (Current Values ARRay). This array uses the VAX Fortran record structure features, and so the elements are usually called records. Records of the CV_ARR will be refered to as CV records. Consult the FORTRAN manual for information on record structures. Briefly, it allows you to define an entity (record) which contains several different data types. This way one record can contain a mixture of real, integer, logical and character variables.

2.2 Logical names

Records are logically grouped together using logical names. These are AL logical names, not to be confused with VAX logical names. For example, the photomultiplier tubes of the CEM for wedge 20 east (there are 20 such tubes) are grouped together under the logical name WEDGE20E_EM. A logical name is a character string of up to 30 characters (usually they are shorter). The logical name assignments are stored in a data file and are loaded at AL startup. At present there are about 450 logical names. An array of these names is also stored in the global section (NODE_ARR) and the CV record for each channel contains a pointer into this NODE_ARR. (Not all channels are associated with a logical name, in which case the pointer is zero.)

Channels are assigned to logical names at system startup. This is accomplished by reading in a text file called ALARMS$SOURCE:RESOURCE.DAT, which contains the information necessary to make the logical name assignments.

The following fragment is a section of the AL RESOURCE.DAT file.

```
FBM_2_22G_TP
    0   12    6    0   10
FBM_2_22G_SN
    0   12    7   14    2
    0   12    7   10    4
    0   12    6   10    6
    0   12    6   26    6
```

There are two logical name entries in this sample. The first one is simpler, so I'll describe it first.

```
FBM_2_22G_TP
    0   12    6    0   10
```

The logical name is FBM_2_22G_TP, and it is used by the user programs to refer to a group of channels. In this case the channels are low voltages from a FASTBUS crate. The five fields are branch, crate, slot, offset, and number.

```
branch \
        \
crate    > from CAMAC address of first channel in group
        /
slot   /

offset - offset from first channel in slot

number - number of consecutive channels to include in grouping
```

These five numbers define a group of consecutive channels from LIMITS.DAT. The *first channel of this group* is defined by part of its CAMAC address (branch, crate, slot) and then by its position within the slot (offset). The last field (number) specifies the number of consecutive channels to be included in the group. Here's the example again:

```
FBM_2_22G_TP
    0   12    6    0   10
```

These numbers specify branch 0, crate 12, slot 6. The offset is 0, which means that we begin with the first channel "in the slot." That is, the channel in the slot with the "lowest" camac address. Since the database is sorted by CAMAC address, this channel is the first (or zeroeth) channel "in the slot." The 10 means take 10 channels, in this case the first 10 channels in the slot.

The requirement of consecutive channels is not restrictive, because you can specify as many groups of channels as you want to be assigned to a logical name. For example:

```
FBM_2_22G_SN
    0   12    7   14    2
```

```
0   12    7   10    4
0   12    6   10    6
0   12    6   26    6
```

Here there of four sub-groups of consecutive channels that are assigned to the logical name FBM_2_22G_SN. The channels all come from the same crate, but from two different slots. Also, the order in which they will appear when displayed can be controlled here, because the order in which they are specified in this file is the order in which the channels will subsequently be retrieved from the database.

2.3 Some programming techniques

In this subsection I present some of the routines available to manipulate CV records individually and in groups (specified by their logical names). Typically, a programmer might want to view the voltages on the channels corresponding to a particular logical name. For example, one might want to read the high voltages on the plug calorimeter. In this case, you need to know the logical name for the group of channels that you are interested in. The desired logical name can be obtained by making the appropriate menu selections in the interactive program MENU. Once the logical name is known, there is a very handy subroutine that returns a list of indices into the CV_ARR (these indices are called CV pointers) for channels corresponding to the logical name. Here it is:

```
call Logical_to_Ptr(Logical_name,num_ptr,ptr_arr)

                          !see ALARMS$SOURCE:Logical_to_Ptr.For
```

LOGICAL_NAME is the character*30 logical name. NUM_PTR (I*4) is the number of CV pointers found for this logical name. PTR_ARR (I*4) is an array containing these pointers. Given the CV pointers, the next step is to fetch these records from the CV_ARR. To read or write a single record from/to the CV_ARR we use two analogous subroutines. The input to the routine is the record number and the output is the record. The CV_ARR element is copied into a local variable. Here they are:

```
call Read_CV_Rec(rec,ptr)          !see Alarms$Source:MEMORY3.FOR
call Write_CV_Rec(rec,ptr)
```

PTR is the CV pointer (I*4) and REC is the local variable to contain the record. REC has the FORTRAN record structure (called CV_REC_TYPE) defined in ALARMS$SOURCE:CV_REC_TYPE.INC. (All .FOR and .INC files mentioned in this note can be found in ALARMS$SOURCE unless otherwise specified). It turns out that Read_CV_Rec and Write_CV_Rec are more suited to reading one or two channels than to reading a list of channels (e.g. the list returned by the call to Logical_to_Ptr). There is a certain amount of overhead involved in accesssing the global section. So, the following routine reads a list of channels:

```
Get_Recs(num_ptr,ptr_arr)      !Alarms$Source:GHV_Extras.For
```

The records specified in the list will be written to an array in common called Rec_Local. The elements of Rec_Local have the same structure as the CV records. You will need the following include file for the common block:

```
GHV_Alarms.Inc
```

Similarly, there are routines to write a list of records to the global section (`Put_Recs(num_ptr,ptr_arr)`) and to write a list of records to both the global section and permanent disk file (`Write_Recs(num_ptr,ptr_arr)`). You must use `Write_Recs` if you want the information to be stored permanently. Otherwise, it will be lost the next time AL is restarted.

