**SPEX (Spectrometer Executive):
General Structure***

J. T. Massimo, B. A. Nelson, I. J. Levinson**

Brown University High Energy Physics

Internal Report #123

August 1972

NOTICE
This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

MASTER

* Work supported in part by the U.S. Atomic Energy Commission under Contract No. AT(11-1)-3130 TB.

** of the Massachusetts Institute of Technology

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Elements of SPEX

Chapter	Contents
1	Introduction to SPEX
2	Core Management
3	Disk Management
4	SPEX Queues and the Node Pool
5	The System Task List and the Active Task List
6	Significant Events and the ATL Scan

Chapter 1

Introduction to SPEX

SPEX is a software system designed to support data acquisition and on-line analysis on a PDP-11 with at least 12K of core and a disk. One of its basic assumptions is that the machine size is less than what is needed for co-residence in core of all the software and buffers used in acquisition, display and analysis of the data of an experiment. The experimental software is therefore to be broken up into logical tasks which are stored on disk and brought into core only when they are needed. On this assumption, a primary function of the SPEX system is to simplify the construction of such tasks as swapping modules on the disk and the manipulation of these modules by program control. All other features of SPEX relate to making the most efficient use of core and CPU activity given the multi-task environment. These features include a task priority structure, priority-ordered queueing of I/O requests, dynamic allocation of task buffers, and support of the console teletype as a multi-task message device. In execution, the system simplifies the triggering and switching of CPU control between different tasks by the occurrence of "significant events", which include the completion of I/O activity, an external hardware signal, or the completion of a computation or logical sequence by an operating task.

A task requests executive services by issuing "directives". A directive is a unique processor TRAP instruction which is usually preceded by pushing addresses of one or more arguments on to the stack. SPEX returns information on the status of the request in the event variable argument.

It should be emphasized that the emphasis in SPEX is on a multi-task, but not a multi-user environment. Tasks executing under SPEX are to be constructed according to a single plan, a program for conducting a computer-assisted experiment. There is no algorithm within the system for resolving unintended conflicts for computer resources by any automatic optimization scheme. The program itself must plan for the resolution of these conflicts within the priority scheme of the system, otherwise degradation in the efficiency of the acquisition system will result. With this orientation, it differs from other "resource sharing" systems currently available from Digital Equipment Corporation for the PDP-11 series. On the other hand, in order to keep system space and time overhead to a minimum on a small computer, SPEX does not retain any part of the Disk Operating System (DOS) developed by DEC for the PDP-11 series during the execution phase, the actual data-acquisition and analysis period.

The relation between DOS and SPEX is strictly complementary. SPEX tasks are assembled and built as swap modules using the builder BUILDX (a modified LINK-11) always under DOS. All usage of the disk by SPEX and its tasks is in accordance with DOS file maintenance, generally involving contiguous files. The co-existence of the two systems on the disk of the computer is retained, and is in fact essential to SPEX.

During the actual conduct of the experiment, the core normally occupied by DOS is occupied by the executive modules of SPEX. The term "executive" is somewhat misleading, although it is used frequently in SPEX documentation. The executive modules of SPEX are simply a collection of services particularly useful in maintaining the multi-task environment described above. The use of these tasks is left to the experimenter to define by one or more "master" or "control" tasks. Only the combination of a control task and the executive modules constitutes a true executive system for controlling a computerized experiment.

Other documents on SPEX are :

SPEX Programming Manual -- referred to here as SPM

SPEX Operations Manual -- referred to here as SOM

Chapter 2

Core Management

Concepts in Core Management

In order to facilitate the management of core as it relates to the executive area, task occupied areas, and buffers allocated to tasks through requests to the executive, all available core in a given PDP-11 is quantized in units of 256 (decimal) words. Occupancy of a given area is registered by the executive by a word in which a single bit represents one 256-word quantum, called a "partition". To simplify the book-keeping involved in core management, the general restriction is placed on any area of occupancy (by tasks or buffers) that they be recordable within a single PDP-11 word of sixteen bits. This restriction introduces the concept of a "page" of core, of length $16 \times 256 = 4096$ words, and commencing at some multiple of 4096 words.

Recalling that byte addressing is employed in the PDP-11, a page of core spans an area of relative addresses from 00000 to 17777. The absolute address is determined by the page number, as follows:

Page	Addresses	Size
0	000000-017777	4k
1	020000-037777	8k
2	040000-057777	12k
3	060000-077777	16k
4	100000-117777	20k
5	120000-137777	24k
6	140000-157777	28k

The size column gives the nominal machine size necessary to accommodate the associated page.

Notice that the absolute location at which a page begins is obtained by (octally) doubling the page number and appending four zeroes.

