MSPS - A simple multiprocessor operating environment

Mark O'Neill,

Department of Photogrammetry and Surveying,
University College London,
London WC1E 6BT.
Electronic mail: oneill@cs.ucl.ac.uk (ARPANET),
oneill@uk.ac.ucl.cs (JANET).

ABSTRACT

A simple operating system intended for small microcomputers is described. This operating environment, MSPS, may easily be implemented at high level, using a suitable language such as BCPL, C or Pascal. The history of the MSPS operating system, and the progress which has been made to date is reported. The modular form of MSPS, and of the utilities which run under it is also described.

1.0 Introduction.

In recent years, there have been many attempts to produce portable operating systems, written in high level languages for mini and microcomputers. Examples of such operating systems include OS6 by Stoy et al [1], the UNIX System [2] by Kernighan and Ritchie, and the Tripos system [3] developed by Richards et al. While they are written in the main, at high level, in practise these operating systems are large and complicated software systems, and take of the order of a man-year to port from one machine to another. MSPS represents an attempt to define an operating environment which may be ported from machine to machine in much shorter time scales than any of the above operating systems. Furthermore, like the Kermit system developed by Columbia University, [4], the MSPS system is designed to be coded in any suitable language, such that it presents a uniform interface to the user. Although MSPS is intended in the main to be implemented on small 8 and 16 bit microprocessor based systems, there is no reason why it could not also be hosted by large mainframe computers. MSPS is a versatile operating system. It is capable of controlling the activities of both single and multiprocessor computer installations. When controlling multiprocessor installations, MSPS is of course capable of providing the user with a true multitasking capabilities. In single processor systems, its behaviour is similar to that of a single tasking operating system, such as the CP/M operating system, widely used to control Z80/8080 or 8085 based 8 bit single processor machines. The MSPS multiprocessor/multimachine environment is reminicent in many ways to the 'Worm' programs of Shoch ad Hup [5]. The SPS cluster, which will be described later, may be viewed as a 'worm'.

2.0 A Brief History of the MSPS System.

The acronym MSPS stands for (M)utliple (S)egment (P)rogramming (S)ystem. The Multiple part of the title, refers to the ability of MSPS to control a multiprocessor environment. The rest of the name Segment Programming System tells the reader something about the organisation of MSPS. MSPS grew out of a requirement to implement large software systems [6], on a small microcomputer system, a BBC Microcomputer Model 'B' equipped with a 6502 second processor and floppy discs. Such large packages could not be run on this system directly, due to the limitation of a 64K address space. Therefore, a simple overlay system SPS was developed. The basis of SPS is to divide the task to be programmed into a number of smaller tasks. These smaller tasks, which are called SEGMENTS, are of such a size that they fit into the system's 64K main memory. An intertask communication system is provided. This enables the tasks, which serially overlay one another to pass status and/or control information to each other. These individual tasks hand control to one another using a mechanism similar to the CHAIN statement supported by some BASIC interpreters. The memory available to an SPS task is divided up in the manner shown in figure 1. The data memory is preserved between task overlays, so that all the tasks able to manipulate the same data structures. In the prototype system, which is coded in Pascal, this is achieved by the use of pointers whic can be allocated to any user determined memory locations in the heap, using an undiscrimenated union within variant records [7]. This permits all tasks to address a predetermined area of memory, which contains the data to be processed, and in addition essential control and status information for the SPS system.

3.0 Nomenclature used in the SPS/ MSPS environment.

In order to efficiently describe the components of the SPS/ MSPS operating environment, a small specialist vocabulary has been developed.

3.1 The SEGMENT.

The SEGMENT is the basic processing structure of the SPS/ MPSS system. A segment is a program, which is capable of communication with other suitably written programs which adhere to the SPS communications protocol. The segment is in software terms a sequential structure, which deals with some convenient fraction of a task the overall execution of which is being handled by an application running under an SPS or MSPS environment.

3.2 The MODULE.

The MODULE is a block of code which defines a well defined subfunction within an SPS SEGMENT. For example, the SPS convention requires that all SPS segments possess a Communications MODULE, which contains code which implements the SPS intersegment communication primitives.

3.3 The TASK.

The TASK is a complete problem which is being dealt with by an SPS application. An example of a complete TASK is for example the monitoring of all aircraft movements at an airport. Individual SEGMENTS of this task may be dealing with aircraft which are taxiing, taking off, and landing respectively.

3.4 The CLUSTER.

The CLUSTER is a set of segments which are co-operating to comple a given TASK. In the simple example given above for example, the CLUSTER of the airport monitoring task is the set of segments {taxiing, taking-off, landing}. In general the segments of such a cluster will share a common data structure consisting of information to be processed and CLUSTER control and status information. A given SPS implementation may be either SERIAL or PARRELLEL. In the SERIAL SPS implementation, the separate segments of the application cluster overlay in a strictly serial sense. That is after a given segment has finished its work on the common cluster data structure, it is overlayed by the segment which will succeed it. The Parrallel SPS implementation, MSPS, is in effect a set of two or more serial SPS systems running on separate but communicating processor systems. Each processor system has its own data structure, and is capable of serially overlaying segments. The net result of this, is that it is possible to run more than one serial application cluster at once on such a multiprocessor system. In the MSPS system, facilities are also provided to enable the individual processor systems to send and receive data structures.