Once the desired CV records have been read into a local variable, the individual fields can be accessed via routines written for that purpose. The files CV_ACCESS.FOR and CV_ACCESS2.FOR contain the routines to access the individual fields of the CV records. For example:

```
crate = crate_r(rec)    !crate_r is a function -- CV_Access.For
setv  = set_value_r(rec)
call set_tolerance1(value,rec)
```

The first two examples read from the local variable REC the field of interest (the CRATE and SET_VALUE fields in this case). The third example writes a new value of the lower tolerance (tolerance1) into REC. Remember, REC is only a local variable here. This write will not be saved in common unless you replace the new record with WRITE_CV_REC (or Put_Recs or Write_Recs).

Here is an example using these routines. Let's say I want to print out on the screen the set values and current values of the endplug high voltage channels. I need to know beforehand that the correct logical name for the endplug high voltage channels is PLUG_HV. The code fragment would then be:

```
C
C Necessary includes/declarations for this fragment.
C
      include 'alarms$source:cv_rec_type.inc' !structure def for cv record
      include 'alarms$source:cv_access.inc'   !function declarations
C                                             !(set_value_r, curr_value_r)
      integer num_ptr,ptr_arr(500),ptr,i
      real sv,cv
      record /cv_rec_type/ rec
C
C First get cv ptrs for this logical.
C
      Call logical_to_ptr('PLUG_HV',num_ptr,ptr_arr)
```

```
C
C Now loop over the pointers getting the records as we go.
C
      do i = 1, num_ptr
C
C get CV pointer from array
C
        ptr = ptr_arr(i)
C
C read that record from CV\_ARR into local record (rec)
C
        call read_cv_rec(rec,ptr)
C
C get set and current values from local record
C
        sv = set_value_r(rec)
        cv = curr_value_r(rec)
C
C write to screen (or whatever)
C
        write (*,*) sv,cv
C
C end loop over ptrs
C
      end do
```

For the purposes of illustration i've used the routine Read_CV_Rec, but I said above that this routine is not efficient for reading a list of channels. So here is the same fragment using the preferred routine Get_Recs:

```
C
C Necessary includes/declarations for this fragment.
C
C This include file contains the common block with has the array rec_local.
C It also has cv_rec_type.inc included, so you don't need it here.
C ------------------------------------------------------------------
      include 'alarms$source:ghv_alarms.inc'
C
      include 'alarms$source:cv_access.inc'   !function declarations
C                                             !(set_value_r, curr_value_r)
      integer num_ptr,ptr_arr(500),ptr,i
      real sv,cv
C
```

```
C First get cv ptrs for this logical.
C
        Call logical_to_ptr('PLUG_HV',num_ptr,ptr_arr)
C
C Get the records out of the global section.
C
        Call Get_Recs(num_ptr,ptr_arr)
C
C Now loop over the pointers accessing the records in rec_local
C
        do i = 1, num_ptr
C
C get set and current values from local record
C
          sv = set_value_r(rec_local(i))
          cv = curr_value_r(rec_local(i))
C
C write to screen (or whatever)
C
          write (*,*) sv,cv
C
C end loop over ptrs
C
        end do
```

2.4 Locking

As mentioned above the CV_ARR is located in a global section, which is available to several different processes simultaneously for reading and writing. Since these reads and writes can occur at any time, it is necessary to employ some sort of locking procedure to prevent loss of information. We have used the VAX locking routines, which are described in detail in the VAX manual on locking. The routines described above for reading and writing the CV array make calls to the appropriate locking routines to avoid overwriting. The reason Read/Write_CV_Rec is inefficient for reading/writing many CV records is because each call to Read_CV_Rec involves a locking and unlocking operation. The Get_Recs version does the locking just once and fetches the list of CV records while the CV array is locked, making it more efficient for fetching (or writing) multiple records.

In certain situations it could be useful to have routines analogous to Read/Write_CV_Rec that do not lock the array. For example, the routines Get/Put/Write_Recs could use such a routine. Such routines do exist and are contained in the file MEMORY3.FOR. If they are used, however, the programmer must be sure to do the locking explicitly. The locking routines are contained in the file CV_ARRAY_LOCKS.FOR.

It is widely believed that locking the CV array is a significant source of slowness in AL.

At one time this was certainly true, but since then (before the '88-'89 run) the efficiency of locking has been improved by using the routines described above. It could be true that the locking functions still use too much of the available resources, but I don't think anyone has ever shown this to be true.

