DOE/ER/03130-51C
COO-3130TC-51

DOE/ER/03130--51

DE90 013324

Level-2 System,

Programmers-Users Manual.

J. S. Hoftun
Brown University
15 June 1990

Introduction.

This manual describes the design and implementation of particular parts of the system for running high-level filter code in the Level-2 "farm" of MicroVAX computers. The various chapters detail the interfaces to this system both from the point of programming TOOLs for inclusion in the filter-code and from the point of writing VMS programs to perform control and/or monitoring of this system. A lot of detailed descriptions about how this system works are omitted.

It is separated into several chapters, each of which may have been released before as separate notes. The information in this manual supercedes ALL such previous notes.

DISCLAIMER

CONTENTS

CHAPTER 1

INCLUDING CODE IN THE LEVEL-2 FILTER SYSTEM.

This chapter describes the design of the interface between the framework which controls all the filters running in each Level-2 node and the individual TOOLs which make up the filters. Such a TOOL is seen as a "subroutine" which goes through ONE particular algorithm and makes a decision on whether the event should be passed or not. It may of course call as many other routines as needed internally. A TOOL should be developed and tested in the offline environment before being used in the Level-2 filter.

1.1 Rules For FORTRAN Programs.

While a lot has been said about the ease of transporting programs from the VAX/VMS environment to the VAXELN Level-2 environment, there are certain limits to this 'direct' transportability. Working with a network of computers instead of with one integrated CPU requires communication of one sort or another. Control of the running in the Level-2 is done via separate programs on the host VMS machine, this is the only mechanism for passing parameters to the running program. The following describes the rules which should be followed when writing any part of a FORTRAN program which may be used for running under VAXELN in a 'farm' of MicroVAXes.

In general all FORTRAN-77 code will run under VAXELN without conversion. But when doing I/O and calling system services one has to be careful.

Here are the specific rules:

1. NO system services (SYS$, LIB$, SMG$ etc.) may be used. (With a few exeptions.)

2. Implicit file opens (just a WRITE(n,*) without an OPEN(n) will NOT work).

3. File opens with logical name assignments for file-names will NOT work.

4. Keyed access files may NOT be used.

5. Do NOT assume infinite amount of memory, the program HAS to fit in the available physical memory space.

6. Do NOT assume that any library used under VAX/VMS may be used in VAXELN, in particular GENERAL, SRCP_UTIL and OFFLINE_UTIL. Each time such a routine is used, it must first be tested in a VAXELN environment. A list of "approved" routines will be kept.

To facilitate the inclusion of special code for the VAXELN case (use of the routines EOPEN, ECLOSE, EREAD and EWRITE for I/O etc.), the programmer should plan ALL the use of files and put the OPEN statements together in ONE place.

1.2 The Loading of 'Calibration-type' Constants.

These constants are thought to consist of a slowly changing set of numbers gotten from calibration runs, surveys of the detector etc. Since the amount of data is probably going to be very large, it would be unwise to reload them at every Begin-run if they don't change. A system where COOR is able to tell the Level-2 system when to load new constants is therefore implemented. Each DETECTOR part provides a routine which is only activated when new constants are needed. The call to these routines looks like:

```
CALL CENTRAL_CONSTANTS

CALL CALOR_CONSTANTS

CALL MUON_CONSTANTS
```

These routines read the constants from files (which actually may be connections to a server), using the special set of I/O routines described below. It is important that each DETECTOR (or part of a DETECTOR) uses a unique ZEBRA-structure to save and retrieve its own constants to avoid having the different DETECTORs step on each other.

A set of special routines is provided to do I/O in the Level-2 programs (but NOT in the main filter code, only at BEGIN-RUN). The main reason for needing these routines is the possible use

of servers and/or the fast multi-port channels for I/O in the future. It turns out to be impossible to make a connection to a server look like a standard file OPEN in FORTRAN. The four routines have the following calling sequences:

1.2.1 EOPEN:

```
CALL EOPEN(FILE_NAME,READ_WRITE,FORM,IRECL,IUNIT,ERROR)
```

where FILE_NAME is a character name for the file (logical name under VMS, and made into a 'real' file name in VAXELN), READ_WRITE is a flag indicating if the file is to be read or written (CHARACTER*1), FORM is file format ('U' for unformatted, 'F' for formatted), IRECL is the record length (if 0, parameter is ignored), IUNIT is a returned unit number to be used (saved in table and used by EREAD or EWRITE to check for opened files), and ERROR is a logical error flag (.TRUE. if an error occurred).

1.2.2 EREAD:

```
CALL EREAD(IUNIT,OUTSTRING,ERROR)
```

where IUNIT is unit number returned by a previous call to EOPEN, OUTSTRING is the bytes (returned as CHARACTER) actually read (an internal read or equivalences may be used to actually convert this to the appropriate form), and ERROR is a logical error flag.