The restriction that a task or buffer area be recordable within a single page-word relates the maximum allowable area to the "partition number" at which the area begins. Violations of this rule are referred to as paging errors. With all numbers expressed in octal, the following table governs allocations within a page:

Part	Start	Max
0	000000	20000
1	001000	17000
2	002000	16000
3	003000	15000
4	004000	14000
5	005000	13000
6	006000	12000
7	007000	11000
10	010000	10000
11	011000	7000
12	012000	6000
13	013000	5000
14	014000	4000
15	015000	3000
16	016000	2000
17	017000	1000

Note that the relative start of a given partition within a page is obtained by appending three zeroes to the partition number. The maximum size is obtained by subtracting the result (octally) from 20000. In decimal parlance, the paging concept of SPFX places an absolute maximum of 4K words on any task. Since buffers may be manipulated separately, and since tasks can call other tasks conceptually as subroutines, this restriction presents no major difficulty in programming for SPFX operation.

The resident SPFX executive generally occupies all of page 0 and the lower partitions of page 1. The exact size depends upon user modules incorporated with the executive at BNILDY time.

Tasks and Data Areas

A fundamental style in SPEX programming is to keep tasks and data areas as separate entities. (Data areas include: buffers, tables of constants, COMMON blocks, histograms, etc.) Tasks which operate on data areas are coded so that the data area is not part of the task's core area. This reduces disk space as well as the transfer time when the task is moved to and from the disk. Also it will allow the efficient use of memory segments with different access times on the PDP-11/45. The core needed for the data areas can be allocated and de-allocated by the GETCOR, ASKCOR, and FREECOR directives. In many cases a buffer is used by several tasks.

If tables of constants are required to be resident, they can be handled as Block Data COMMONS. That is, space is reserved in the resident code area and the table entries are TRANed in from the disk at SPEX initiation time. non-resident data can be QTRANed between core areas allocated by GETCOR and DOS files. None-the-less, it is sometimes more convenient to move a large data area in and out as part of a task. An example might be a task which is building a histogram.

Chapter 3

Disk Management

All SPEX disk transfers are to DOS "contiguous" disk files. The DOS program, PIP, can be used to move them, to convert them to linked files, or to create them from linked files. It must be remembered, however, that moving a contiguous file to a device with a larger block size (i.e. DECTAPE) and then moving it back will result in the file size being rounded up to the larger block size.

Tasks running under SPEX are able to use the disk in four ways:

1) Scratch Area

Tasks can dynamically request and relinquish disk scratch areas during the run. SPEX allocates these areas from a single, contiguous DOS file created by the initializer. The experimenter then uses the QTRAN directive to transfer to and from the scratch area. See SPM: ALLOCATE and DEALLOCATE Directives.

2) Referencing the Experimenter's DOS Files

The experimenter can reference his own DOS files by naming them in a file descriptor that is read by the initializer (from the special file called PHACOM.SPX). The initializer optionally allocates the file, and sets the necessary pointer to the file. The experimenter can then use the QTRAN directive. (References to linked files are limited to reading only.) See SPM chapter on PHACOM.

3) COMMON Block Files

Experiments often need tables of parameters available to tasks. These are usually stored in named COMMON blocks. The experimenter can specify that a COMMON block is to be initialized with values from a DOS disk file by the initializer. A pointer to the file is also set to enable the experimenter to update these disk files using QTRAN's. See SPM chapter on XCOM.

4) Task Image Files

Of course the tasks themselves reside on the disk in an exact core image. A task image is brought into core by the REQUEST and PIX directives. It is also possible for a task to roll itself out to the disk, thereby freeing its own core area, and to have the next REQUEST for it bring in and restart this rolled out version. This is done by the PAUSE directive and the intermediate disk files are called task NOW files. See SPM: REQUEST, PIX, PAUSE, REFRESH directives.

Chapter 4

SPEX Queues and the Node Pool

SPEX, like any executive system, requires sets of information blocks to carry out its control logic. The few control blocks which are of fixed length and number (such as Physical Device Blocks) are directly coded into the executive and can only be modified by reassembly of the relevant SPEX executive module. These are kept to an absolute minimum. The control blocks of interest here are those whose number continuously varies according to the current activity of the system - queued requests to I/O devices, the current active tasks, and so on. In the simplest scheme, a set number of control blocks are set aside for each activity. While there is the advantage that the control blocks can be of different size, each exactly matched to the activity being controlled, this is offset by the strong disadvantage that one activity may have many spare blocks (wasted space) at a given time, while another has exhausted all its control space and cannot sustain further queuing. A considerable effort would be required to maintain an appropriate balance of control block space according to expected activity of the entire system, and this could well require modification, in a way that is difficult to anticipate, according to the input data that the acquisition system is handling in any given experimental run.

SPEX has adopted the concept (and terminology) of the Digital Equipment RSX-15 software system, of containing a single "pool" of control blocks of fixed size that may be drawn upon by all activities requiring variable numbers of control blocks at different times. In this way exhaustion cannot occur on individual activities, but occurs only when the common pool is completely empty of control blocks. The corrective step to be taken when it is found that this situation can occur is straightforward - the system must be reassembled with a larger pool. In SPEX, the pool size is determined by the experimenter in the "XCOM" module which he supplies in building the executive, and no re-assembly or understanding of the basic logic modules of the executive is required to do this.