3 *Macro* Alarms & Limits

I now move on to the *macro* part of AL, which means describing the different processes and how they interact. I will begin this section by considering what happens when AL is started up. Each process will be dealt with as it begins executing and the relationships among them will (I hope) become clear. Alarms & Limits is started by typing the command `GO_ALARMS` on the LIMITS account on the B0SCCC VAX. (Warning: The account name, VAX node, etc. are subject to change. This is how things were during the '88–'89 run.) A command file LIMITS$COMMANDS:STARTTEST.COM is submitted to the batch system. This command file first spawns as a subprocess LAM_MONITOR.

3.1 LAM_MONITOR

The first thing LAM_MONITOR does is call the routine that initializes the CV array. The routine `Create_Global_Section` in CREATE_GLOBAL.FOR establishes the CV global section. The LIMITS.DAT file is read in and the CV_ARR is initialized. The RESOURCE.DAT file is also read in and the logical name assignments are made. A routine to initialize the global section locking is called. LAM_MONITOR then goes on to call the routines necessary to initialize itself. Recall that the function of LAM_MONITOR is to receive lam generated messages from CAMAC, specifically from the LSI's. To do this LAM_MONITOR employs the EVENT_HANDLER software developed in the Fermilab Computing Department. The details are beyond the scope of this note, but a lam in CAMAC will set an event flag on the VAX, and a VAX process can determine which crate generated the lam. Typically, LAM_MONITOR is in a hibernation state. When a lam is generated by one of the AL CAMAC crates (LAM_MONITOR can request that it is notified only when lams from specified crates are generated), LAM_MONITOR asynchronously branches to its interrupt routine. This interrupt routine checks which event flag is set, and thereby identifies the CAMAC crate that produced the lam. The routine then reads the lam register from the crate in question to determine which slot within the crate generated the lam. If the slot corresponds to an LSI, the lam monitor reads from the LSI a single instruction word. It then stores this word in a buffer and exits the interrupt routine. Once out of the interrupt routine LAM_MONITOR begins to service the requests that are in the buffer. When all of these are completed, the program goes into a hibernation state, awaiting the next lam. More details on specific LSI instructions are provided below. After the LAM_MONITOR process is spawned and initialized, the process VAX_SCANNER is spawned.

3.2 VAX_SCANNER

After LAM_MONITOR has been spawned and is in a hibernation state, the process called VAX_SCANNER is started. This process is primarily responsible for collecting data, both via the LSI's and from direct CAMAC reads, as discussed above. In the initialization stage, VAX_SCANNER downloads the LSI executable programs into the LSI's themselves (the routine is `init_lsi_systems`). It also maps into the CV global section, as well as other global sections that will be described later. Also in the initialization stage VAX_SCANNER sets up a list of critical channels that must be read out more often than the bulk of channels (see the subroutine `read_values`).

After these tasks have been completed, VAX_SCANNER begins its polling of the CAMAC crates. It is essentially in an infinite loop, performing a partial readout for each execution of the loop. Every n^{th} loop, where n is set at the beginning of the program, VAX_SCANNER reads out all the CAMAC crates. It first uploads (`get_curvals_table`) and stores (`put_curvals_database`) the data that has been collected by the LSI's. It then goes on to read out the CAMAC channels that are not associated with LSI's (`read_non_lsi_crates`). It also calls the routine `read_ctc` to retrieve the CTC voltage and current values from the IBM PC that controls the CTC.

VAX_SCANNER also is responsible for re-filling the alarms global section. I will describe in detail the alarms global section below. In theory, the alarms global section is filled by asynchronous messages coming from the LSI's or from VAX_SCANNER. However, it can happen that alarm messages from the LSI's never make it to the VAX. For this reason every 12 loops or so VAX_SCANNER will update the alarms global section by checking the alarm bit of every channel in the system. This is time consuming and inefficient, but necessary because the asynchronous alarm reporting is not 100% efficient.

Another feature of VAX_SCANNER is the `fast_scan` mode of operation. This mode enables VAX_SCANNER to read out a very small number of channels at a very high rate ($\sim$ 15 seconds). It is used in conjunction with the `alarms_display` feature of the MONITOR program, which makes a bar graph display depicting the status of the detector's high voltage systems. VAX_SCANNER goes into `fast_scan` mode when a flag is set true by a user using the MENU program. This flag is stored in the display global section, which is used by these three programs.

VAX_SCANNER runs in an infinite loop—the only way it stops is by crashing or by detecting a global failure of the CAMAC system. It does this by making sure each crate is responding to a simple CAMAC query each time through the loop. If CAMAC is down, VAX_SCANNER will terminate. The MONITOR program has a routine which checks to see that all the alarms processes are alive and well, and if it finds that VAX_SCANNER has stopped, it will display an appropriate message on the alarms display terminal in the B0 control room.

3.3 MONITOR

The MONITOR process makes sure that alarm and other status information is displayed in the control room. It is run as a batch job, but it attaches itself to a color display terminal in the control room, where it displays alarms and high voltage information.