1.2.3 EWRITE:

```
CALL EWRITE(IUNIT,OUTSTRING,ERROR)
```

where IUNIT is unit number returned by a previous call to EOPEN, OUTSTRING is the bytes (passed as CHARACTER) to write (an internal write or equivalences may be used to convert this from other types of variables before calling this routine), and ERROR is a logical error flag.

1.2.4 ECLOSE:

```
CALL ECLOSE(IUNIT,ERROR)
```

where IUNIT is unit number returned by a previous call to EOPEN and ERROR is a logical error flag.

1.2.5 GET_NODE_NAME:

To get the name of the node the Level-2 program is running on, use the following call:

```
CALL GET_NODE_NAME(NODE_NAME)
```

where NODE_NAME is a CHARACTER*6 variable which receives the name of the node the program is being run on.

1.3 Passing of 'Run-time' Parameters.

These constants are thought to consists of cuts, weights etc. which may well change from run to run. Each TOOL has to provide an entry-point (or separate subroutine) which is called at BEGIN-RUN time to do the loading of these parameters into a common block or ZEBRA-structure. The calling sequence for this entry point is:

```
CALL 'tool_name'_PARAMETERS(NUMB_OF_SETS)
```

where NUMB_OF_SETS is the number of parameter sets to read in, 0 means that NO reading should be done. This routine IS called also when NUMB_OF_SETS is zero in case any internal run-specific counters should be zeroed etc.

The routines EOPEN, EREAD and ECLOSE should be used to do the actual parameter input because of the same restriction on the use of a connection to a server as stated above. This is also true if the TOOL uses the Static Parameter Banks for their parameters, a special version of the lower level input routines will have to be provided for this case. If a TOOL is using a common block for storing these parameters, it must be unique to the TOOL. The name of this common block should be 'tool_name'_PARAMETER with an INC file to go with it.

Each parameter in the common block should be an array such that the TOOL may be used with different versions of the cuts. The structure treats each of these separate sets

semi-independently and it is up to the parameter-loading entry-point to keep track of how many sets have been received.

The structure has a set of "filter-scripts" defined by name. Each script is a list of TOOLs in the order they should be activated together with a parameter-set-number for each TOOL. COOR keeps track of these set numbers to make sure the number of sets read via 'tool_name'_PARAMETERS correspond to the set numbers used in activating a TOOL.

1.4 Calling Sequence of a TOOL.

Each TOOL has a very specific calling sequence in order to enable a simple 'dispatch' list of calls to build up filters and the whole Level-2 structure. Each TOOL must make a single decision on whether to pass the event or not. The structure takes care of setting the particular filter-bit to zero and abort the specific filter if the tool returns .FALSE.. The input arguments needed by the TOOL are the parameter-set-number as described above and a 32-bit word specifying which hardware trigger-bit actually set up the call to this filter. The hardware bit number is needed in case the TOOL wants to look at selective information from the Level-1 trigger block etc. The declaration of such a TOOL subroutine is:

```
  SUBROUTINE 'tool_name'(PARAM_SET_NUMB,HARDWARE_BIT,
 *            RETURN_FLAG,EXTRA_FLAG)
```

where PARAM_SET_NUMB is the number of the parameter set to use, HARDWARE_BIT is a 32-bit mask with the bit set for the Level-1 trigger which caused the filter to be activated, RETURN_FLAG is the actual result for the tool (.TRUE. if event passed the TOOL and should be handed to the next TOOL or filter) and EXTRA_FLAG is an unimplemented flag which should always be set to .FALSE..

1.5 Defining and Using TOOLs.

An 'editor' which adds a new TOOL call to the dispatch structure in the framework has been made. It is part of the L2STATE program which is used to manipulate specific Level-2 node information, Level-2 TYPE information in addition to the TOOLs as described here. For the TOOL definitions it uses a definition file which contains all the defined TOOLs. This same file is used by COOR (and the stand-alone program SUPCON) to set up the index for each TOOL as described above. L2STATE is able to add, delete and modify the information given for each one. It also edits the file of Level-2 TYPE definitions which uses the TOOL definitions to include specific TOOLs in each TYPE. Under the Level-2 Type Menu it has a command to produce two FORTRAN routines and possibly compile them, link the TYPE-specific Level-2 program and download the new program to the appropriate nodes.

The first routine made up by L2STATE is called by the framework once (and only once) at the start of the program and only makes up a table of addresses for each of the TOOLs included for the specific TYPE. It has the declaration:

```
SUBROUTINE FILTER_INIT
```

The second routine made up by L2STATE is called by the framework once at the beginning of a new run to call all the parameter routines. It has the declaration:

```
SUBROUTINE FILTER_PARAMETERS(RUN_NUMBER,NEWPAR)
```

where RUN_NUMBER is the number of the run being started and NEWPAR is the array (1 entry for each possible TOOL) of parameter sets to read.