To implement this concept, the fixed-length control blocks, which remain in a fixed absolute core area regardless of the activity to which they are assigned at the moment, internally contain the indication of their current utilization. This is accomplished by reserving the first two words of each control block as forward and backward pointers, which link it to the next and preceding control block respectively. Such a linked control block is called a "node" in SPEX. The optimum size chosen in SPEX for its nodes is ten (decimal) words, the first two being pointers, and the remaining eight being available for the recording of data appropriate to the instantaneous use of the node. Every list or queue to which nodes may be linked requires a two-word "listhead" which contains the same pointer setup as the first two words of the node. Linked node lists are particularly powerful when, as occurs frequently in SPEX, the list is not simply FIFO or LIFO ordered, but priority ordered, so that insertions must be made not simply at one or the other end, but at some point that depends on the insertion data and the current contents of the

list. This operation in SPEX requires only the manipulation of pointer data, without the physical transfer of control block data in the list.

Not all lists in SPEX are "double-linked". In particular, the pool itself is simply forward-linked, and the operations of obtaining a node from the pool or restoring one to it always involve the first node pointed to by the listhead of the pool.

Chapter 5

The System Task List and the Active Task List

At the time of the run there is some finite number of tasks which may be REQUESTED to execute. They have been assembled and passed through the program BUILDX which creates a task core image file on the system disk (linking together separate modules into one task if desired). SPXX, however, must know where a given task is on the disk, where it is to be loaded into core, its size, its default priority, and other attributes. All this information is contained in the System Task List or STL. It is a list of nodes, each node containing information about one task. The list is in two parts: the system part, which contains I/O Handler Task STL nodes and the experimenter's part, the XSTL, which contains a node for each experimenter task. Any task not in the STL is unknown to SPXX. The experimenter must therefore build his own XSTL which refers to his tasks. This is easily done since most of the information for an STL node can be entered by the initializer. (See SPM Chapter on XSTL.)

A REQUEST for a task specifies its software priority, i.e. a number between 0 and 77, with 0 the lowest priority (200-277 for IOWT's). This is the priority with which the task is given processor control and with which its I/O requests are queued. It is not to be confused with the processor hardware priority. All SPXX tasks run at hardware priority zero (or one -- see SPM chapter on Interrupt Programming, the section on Holding Priority.). Only device "interrupt level service routines" (ILSR's) run at higher hardware priorities.

When a task is REQUESTED a node is created and put on the Active Task List (ATL). This list is ordered by task software priority. The tasks remain on the ATL while their actual execution is suspended for reasons such as: waiting for completion of an I/O operation, waiting for a buffer area to become available, voluntarily giving up control until reactivation, etc. They are removed from the ATL only when they EXIT.

Chapter 6

Significant Events and the ATL scan

One of the key concepts of SPEX is that of the significant event and ATL scan. It is best explained by a scenario:

Say there are two tasks, WRK and GET. GET accepts raw data from some hardware device (e.g. CAMAC) and does preliminary packing and rearranging of the data. WRK then processes the resulting buffers of data and returns them to GET. GET is a higher priority task than WRK. We assume that the data can come any time after it is expected, such as data generated by a spectrometer event. Presumably, at the time of the spectrometer event, GET is not doing anything and WRK is processing some buffer from a previous operation. So the state could be described as follows:

WRK is executing

GET is WAITING on event variable SEV, which is to be set non-zero by the CAMAC ILSR when a spectrometer event occurs.

When CAMAC has filled a buffer, WRK is interrupted and must give up control to GET and GET must be signalled that its buffer is ready. This is done as follows:

The CAMAC ILSR sets the event variable, SEV, and "declares a significant event". This says to SPEX that "something important has happened that may alter which task should have control of the processor". In response SPEX saves the state of WRK and then does an "ATL Scan" -- i.e. it examines in priority order each node to find the highest priority task that is ready to be given control. It finds GET before WRK and notes that GET is waiting for SEV to be non-zero, that SEV is indeed now non-zero, and so then gives control to GET at the instruction after its WAIT directive.

So now:

GET is processing its buffer and WRK is in the "interrupted state".

GET soon finishes preliminary processing; the buffer is ready for WRK. GET queues the buffer on WRK's input queue, sets some event variable for WRK (in case WRK was not busy at the time of the CAMAC interrupt), and finally issues a WAIT on SEV, i.e. the next CAMAC interrupt. This WAIT is a significant event (as are EXIT, REQUEST, PRRPCORE, and other directives). So the ATL is scanned and control finally returns to WRK. (It has been assumed in this example that WRK eventually catches up to GET before GET runs out of buffers for CAMAC to fill.)

The reader may wonder why subroutines could not be used. It must be remembered that SPEX handles the interactions between WRK and GET despite the simultaneous, but asynchronous, interactions of other tasks.