Systems

I must now introduce the concept of systems. As discussed above, individual channels are grouped together into logical units, identified by logical names. In an analogous way, logical names are grouped together into systems, where each system is denoted by a three letter name, e.g. CEM for the central electromagnetic calorimeter. The file ALARMS$SOURCE:WEIGHTS.DAT contains a list of the systems and their associated logical names. The subroutine `init_monitor2` reads in this file and initializes the system array (`sys_arr`) which contains a list of the systems and pointers that point to another array (`logname_ptr_arr`), which contains a list of pointers to the array containing the logical names (`node_arr`). All these arrays of pointers make it easier and faster to do the vast amount of cross referencing necessary to manipulate these channels, logical names, and systems. They are defined in the file ALARM_COMMON.INC and appear in the alarm global section, which is created by the routine `create_alarm_global_section`.

The status of each system is displayed on one line of the alarms display terminal. For each system, four quantities are displayed:

- the number of "normal" alarms, i.e. the number of out of tolerance channels (excluding "severe" alarms, to be defined below);
- the number of "severe" alarms;
- the number of disabled channels;
- and the total number of channels being monitored.

Alarm types

Internally AL has three types of alarms: normal, severe, and sigma alarms. Certain channels are considered to be critical and if they are out of tolerance the data taking must be halted. These are labeled severe alarms. A sigma alarm channel is one for which the alarm condition becomes severe if the channel is very far out of tolerance. The definition of "very far" is contained in the "sigma value" associated with the channel. This is an integer between 1 and 16, which multiplies the nominal tolerance. That is, if $value > nominal + sigma * (max - nominal)$ (with the analogous definition for channels under nominal) then the alarm becomes severe. Channels that are not designated severe or sigma alarm channels are called normal alarm channels. These assignments are made in the routine `fill_severe_ptr` (called by `init_monitor2`) during initialization. The `alarm_weight` field

of the CV record contains a flag specifying the type of alarm (and the value of sigma for sigma alarm channels).

As mentioned above, alarm reporting can be turned off for specified channels. The number of channels that have been disabled in this way is also listed on the display terminal. These channels continue to be read out, but alarm conditions are not reported for them. This feature is useful when debugging a system, or bringing up AL after a long down period. The total number of channels in a system is displayed to let the user know that something is indeed being monitored. If a CAMAC crate dies, for example, the total number of channels in a system will decrease (and the numeral will change color to red). If there are no channels being monitored for a system, a message to that effect will be displayed.

Alarm reporting

The process MONITOR is driven by the list of systems. For each system it determines the number of alarms, both normal and severe, and writes the results using DI3000 calls to the alarms display terminal in the control room. The routine `get_alarms2` is used to retrieve the alarm information. This routine does not check each channel of the entire system to see if its alarm bit is set. This is a possible way of locating all alarms, but is too time consuming. Instead, it simply reads through a common block (`alarm_common`) that contains a list of the current alarm channels. The array `alarm_arr` contains the camac address, an alarm code (=1, not used for anything), and the time of the alarm.[3] The `alarm_arr` is maintained by the routine `send_alarm`. It is called via the routine `log_alarm` (called by LAM_MONITOR) or the routine `check` (called by VAX_SCANNER). A short description of alarm reporting is in order.

An alarm is detected either by an LSI in a CAMAC crate or by the process VAX_SCANNER when it is performing a direct CAMAC read. In either case the alarm is reported as it is found. Likewise, if a channel that is out of tolerance comes back within tolerance, that is also reported so the `alarm_arr` can be brought up to date. If it is an LSI that detects the alarm, it interrupts the VAX to send it the alarm information (primarily the CAMAC address, and whether the channel has gone out of tolerance or come back within tolerance). It is the LAM_MONITOR process that receives this information. It then calls the routine `log_alarm`, which updates the alarm common block. The process VAX_SCANNER can also detect an alarm when performing CAMAC reads in the routine `read_values`. The routine `check` determines if the the alarm common needs updating. Furthermore, both `check` and `log_alarm` call the routine `alalin_send_intmsg`, which sends the alarm information to the ALALIN server for recording in the CDF database.

As mentioned above, the routine `get_alarms2` is called by MONITOR to determine how many channels in each system are out of tolerance. The routine simply reads through the `alarm_arr`, using the cross-referencing routines to translate the CAMAC id for each channel into the appropriate system name. If a severe alarm is detected in any system, a

[3] The time is in seconds from DEC's t=0 point. See the file TIME.FOR for routines to manipulate time values.

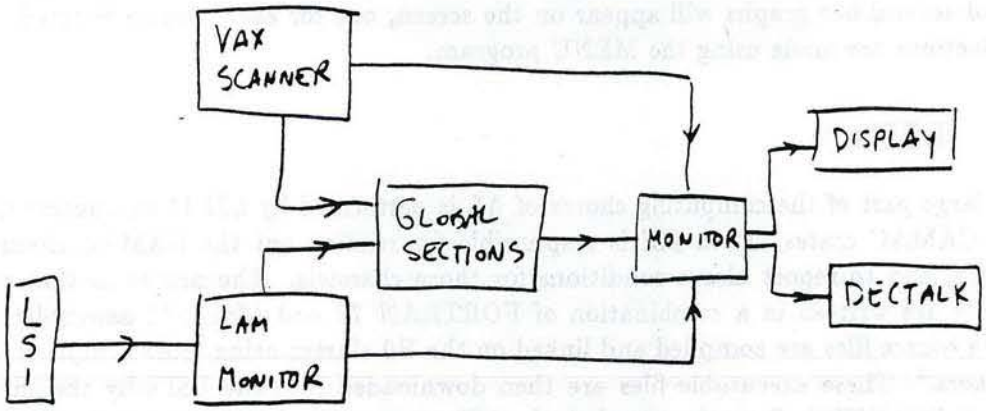

Figure 2: Schematic representation of alarm reporting in Alarms & Limits.