The actual dispatching is done via a routine with the following declaration:

```
SUBROUTINE TOOL_DISPATCH(TOOL_ADDR,PARAM_SET_NUMB,
*           HARDWARE_BIT,RETURN_FLAG,EXTRA_FLAG)
```

where TOOL_ADDR is the address of the TOOL as found in the table set up by FILTER_INIT and the rest of the arguments as described above under "Calling sequence of a TOOL". The TOOL structure has to work with such an indexed list of TOOLs because it is very inefficient to use a search through a list via an IF-ELSEIF-ENDIF structure. COOR takes care of assigning the index of each TOOL in the definition of a filter-script.

1.6 Recording of Permanent Results from each TOOL.

The filter framework lifts the main FILT bank and puts in a copy of the 128-bit filter word, another 128-bit word describing which filters were actually tried, and other information. Each TOOL which wants to record any permanent information in this structure, should call a special utility routine to get the link to the next bank in a linear chain of result banks hanging from FILT. This utility routine will be provided by the filter framework (to be specified in detail later). In this bank the TOOL should put information about how far into the code it went before making it's decision etc. This information is important for monitoring purposes where the data will be looked at offline (possibly online in spy mode) to look for long term effects of particular parts of the algorithm.

1.7 Definitions and Uniqueness of TOOLs.

The way to make sure the TOOLs are uniquely defined, correlated with the information in the L2TYPE definitions and then included in the possible Level-2 system files is as follows:

1. Each possible TOOL is defined via the "TOOL Definitions" submenu in the L2STATE program. The parameter information for each one is also entered there. This information is stored in the L2TOOL.DAT file. A unique numbering of the TOOLs is set up this way, and these numbers are then used when building filters and must also be used when setting up the NEW_PARAMS argument to BEGIN_LEVEL2.

2. Individual Level-2 TYPEs are defined via a set of definitions stored in a file L2TYPE.DAT which is written and manipulated with the "Level-2 Type Definitions" submenu in the L2STATE program. The numbers of the TOOLs as described above are used to set up the call to 'tool_name'_PARAMETERS etc. as described in the Framework--TOOL interface chapter. The menu item "Make Filter Routines" makes two subroutines which are used when linking the L2_MAIN program for each TYPE.

1.8 Performance Monitoring of TOOLS.

Each TOOL has to be monitored closely as a run progresses. For error reporting, a utility routine which reports such errors to a central monitoring task as well as possibly the SORT_ALARM task, will be available. The calling sequence for this routine is the same as outlined in the manual for the ERROR_UTIL library and is:

```
CALL ERRMSG(IDSTRNG,SUBRID,VARSTRT,SEVERITY)
```

where IDSTRNG is an identifier for the TOOL (name), SUBRID is an identifier for the subroutine (name) calling ERRMSG, VARSTR is additional text comments on the error and SEVERITY is a CHARACTER*1 code for identifying the severity of the error and thereby where to record the error. The framework keeps track of how many times it calls a given TOOL and how often it passes an event. Further monitoring is possible via the recording of results in the FILT structure by looking at the data with an online (or offline) process.

1.9 Histogramming of Quantities from TOOLs.

The filter framework will use a set of utility routines to book histograms and fill them with information about the running of the filters and individual TOOLs. These histograms will be available as special event records at END_RUN. The routines used in the running system may be a modified version of the HBOOK set to gain speed.

A special version of the Level-2 program will be set up where the routines are actually the HBOOK ones such that the histograms may easily be looked at online etc. This program may or may not be running in a subset of the available Level-2 nodes at any given time.

1.10 Timing of a TOOL.

Most of the detailed timing of a TOOL should take place off-line, but some nodes will be set up with an accurate real-time clock and an interface routine which returns the timing information. The call to read the clock is simply:

```
NEWTICKS=READ_CLOCK()
```

This function will also have versions for VMS and VAXELN which use the standard time functions in the two cases. Under VMS this is actual CPU time used while in VAXELN as well as with the real-time clock it returns actual elapsed time. The number of ticks are always in units of microseconds. To get time differences one has to call the function before and after the section of code to be timed and do the difference explicitly. One will have to worry about the extra overhead of calling the READ_CLOCK function, especially if there are many of these timing calls in the code. The technique described in the FORTRAN manual for turning off/on code via "D-type lines" is recommended for including these calls to READ_CLOCK.

CHAPTER 2

TESTING TOOLS UNDER VAX/VMS.

This chapter describes the method for testing the TOOLs written according to the above specifications. The VAX/VMS test-bed is using the standard D0USER framework, and the user should be familiar with its operation. Documentation may be found in the file: D0$D0USER:D0USER_GUIDE.MEM. The PROGRAM_BUILDER is used to put the VMS_FILTER into the D0USER framework as described in the above manual and as described in D0$PROGRAM_BUILDER:USER_GUIDE.MEM. To make a standard VMS_FILTER version of D0USER, the user does NOT have to know how to run the PROGRAM_BUILDER.