message is sent to DECTALK, which audibly notifies the shift workers of the severe alarm condition. A schematic of the flow of information during alarm reporting is given in Fig. 2.

Graphics in MONITOR

MONITOR has the capability of presenting a graphical representation of the state of the detector's high voltage systems. During normal operation, the display terminal shows a bar graph in the lower left corner. Each bar represents a different high voltage system. The height of the bars range from 0% to 100% of nominal full scale high voltage. Normally, several (sometimes many) channels comprise a high voltage system, and the bar in the graph is for the channel with the lowest percentage of full scale. The bars are color coded so that systems at less than 30% of nominal have red bars, between 30% and 95% yellow bars, and greater than 95% green bars. The refresh rate of the graphics in normal mode is rather slow, up to 4 minutes. For more up to date information, `fast_scan` mode is used.

`Fast_scan` mode is selected from the MENU program. It causes both VAX_SCANNER and MONITOR to enter into this mode of operation. In this mode, the alarms information is erased from the display terminal and the bar graph occupies the full screen. The plot is updated every 15 seconds or so. This mode is useful for monitoring the status of the

high voltage systems when bringing all the voltages up at the start of a run, for example. Furthermore, in this mode the user can select a particular system, and the bar graph will show the individual channel voltages for that system. Several systems can be selected and several bar graphs will appear on the screen, one for each system selected. All these selections are made using the MENU program.

4 LSI's

A large part of the computing chores of AL is performed by LSI-11 computers that reside in CAMAC crates. Each LSI is responsible for reading out the CAMAC channels in its crate, and to report alarm conditions for those channels. The programs that run in the LSI's are written in a combination of FORTRAN 77 and Macro-11 assembler language. The source files are compiled and linked on the B0 cluster using cross compilers, and cross linkers.[4] These executable files are then downloaded into the LSI's by the the program VAX_SCANNER. Once downloaded, the LSI programs run in an infinite loop, reading out voltage values, and reporting alarms when they are detected. Periodically, VAX_SCANNER interrupts the LSI to upload the collected data. The next subsections describe the organization of the LSI code and the protocol used for VAX - LSI communications.

4.1 LSI code

It is impossible in a note of this kind to fully document the code, but I will give here a general outline of the programs that run on LSI's. For more details, you will have to look at the code itself. The main program is a macro (assembler) file, but it simply calls the routine `lsi` (`lsi.ftn`) which is the main routine. This routine executes a list of instructions that have been downloaded from the VAX along with the executable. This list, called the instruction table, is contained on disk in files ALARMS$DATA:INSTBL_⟨*cr*⟩, where *cr* is the crate address for the particular camac crate. The list contains a number of records, each record containing an OPCODE and several other fields. These other fields typically specify a CAMAC address or data, or perhaps they are unused. The routine `lsi` contains a big IF block, which calls the routine appropriate to the current opcode, passing to the subroutine any necessary data. There are many opcodes defined, but only a handful are used regularly. I list a few of them here:

- OPRLMT=11. This specifies that the program should loop over all channels in the limit table, reading out each one as it goes. The routine is `sbrlmt`. Each channel has a device code, which tells the program what CAMAC module it corresponds to. The routine `sbmnch`, called from `sbrlmt`, calls a routine to read out the channel based on its device code.

- OPCHCH=15. This changes the value of one or more fields of a record. This opcode is executed when a download of a single record is performed (see next section).

[4] Complete instructions for compiling and linking all AL programs is given below in the Appendix.

- OPHALT=9. This puts the LSI in a pause state. Usually used to allow CAMAC operations to be performed directly from the VAX.
- OPSTRT=24. This restarts an LSI that as been paused with the OPHALT opcode.

In fact, these last three opcodes are generally issued via an input table, as opposed to an instruction table. An input table contains a list of instructions like an instruction table, but it has a higher priority and it is only executed once. Thus, it is often used to perform one or two commands one time. There is more information on input tables in the next section.

When all the instructions in the list have been processed, the routine `lsi` goes back to the top of the instruction list and executes again, in an infinite loop. Thus, the LSI's are continuously taking data and checking for alarms.

4.2 Vax - LSI Communications

Generally, VAX - LSI communications are carried out via a pair of CAMAC registers in the LSI. When written to, these registers generate interrupts either on the VAX or the LSI. When the vax wants to make a request of an LSI, it writes a specific word in the LSI's host request register (hrr). The LSI immediately branches to its interrupt routine whose first action is the read the word in the hrr, which is typically interpreted as an instruction. In general, the LSI is asked to prepare itself for a data transfer either from or to the vax. The code word written by the vax specifies the length of the transfer and the type of data (i.e. the LSI address at which the transfer will take place). The LSI must set up a number of registers in the Dataway Communications Link (one of the CAMAC modules that make up the LSI) and then notify the vax that it may issue the camac calls to transfer the data. The LSI notifies the vax by writing a word to its remote request register (rrr), which in turn generates an interrupt on the VAX. Likewise, the vax's first task is to read this word (via CAMAC) and to decode its contents. When it sees that the LSI is read for the data transfer it issues the appropriate camac calls to effect the transfer.

It is instructive to consider a specific example. The process VAX_SCANNER running on the vax does periodic uploads of data from the LSI's. The sequence of handshaking is as follows:

1. VAX_SCANNER writes a code word to the LSI's hrr. The code word specifies that the limit table (main data array) is to be uploaded to the vax. It then pauses, waiting for the LSI to respond. (see `get_curvals_database` in `vax_extras.for` and `sendcode` in `lsi_io.for`.)
2. When the hrr is written by the vax, the LSI asynchronously branches to its interrupt routine. It sets up the appropriate registers for a data transfer. These are the word-count register (length of transfer), memory-address register (address of first word of array to be uploaded), and various bits in the control status register (e.g. to specify the direction of the transfer, in this case from the LSI to the vax). (see `vaxint.mac`, which contains the LSI interrupt routine).

3. When the LSI is done with the setup, it writes its own code word (LSI ready) to the remote request register (rrr). This generates a vax interrupt (by setting a lam). The process LAM_MONITOR intercepts the interrupt and reads the LSI's rrr. When it sees the LSI ready message, it sets a flag in the `lsi_common` common block. This common block is in a global section, so it is available to all vax processes. (see `lam_service, lam_int.for` and `service_lsi, lam_initialize.for`.)

4. VAX_SCANNER sees the `lsi_resp_status_flag` go true and proceeds. It issues the camac commands that transfer data from the LSI. The camac routine returns the number of words successfully transfered and if it agrees with the number of words requested, the upload has been successful. (see `get_curvals_database` in `vax_extras.for` and `sendcode` in `lsi_io.for`.)

4.2.1 Types of VAX - LSI communications

There are a number of functions that require communication between the vax and the LSI's. The following functions are initiated by the vax:

- Downloading the LSI program and data (`start_single_lsi.for`). This proceeds in four steps:

  1. download of the executable file (e.g. LSCAN.EXE) into the LSI.
  2. download of the instruction table: this is a list of instructions which the LSI will perform in an infinite loop.
  3. download of the limit table: this is the main data array, the portion of the CV global section that corresponds to the particular crate in question.
  4. download of the start command: initially, the LSI is just polling a flag waiting for it to turn true. The start command sets that flag, whereupon the LSI begins executing the instruction table.

- Download an input table:

  An input table contains a list of commands for the LSI to execute. It is similar in form to the instruction table, but its priority is higher. Thus, if you want the LSI to perform one or several special commands one time, you send it an input table specifying those commands. After finishing the commands of the input table, the LSI returns to executing the instruction table.

  Input tables are used almost exclusively to put the LSI in a pause state or to re-start it after it has been in a pause state. Pausing an LSI is necessary whenever the vax needs to perform camac operations in the crate (excluding camac operations to/from the LSI). In particular, the vax executes most of the commands pertaining to the high voltage control, and before it can do that it must pause the LSI in the relevant camac crate. When the vax is finished sending camac commands to the crate, it must re-start the LSI. When an LSI is paused, it is just in a polling loop waiting for a flag

to be set. The full set of handshaking steps as described above for uploading the limit table is also necessary for pausing or re-starting an LSI (see `lsi_control.for`).

- Upload the limit table:

  Transfer the main data array for a particular crate from the LSI to the vax. This array contains the most recently read values of the channels. Uploading the limit table is the VAX - LSI handshaking sequence described above.

- Download a single record (channel):

  This is like downloading the limit table, but only one record (i.e. the full information for a single channel). Usually this is done from the program MENU when a channel's data is changed interactively (see `store_input_buffer, input_buffer.for`).

Some transfer functions are initiated by the LSI:

- Download GHV calibration array:

  This is a special function, pertaining only to the gas calorimetry programs PSCAN and FSCAN. The current channel readout calibration data are downloaded at the LSI's request during initialization.

- Upload alarm message:

  This is done when the LSI detects a new alarm. An alarm is defined as a channel whose value is outside of the specified limits. The message is not repeated each time the channel is read, only the first time the alarm is detected. Likewise, if a channel that was out of tolerance comes back within tolerance, a message is sent to the vax. This way the vax can keep a running list of all channels that are out of tolerance.

- Upload error array:

  Uploads an array of error codes to the vax. This was developed as part of the error handling for the gas calorimetry high voltage code, but it is quite a general routine that can upload an array of any size to the vax (see `errsnd` in `lmac.mac`).