The TOOL definitions described above is stored in a file as described above. For the test version this may be a different file than the standard Level-2 definitions (which are in D0DAQ$:L2TOOL.DAT). The logical name L2TOOL is used for this file, and there is a submenu in the USER DIALOG part of the VMS_FILTER_D0USER program which may manipulate the information in this file. If you do NOT make a logical definition and do NOT have a file L2TOOL.DAT in the directory you are working from, the item WRITE TOOL INFORMATION in the above submenu will make a new one for you. After the TOOL information has been changed, the command MAKE INIT ROUTINES must be issued to make up the two routines FILTER_INIT and FILTER_PARAMETERS using the current set of TOOL definitions. These routines are compiled by the procedure described below to make a new version of the VMS_FILTER_D0USER program.

Similarly, a file is looked for with the logical name RUN_FILTER. It is used to store all the filter-script definitions as well as the mapping between hardware trigger bits which may be present in the data to a set of filter-bits. The easiest way to find out which trigger bits are present is to run the program on a few events and look at the histogram TRIGGER BITS SET. The USER DIALOG submenu in D0USER manipulates all the information which is stored in RUN_FILTER. Notice that it is ONLY written out when the MAKE RUN_FILTER FILE command is issued. But if you make changes before starting to process

data, those changes WILL be used in the subsequent processing of data. This is a way to test out changes before permanently recording them in the RUN_FILTER file.

2.1 How to Include Your Own (or Library) Routines in the Link.

The link procedure (described below) looks for a standard option file, D0_FILTER.OPT (or DEB_D0_FILTER.OPT for a DEBUG version), in the directory being used for the link. In that file you should put the names of special .OBJ files and/or library specifications needed to complete the link.

2.2 The Procedure To Make VMS_FILTER_D0USER.

A command file (D0$VMS_FILTER:FILTER_MAKER) is used to make a new version of the program. It will first ask you if it should run the PROGRAM_BUILDER to make all the interface files needed. This should ONLY be neccessary ONCE. It will then check if the routines FILTER_INIT and FILTER_PARAMETERS are present, and if not, make new ones according to the file pointed to by L2TOOL. If no file is found for L2TOOL, an empty one will be created. No TOOLs will be included in the version of VMS_FILTER_D0USER linked after this. The procedure will also check for the OPT files described above and make empty ones if needed. It then finally links the program and runs the setup file such that the command D0USER (or DEB_D0USER) will start the program.

CHAPTER 3

LINKING MAIN PROGRAM FOR LEVEL-2.

This chapter describes the method for linking a new version of the main program for the Level-2 nodes. The EXE is tied to the TYPE of nodes as defined in the L2STATE program. (See separate chapter about the use of this program.) This program has to used to make two FORTRAN files for the selected TYPE. The link procedure compiles these files (to make sure they are available in the directory being used for the link).

3.1 How to Include Your Own (or Library) Routines in the Link.

As for the VMS case, the link procedure (described below) looks for a standard option file, D0_FILTER.OPT, in the directory being used for the link. In that file you should put the names of special .OBJ files and/or library specifications needed to complete the link.

3.2 The Link Procedure Itself.

The command file used to do the link is: D0$D0DAQ:L2_MAIN.LNK. The TYPE may be specified as either the first or the second parameter and a debug version may be made by specifying DEBUG as either the first or the second parameter. I.e. to make the standard EXE with debug one would type:

```
@D0$D0DAQ:L2_MAIN.LNK REGULAR DEBUG
```

If the TYPE is left off, it will be prompted for.

To download a newly linked program to the appropriate Level-2 nodes, you may either change the load program for a TYPE in the L2STATE program or, if it is already set to the correct one, use the "Force Load Flag" command in the same program. The actual load takes place either when the "Write Changes" command is executed or before exiting the program.

CHAPTER 4

ERROR HANDLING IN LEVEL-2 PROGRAMS.

There are two distinct ways of handling run-time errors in the filter code under VAXELN. The 'normal' way is to trap the error with the VAXELN debugger. The debugger (EDEBUG) does NOT have to be connected to the node at the time of the error for this trap to work. When connected with EDEBUG, you will see where the error occurred and what the error was. You may also examine variables etc. at that point. (This assumes that the source code was linked in DEBUG mode and that the program was linked with DEBUG.) For more information on EDEBUG see the VAXELN documentation.

The other way to trap the run-time errors is with the ERR_HANDLER module, which, when included in an ELN program, allows the programmer to have some control over the system's response to a run-time error. This is especially useful well debugged code when running in real life, where events may do unexpected things or probe little used areas of the code. Using this module, an event which produces an error can be ignored, and processing can skip to the next event. The ERR_HANDLER is a module which enables user defined action on the occurrance of a run-time error in a VAXELN program. If an error occurs, the ERR_HANDLER makes the program jump back to the next statement after the top level routine in the chain where the error occurred. The main program tests for this condition and reports the error if one is found.