5 High Voltage Control

Control of CDF's high voltage systems is achieved by performing CAMAC operations from the VAX. The LSI's are bypassed in these operations. For the VAX to access crates that have an LSI in them, it must pause the LSI before performing the CAMAC operations. The VAX may then issue CAMAC reads and writes while the LSI is in a paused state. After finishing, the VAX must restart the LSI so it can resume its readout of the crate.

The user controls the high voltage via the user interface program MENU. The user selects the system he/she wants to control, and then the function he/she wants to perform. All the high voltage systems have at least three functions:

- ON - sets high voltage to 100% of nominal value.
- STANDBY - sets high voltage $\sim$ 30% of its nominal value. The actual percentage varies among the systems.
- OFF - sets the high voltage to zero.

Many systems have other functions that are relevant to that system only. For example, for the forward calorimeter it is possible to manipulate the current relay states from the high voltage control menu.

When an HV control function is requested the MENU program spawns as a subprocess the program HV_MENU_SPAWNED to perform the task. Up to four tasks can be performed simultaneously, which means four different HV_MENU_SPAWNED subprocesses will be running under a single MENU program. The output from the subprocesses is directed to a window on the MENU terminal, so the user can verify the success of the requested task. This technique of spawning the HV tasks allows the user to continue using MENU while the task is being executed. This is a useful feature since most of these high voltage control functions take several minutes to execute.

A description of how to raise and lower high voltages using MENU is available in the B0 control room.

6 Systems monitored and controlled

In this section I will give a list of the systems monitored and/or controlled by AL. For each system I will indicate briefly the relevant hardware and software. First, the systems that are only monitored:

- Fastbus Monitoring (FBM) - The FASTBUS racks in B0 are monitored for voltage, temperature, humidity, etc. Signals are produced by an "AL box" in each FASTBUS rack and fed into CAMAC ADC's called SAM's. These SAM's are read out either by an LSI or by the VAX directly. There are about 1000 to 2000 channels in this system. Bill Wickenberg's group maintains the hardware.
- Central/Endwall Calorimetry (CEM,CHA,ENW) - These are phototubes that are read out via the Pisa Phototube High Voltage system. An LSI reads these channels out. The LSI code was written by the Pisa group, is quite complicated, and unfortunately not very well documented. It works, but there is not much expertise still on CDF for this software. In general, the phototube voltages are typically left at full value, so control of the phototube voltages are not effected via software. There are about 2500 phototubes in these three systems.
- Forward Silicon (SIF) - These channels are also read out by the Pisa system, although they are not phototubes. Stefano Belforte is the expert for this system.
- Solenoid (SOL) - Solenoid currents, read out from a SAM module.

- ASD low voltages, VTPC temperatures - The ASD crates are front end crates for the tracking systems. The signals are read out with SAM's. Morris Binkley is the expert.

- Gas Energy Scale, Gas Gain - These are gas gain results that are shipped to AL, so alarms may be detected. These channels are all in software, there is no CAMAC hardware that corresponds to these channels, although they are assigned CAMAC addresses. Tom Phillips is the expert for this system.

The following high voltage systems are both monitored and controlled by AL:

- Gas Calorimetry (PEM, PHA, FEM, FHA) - These detectors use the LeCroy 4032 high voltage supplies. The 4032 is read out using LSI's in two crates (one each for plug and forward). Originally, the current on each chamber was read out (about 1000 channels), but this was discontinued after it was determined that it caused noise in the calorimeters. There are about 50 high voltage channels total for these systems. Carl Haber wrote the LSI code, presumably the gas calorimetry groups have appointed a new expert.

- Forward Muon (FMU) - This system has two parts: i) phototubes read out by the Pisa system and not controlled; ii) chambers attached to Droege high voltage supplies that are read out by the VAX via the SAM ADC modules. One of several systems on the Droege supplies that are controlled using the routines in `droege.for`, authored by Nigel Lockyer and John Walsh. Karen Byrum is the hardware expert.

- CDT/CMU (CDT, CMU) - Both systems on the LeCroy 4032's. Originally used the same LSI programs as the gas calorimetry system, but Phil Schlabach has rewritten this code, improving the readout speed significantly. This was quite necessary because these systems together have over 100 channels. Phil is preparing a document to explain the features of this code.

- Central Strips (CES) - On Droege supplies, read out by SAM's and controlled by `droege.for` as the FMU above. Bob Wagner (ANL) is the hardware expert.

- VTPC (VTP) - Same as for CES. Morris Binkley is the hardware expert.

- FTC (FTC) - System removed in middle of last run. HV is the same as for CES. M. Atac was the expert.

- CTC (CTC) - This is a special system. It has its own special supplies and is read out by an IBM PC. There is an interface to the VAX, which provides the PC information to the AL global section. Aseet Mukherjee wrote the PC code and the VAX interface and is the CTC HV expert. This VAX code is kept in [LIMITS.CTC], not in ALARMS$SOURCE.

A Building Alarms Programs

A.1 Linking executable programs

There are programs that run on the VAX and others that run on LSI's. For VAX programs use the LIMITS account to link, using any B0 cluster VAX (except B0SCCC, which is busy running AL). These are the VAX programs and how to link them:

- VAX_TEST.EXE - process name is VAX_SCANNER. Command file to link is ALARMS$WORK:VAX_TEST.COM. The executable is created in ALARMS$PROGRAMS. This exe runs as a subprocess, created by the command file STARTTEST.COM.
- LAM_TEST.EXE - process name is LAM_MONITOR. Command file to link is ALARMS$WORK:LAM_TEST.COM. The executable is created in ALARMS$PROGRAMS. This exe runs as a subprocess, created by the command file STARTTEST.COM.
- MONITOR.EXE - process name is "Alarms Monitor ". Command file to link is ALARMS$WORK:MONITOR.LINK. The executable is created in ALARMS$PROGRAMS. This exe runs as a batch job, submitted by the command file STARTTEST.COM.
- HISTORY.EXE - process name is "Alarms History". Command file to link is ALARMS$WORK:HISTORY.LINK. The executable is created in ALARMS$PROGRAMS. This exe runs as a batch job, submitted by the command file STARTTEST.COM. There is no other information on HISTORY in this document. It is capable of storing AL information permanently and provides plotting functions, etc. See the file ALARMS$DOC:MENU.DOC for more information.
- MENU.EXE - This program is run interactively. Command file to link is ALARMS$WORK:MENU.LINK. The executable is created in ALARMS$PROGRAMS:MENU_NEW.EXE. This exec must be renamed MENU.EXE for it to become the default version. See the file ALARMS$DOC:MENU.DOC for more information.
- HV_MENU_SPAWNED.EXE - This program is spawned as a subprocess of MENU.EXE. Command file to link is ALARMS$WORK:HV_MENU_SPAWNED.LINK. The executable is created in ALARMS$PROGRAMS.

For LSI programs use account ALARMS on B0HOST. There are 4 different programs that run in the various LSI's:

- LSCAN.EXE - Contains the code to readout the Pisa Phototube High Voltage systems. Also used for reading the SAM ADC modules (Fastbus Monitoring and other miscellaneous systems). Linked interactively in ONLINE$UTILITY:[ALARMS.LSI]. The command file is LSCAN.COM. The exe file must be copied into ALARMS$PROGRAMS.

- FSCAN.EXE - Contains the code to readout the LeCroy 4032 HV systems. Used by the forward calorimeter. Linked interactively in ONLINE$UTILITY:[ALARMS.LSI.TEST]. The command file is FSCAN.COM. The exe file must be copied into ALARMS$PROGRAMS.

- PSCAN.EXE - Contains the code to readout the LeCroy 4032 HV systems. Used by the plug calorimeter. Linked interactively in ONLINE$UTILITY:[ALARMS.LSI.TEST]. The command file is PSCAN.COM. The exe file must be copied into ALARMS$PROGRAMS. At the time of this writing, the programs PSCAN and FSCAN are in fact identical.

- CSCAN.EXE - Same as above, but used by CMU/CDT systems. Substantially revised by Phil Schlabach, U of I. I believe the linking file is CSCAN.COM in ONLINE$UTILITY:[ALARMS.LSI.WORK], but this must be confirmed by Dr. Schlabach. Exe must be copied to ALARMS$PROGRAMS.

A.2 Compiling and updating libraries

The VAX source code is located in ALARMS$SOURCE on the ONLINE$UTILITY disk on the B0 VAX cluster. A CMS library does exist (ALARMS$CMS), but it has not been kept up to date.

The AL VAX library is ALARMS$DEBUGLIB:ALARMSLIB.OLB. All the routines are compiled with the /debug/nooptimize qualifiers. There is no non-debug library.

The command file CLIB.COM in ALARMS$WORK compiles and replaces in the alarms library. It takes the filename (no extension) as its only parameter. It looks in ALARMS$SOURCE for the source code. After compiling and replacing it deletes the object file. CLIB should therefore not be used compiling the files containing main programs. To compile such files, set default to ALARMS$WORK and type FORD ALARMS$SOURCE:⟨filename⟩.

The LSI source code resides in 3 places, see the linking instructions above for the directory names. Be sure to use the ALARMS account on B0HOST for linking or compiling LSI code.

LSCAN: there is no library for user written code. Use the command file F7.COM to compile the FORTRAN source code. This source code must be written in FORTRAN 77. F7.COM invokes the FORTRAN 77 cross compiler. You must supply the filename as the first parameter. You may not supply the extension, which must be .FTN. Macro-11 files have the extension .MAC and are assembled using the command MAC/RSX ⟨filename⟩.

FSCAN or PSCAN: Use the command file F7.COM to compile the FORTRAN source code. This source code must be written in FORTRAN 77. F7.COM invokes the FORTRAN 77 cross compiler. You must supply the filename as the first parameter. You may not supply the extension, which must be .FTN. Macro-11 files have the extension .MAC and are assembled using the command MAC/RSX ⟨filename⟩. There is a library for FORTRAN 77 code only: ONLINE$UTILITY:[ALARMS.LSI.TEST]ALRM77. To replace or insert into this library use this command: LIB/RSX ALRM77 ⟨filename⟩.