This report is currently ONLY written to a 'data-base' file on the host VAX. The information in the report is translated as to which routine caused the error, and the time-stamp, run number, event number, node type, node name and program running in the node are also included. This translation uses a file ('l2type'_L2_MAIN.CODE which is produced from the MAP file when a Level-2 program is linked using the L2_MAIN.LNK command file. It is VERY important that that file is kept with the EXE as pointed to by the entry in L2TYPE.DAT. If the EXE is linked in one directory and subsequently moved, the CODE file MUST be moved as well. Otherwise the translated information will be

WRONG. A special interface program has been written to do searches in this data-base as described below.

4.1 L2ERR_READER Program.

The program accessed via the command L2ERR_READER interfaces to the data-base of recorded errors in the Level-2 system as described above.

The program first presents a menu of possible ways of searching the data-base. Each time you have performed a search, it will ask you if you want to perform a further search on the already selected entries.

The search items are:

1. TIME of error

   The actual time when the error occurred is recorded and the search will let you choose the START and END time of the search.

2. Find error in ROUTINE

   You may also see if a particular routine caused any errors. The program will ask for the name of the routine to look for

3. Find error in NODE_TYPE

   The nodes are separated into TYPES as described before. This search will let you enter the TYPE to look for in the data-base.

4. Specific NODE_NAME

   If you suspect errors in a specific node (for some unknown reason), this will let you search the data-base for a specific node-name.

5. RUN_NUMBER sequence

   If you only want to look at errors in a particular set of runs, this will allow such a search.

See the EXAMPLES at the end of this manual for the actual screen displays from this program.

CHAPTER 5

INTERFACE FOR CONTROL (COOR).

This chapter describes the design of the part of the Level-2 system used to interface between the framework which controls all the filters running in each Level-2 node and the host program COOR which coordinates all the data taking. The information COOR uses to set up the Level-2 system resides in several parameter files on the host disk. These "database" files are written to by a special interface "editor" program, L2STATE. This program may only write new information to these files when it is able to gain run-control. Normally the program COOR will have this control and will have to be asked to give it up before information can be written. The OPEN of the run-control file is recorded in a special file such that the program which needs control can be told who last opened it. See the routines RECORD_FILE, GET_FILE and REPORT_FAILURE described below. The method for asking COOR to give up control of these files is implemented as described below under REQUEST_LEVEL2, DEMAND_LEVEL2 and RETURN_LEVEL2.

COOR must in particular control the following aspects of the running in the Level-2 system:

5.1 The Loading of 'Calibration-type' Constants.

COOR has to decide when a new set of calibration constants are needed in the Level-2 nodes. This is of course only possible at Begin-Run time. In the call to BEGIN_LEVEL2 as described below, the argument CONSTANT_LOAD indicates which, if any, of the detector parts needs to load constants. Detector parts are in this context thought to be Central Detector, Calorimeter and Muon Detector. If a "finer" division of the major detector parts is needed to do the constant loading for only parts of one of these detectors, it should be done internally via the files COOR assigns for each major part.

5.2 Passing of 'Run-time' Parameters.

This is COOR's main task as far as the Level-2 is concerned. COOR must read the filter definition files, assign filter-bits to each individual filter, check that the assigned Level-2 TYPE has the TOOLs needed for the filter, set up the mapping of hardware bits into filter-bits and set up the filter-scripts which tells the Level-2 framework which TOOLs to call and which parameter set to use for each TOOL. The filter-bit mapping is passed via the FILTER_BIT_SET argument to BEGIN_LEVEL2 and the filter-scripts is passed via the TOOL_SCRIPT argument. For each of the filter-bits COOR also has to indicate whether it is to be run in 'force' mode or not. Force mode means that if the particular filter-bit is set, that filter always has to be run, even though another filter may already have passed the event. The indication of force mode is done via the FILTER_FORCE argument to BEGIN_LEVEL2. There is always a desire to be able to pass a certain fraction of the events unfiltered to have a sample to check out the efficiencies of the filters with. This is like a prescaling of the unfiltered events. The particular filter is run as normal on these events, but then a given fraction will be passed to the host anyway. This fraction is specified via the UN_FILTER_RATE argument to BEGIN_LEVEL2. The loading of parameters is likely to cause some amount of overhead, it is therefore useful to be able to only read in parameters where they actually have changed. This is done via the NEW_PARAMS argument to BEGIN_LEVEL2. It is an array where the number tells how many sets the TOOL should expect to read.

5.3 Calling Sequence of Routines

There are several routines written to interface with the Level-2 system from a host control program. They are used by COOR as well as by SUPCON, a standalone program for testing the Level-2 control. The routines described in this chapter all are found in the D0$L2CONTROL library area. The calling sequence for these routines are:

5.3.1 ATTACH_LEVEL2:

This routine is the initialization routine for the Level-2 interface which gets control over the run in the Level-2 system. It reads in the node information from the file L2STATE.DAT, Level-2 TYPE information from L2TYPE.DAT, hardware information from L2SUPER.DAT, as well as run information from RUNOLD.DAT and RUNNUM.DAT.

```
CALL ATTACH_LEVEL2(TOT_NUM,NAM_ARR,TYP_ARR,ATTOK)
```

where TOT_NUM is the total number of Level-2 nodes currently available (using 50 as max.), NAM_ARR is a CHARACTER*8 array of names for each one of the Level-2 nodes. TYP_ARR is a CHARACTER*16 array of TYPEs for each one of the nodes and ATTOK is a LOGICAL flag set to TRUE when successful. See GET_L2TYPE_INFO below for description on how to interpret the TYPEs. This information must be used when setting up the L2_MASK argument to BEGIN_LEVEL2.

5.3.2 DETACH_LEVEL2:

This routine is used to release the Level-2 system when run control is no longer needed and may have been requested by another program (COOR provides the routines REQUEST_LEVEL2, DEMAND_LEVEL2 and RETURN_LEVEL2 to communicate the desire to take over run control.)

```
CALL DETACH_LEVEL2
```

5.3.3 READ_NAMES:

This routine is used to get the name of the Supervisor in case run control is not needed. It reads in the node information from the file L2STATE.DAT.

```
CALL READ_NAMES(SUPER_NAME)
```

where SUPER_NAME is a CHARACTER*6 variable which receives the name of the current Supervisor as defined in L2STATE.DAT (via the L2STATE interface program).

5.3.4 GET_RUNNUMBER:

This routine is used to get the next run-number in a unique series. It should be called before any run setup is done. The run-number is recorded in a file, RUNNUM.DAT (with exclusive write-access by one program; this is the way run-control is assigned). In this file is also recorded all the information about the last run, like trigger bit assignment, filter-scripts etc. This makes it possible to start up a run with the same conditions as the previous one without using COOR. Even if the run is never actually started (if BEGIN_LEVEL2 is not called or a failure occurs in trying to start the run), this run-number will never occur again. GET_RUNNUM also records the old runnmuber together with its comment in the file RUNOLD.DAT.

```
CALL GET_RUNNUMBER(RUNNUM)
```

where RUNNUM is an integer return argument which receives the new run-number.

5.3.5 BEGIN_LEVEL2:

This routine is the actual setup and communication with the Level-2 system at begin-run. It has several arguments which are mentioned above for setting up the actual filtering in each Level-2 node.

```
      CALL BEGIN_LEVEL2(RUNNUM,COMNEW,L2_MASK,DP_MASK,
     *     CONSTANT_LOAD,FILTER_BIT_SET,TOOL_SCRIPT,FILTER_FORCE,
     *     UN_FILTER_RATE,NEW_PARAMS,FILTER_ORDER,FILTER_MAX,
     *     OK_FLAGS,ERROR)
```

where RUNNUM is the run-number returned from GET_RUNNUMBER (used to check that GET_RUNNUMBER was actually called before the call to BEGIN_LEVEL2), COMNEW is a comment (CHARACTER*64) for the run to be included in the run-number file, L2_MASK are masks of trigger bits by node (one 32-bit word per available Level-2 node, for a maximum of 50 currently; also used to mask out which nodes should actually take part in the run), DP_MASK are masks of dual-port channels by triggers (one 8-bit BYTE per trigger, 32 in all, TRGR always included if more than one channel in use), CONSTANT_LOAD is a LOGICAL*1 array with one entry for each detector part (3 currently) where a .TRUE. means that the particular detector part needs to load new constants for this run, FILTER_BIT_SET is a 4X32 integer array which forms the 128-bit filter-bit mask for each hardware trigger bit, TOOL_SCRIPT is a 2 X MAXTOOL X 128 integer array (MAXTOOL as set in L2_TYPE.DEF, currently 20) which gives the order of TOOLs and parameter-sets for each of the TOOLs for each filter-bit, FILTER_FORCE is a 128 LOGICAL*1 array where the entry is .TRUE. for each filter-bit which is to be run in force mode, UN_FILTER_RATE is a 128 integer array which gives the fraction (1:fraction) of events which is to be passed on to the host regardless of the result of the filter (0 means pass all events unfiltered), NEW_PARAMS is an MAXTOOL array of BYTEs which tells each TOOL how many parameter sets to read (0 means no new parameters to read, the order of the TOOLs in this array MUST be as set up in the file L2TOOL.DAT), FILTER_ORDER is a 128 BYTE array giving the actual order the filter_scripts should be tried, FILTER_MAX is an INTEGER*4 number giving the maximum number of entries used in the FILTER_ORDER array, OK_FLAGS is a LOGICAL*1 array with one entry for each available (using 50 as max.) Level-2 node where a .TRUE. is returned when the node successfully received the new run information (use NAM_ARR returned from ATTACH_LEVEL2 to turn these flags into actual node names) and ERROR is an integer error code (0 means everything OK, anything else is an error).

The current set of defined errors are: 2: Error in connecting or sending/receiving messages to/from Supervisor, 3: Run already in progress, 4: Run control not ON, 6: Bad match in Runnumber between this routine and GET_RUNNUMBER, 8: Asked for bank without matching datacable.

5.3.8 CONTINUE_LEVEL2:

This routine is called when the data-taking which was paused via a call to PAUSE_LEVEL2 is to be resumed again.

```
CALL CONTINUE_LEVEL2(OK_FLAGS,ERROR)
```

where OK_FLAGS is a LOGICAL*1 array with one entry for each available (using 50 as max.) Level-2 node where a .TRUE. is returned when the node successfully received the continue command (use NAM_ARR returned from ATTACH_LEVEL2 to turn these flags into actual node names; only nodes which had a non-zero L2_MASK in the last call to BEGIN_LEVEL2 will possibly have a return of .FALSE.) and ERROR is an integer error code (0 means everything OK, anything else is an error. The currently defined errors are: 2: Error in connecting or sending/receiving messages to/from Supervisor, 4: Run control not ON).

5.3.9 CHECK_LEVEL2:

This routine is called when the flush status of each node is needed outside of a call to END_LEVEL2 (or PAUSE_LEVEL2). This may happen when some of the nodes are to be switched to a different mode etc.

```
CALL CHECK_LEVEL2(FLUSH_FLAGS,ERROR)
```

where FLUSH_FLAGS is a LOGICAL*1 array with one entry for each available (using 50 as max.) Level-2 node indicating their flush status (use NAM_ARR returned from ATTACH_LEVEL2 to turn these flags into actual node names; only nodes which had a non-zero L2_MASK in the last call to BEGIN_LEVEL2 will possibly have a return of .FALSE.) and ERROR is an integer error code (0 means everything OK, anything else is an error. The currently defined errors are: 2: Error in connecting or sending/receiving messages to/from Supervisor, 4: Run control not ON).

5.3.6 END_LEVEL2:

This routine is called when a run is to be ended. It sends a message to the Supervisor which in turn tell all the nodes to end and report back whether they are "flushed" (done with the current event) or not. This routine may be called more than once to make sure the run really ended and that all the nodes are flushed (see CHECK_LEVEL2).

```
CALL END_LEVEL2(FLUSH_FLAGS,ERROR)
```

where FLUSH_FLAGS is a LOGICAL*1 array with one entry for each available (using 50 as max.) Level-2 node indicating their flush status (use NAM_ARR returned from ATTACH_LEVEL2 to turn these flags into actual node names; only nodes which had a non-zero L2_MASK in the last call to BEGIN_LEVEL2 will possibly have a return of .FALSE.) and ERROR is an integer error code (0 means everything OK, anything else is an error. The currently defined errors are: 2: Error in connecting or sending/receiving messages to/from Supervisor, 4: Run control not ON).

5.3.7 PAUSE_LEVEL2:

This routine is called when a pause in the data-taking is needed for some reason, but may be resumed later with the same parameters.

```
CALL PAUSE_LEVEL2(FLUSH_FLAGS,ERROR)
```

where FLUSH_FLAGS is a LOGICAL*1 array with one entry for each available (using 50 as max.) Level-2 node indicating their flush status (use NAM_ARR returned from ATTACH_LEVEL2 to turn these flags into actual node names; only nodes which had a non-zero L2_MASK in the last call to BEGIN_LEVEL2 will possibly have a return of .FALSE.) and ERROR is an integer error code (0 means everything OK, anything else is an error. The currently defined errors are: 2: Error in connecting or sending/receiving messages to/from Supervisor, 4: Run control not ON).

5.3.10 CHECK_RUN:

This function is used to check if a run is in progress or not. The first time it is called it actually asks the Supervisor for the run status, on any subsequent calls it uses a flag kept in a common block (and set/reset by BEGIN_LEVEL2, END_LEVEL2) to determine the status.

```
LOG_VAR=CHECK_RUN()
```

5.3.11 GET_L2TYPE_INFO:

This routine is used to get the information about a certain TYPE of Level-2 node. The TYPE definitions are stored in the file L2TYPE.DAT.

```
CALL GET_L2TYPE_INFO(TYPE_STR,NUMBER_TOOLS,TOOL_NUMBERS)
```

where TYPE_STR is an up to CHARACTER*16 string with the name of the TYPE to return information for, NUMBER_TOOLS is the number of tools loaded for this TYPE, TOOL_NUMBERS is an integer array of the number of the TOOLs as stored in L2TOOL.DAT.

5.3.12 GET_L2TOOL_INFO:

This routine is used to get the information about a certain TOOL in the Level-2 system. The TOOL definitions are stored in the file L2TOOL.DAT.

```
CALL GET_L2TOOL_INFO(NUMBER,TOOL_NAME,PARAM_COUNTS,
*     PARAM_NAMES,PARAM_TYPES)
```

where NUMBER is the number of the TOOL to get information for (as returned by GET_L2TYPE_INFO), TOOL_NAME is CHARACTER*32 string receiving the name of the TOOL, PARAM_COUNTS is the number of parameters needed for this TOOL, PARAM_NAMES is a CHARACTER*16 array of parameter names and PARAM_TYPES is a CHARACTER*1 array of parameter types.

5.3.13 REPORT_FAILURE:

This routine is used to report the failure in the OPEN on a file. If the file is actually locked by the run-control system, the message will tell who locked it, otherwise the FORTRAN error will be spelled out.

CALL REPORT_FAILURE(FILE_NAME) where FILE_NAME is a CHARACTER string indicating which file to report the failure of (same as used in OPEN statement).

5.4 Routines Supplied by COOR.

To be able to gain control over the running of the Level-2 system even when COOR is actively running, a set of routines is supplied in the D0$ONLINE_UTIL library to request and/or demand the release of the Level-2 system (i.e. the files as described above) and to return control back to COOR.

5.4.1 REQUEST_LEVEL2:

This routine is used to request the release of the Level-2 system, the request is approved or denied according to the current state of the run.

CALL REQUEST_LEVEL2(BROADC_STR,RESPONSE)

where BROADC_STR is a string to broadcast to all TAKERS and RESPONSE is a LOGICAL*1 flag which is TRUE when permission was given, FALSE when request was denied (or COOR not found).

5.4.2 DEMAND_LEVEL2:

This routine is used to demand the release of the Level-2 system. This is always granted, but the routine does NOT return until COOR has shut things down properly and given up control.

```
CALL DEMAND_LEVEL2(BROADC_STR,RESPONSE)
```

where BROADC_STR is a string to broadcast to all TAKERS and RESPONSE is a LOGICAL*1 flag which is TRUE when the demand was successful, FALSE when demand failed (could NOT find COOR etc.).

5.4.3 RETURN_LEVEL2:

This routine is used to return control to COOR when done with the Level-2 system.

```
CALL RETURN_LEVEL2(RESPONSE)
```

where RESPONSE is a LOGICAL*1 flag which is TRUE when successful in returning the Level-2, FALSE usually means that COOR is not available.

CHAPTER 6

LEVEL-2 DISPLAY/MONITORING INTERFACE

This chapter describes a set of routines which a "user" may call in a program which needs to display event-flow and/or wants to monitor actual throughput in the system. Other sub-systems, like Level-1 and COOR will provide similar routines. The routines described in this chapter all are found in the D0$L2CONTROL library area.

6.1 L2_SNAPSHOT:

For monitoring purposes, the actual exact count of various entities might be important (to look for "conservation of events"). Therefore a routine which gathers ALL the information as close to simultaneously as possible is provided. Its calling sequence is:

```
CALL L2_SNAPSHOT
```

After one call to L2_SNAPSHOT, the routines described below may be called as many times as needed, the information does NOT change between calls (unless L2_SNAPSHOT is called in the meantime). All counts are reset at begin-run time (when the GLOBAL run-number changes). If the user does NOT call L2_SNAPSHOT, it is called from the first routine in the set below, and then again everytime one of the routines is called.

6.2 L2_INFO_SUPER:

To get a report of the counters kept in the Supervisor, use the follwing call:

```
CALL L2_INFO_SUPER(TOT_EVENTS,TRIG_COUNT)
```

where TOT_EVENTS is the total number of events seen by the Level-2 Supervisor and TRIG_COUNT is an integer array (32 entries) which gives the count for the individual hardware trigger bits.

6.3 L2_INFO_NODES:

To get a report of the counters kept in in each node (by TYPE), use the follwing call:

```
CALL L2_INFO_NODES(TYPE,FILTER_COUNT,TOTAL_COUNT)
```

where TYPE is an input integer with the index of the Level-2 TYPE the information should be returned for (the TYPE is defined in the L2STATE program and the routine GET_TYPES should be used to access this information, a zero means return infor for ALL TYPEs), FILTER_COUNT is an integer array (3X128) entries which returns the total number of times a bit was set in an event, the total number of times its filter-script was actually activated and the number of times the event passed that filter-script, and TOTAL_COUNT is an integer array (3 entries) giving overall number of evnts IN to the Level-2 nodes of that TYPE, number of events OUT and total number of ERRORS detected (IN+ERRORS should for all TYPEs should equal the total number of events seen by the Supervisor, except when an event is sent to two different nodes at the same time).

6.4 GET_TYPES:

To get the current set of defined TYPEs, use the follwing call:

```
CALL GET_TYPES(NUM_TYPES,TYPE_NAMES)
```

where NUM_TYPES is the number of TYPEs currently defined and TYPE_NAMES is an array of CHARACTER*16 (at least NUM_TYPES entries) which returns the names of each individual TYPE.

CHAPTER 7

EXAMPLES

7.1 L2STATE Program

```
  Main menu

        1: Change State of Nodes          2: Hardware Setup
        3: Show Hardware Setup            4: Add a Node
        5: Delete a Node                  6: Force Load Flag
        7: Display Node List              8: Level_2 Type Definitio
        9: TOOL Definitions              10: Write Changes to Files
       11: Run Command File              12: Set up Command File
       13: Menu Control

Menu: Level_2 State Setup Program
Select: {command, #, HELP (#), MENU, EXIT} >
```