4.0 Segment structure.

As it has been mentioned, the basic structure in any SPS/ MSPS system is the segment. SPS is really a definition of a programming environment for small microcomputers. It therefore deliberately does not try to force a choice of programming language onto the system implementor. What it does do is to define a set of standards for a user environment which may be realised in one or more of a number of modern block structured languages. In particular, application segments in an SPS/ MSPS environment may be programmed in whatever language is optimal for the function of that segment. The only constraint is that the language chosen must be capable of supporting the SPS communication conventions. Irrespective of the programming languages which are chosen to code them, all SPS segments have the same conceptual structure. This conceptual structure, illustrated in figure 2, is built up of subunits known as MODULES. A typical SPS segment will contain a subset of the following modules:

a) The General Library Module. M

b) The Context Help System Module.

c) The Communication Modules. M

d) The Security Module. M

e) The User Interface Modules. M

f) The Application Code.

M indicates a module whose inclusion in a segment is mandatory if that segment is to adhere to the SPS convention.

4.1 The General Library Module.

The purpose of this module is to provide a library of standard procedures and functions which are not provided by the runtime system of the language in which a given segment is coded. For example the ISO Pascal language does not possess any built in string handling functions. Therefore the general library module for segments coded in this language would contain a set of standard string handling procedures to input, output, concatenate, and compare strings. The general library module also provides standard routines to maintain an overlay window system on the console. It may also provide extended file handling and error trapping utilities, although in the latest version of the MSPS environment these utilities have been given their own library modules.

4.2 The Context Help Module.

The purpose of the context help module is to provide the SPS segments with a comprehensive help system. Because of the modular nature of the SPS system, it is sensible to make the help system context orientated. Thus the help system when invoked, will supply help information pertaining to the segment that is active at the time. It is conceivable that large and complex segments may contain several subsystems within them, each of which may be sufficiently complex to warrant help information on them. The help system is therefore flexible enough to return help information not only at segment level, but also at the level of each of the major subsystems within a given segment.

4.3 The Communications Module.

The task of the communications module is that of providing a standard sub-system by which co-operating SPS segments may exchange both data and control and status information, and hence regulate successor activities. The complexity of this module is dependent upon the type of SPS implementation. For a the parrallel MSPS implementation, the module must contain primitives which allow information to be exchanged with segments running in other machines within the SPS environment. In the case of the serial implementation, control and status information is passed to other segments of the system via the control/status section of the shared data structure. In general primitives which communicate via the shared data structure are much more simple to code than those which have to Pass data between separate machines. Thus it will be appreciated that the communication module for the serial SPS is much more simple than that for the full MSPS environment.

4.31 The communications primitives.

Sys_error(err_no)

Output system error condition to the console. The system error is passed via the variable 'err_no'.

gm_poke(val, address).

Write to memory location in shared memory area. In the BBC Microcomputer installation this shared memory is in the I/O processor. The gm_poke primitive provides the basis for the MSPS intermachine communication system. 'val' is a byte value, and 'address' is 16 bit address in the shared memory area, which is accessible by all processors within the MSPS environment.

gflag(act, flg, p_ad).

Set/test/reset a global flag in shared memory area. This routine enables individual processors in the MSPS system to update the process table in the shared memory area. 'act' signifies whether the flag is to be read, tested or reset. 'flg' specifies the flag being considered. 'p_ad' is the address in the status table of the machine record whose flags are to be manipulated.

flag(act, flg).

Set/test/reset a local flag in the communication/status data structure of the machine running the current segment. This primitive allows the currently executing segment on a given machine to read status information supplied by an ancestoral segment, and to write status information to its successor segment. it will be performed.

putv(vec, da).

This primitive enables an SPS segment to initiate an successor. specifies the intersegment communications channel to which the initalisation command is to be consigned. In the present SPS implementation there are 3 command channels. The first is reserved for MASTER segment initalisation. The second is reserved for back initalisation of the current segment, thus allowing successors to call their ancestor segments with a minimum of overhead. The third channel used to hold the initalisation sequence for the successor segment. 'da' is a string entity and holds information about where the binary image of the successor is to be found on media.

putp(vec, da).

The putp primitive is similar to putv. It is used to initiate a process, that is, to start executing a segment on a processor other than that which is currently communicating with the user. As with the putv primitive there are three separate communication channels specified by 'vec'. 'da' is a string entity containing the address of the processor on which the next process is to executed.

pass(vec).

This procedure is used to pass the contents of the vectors loaded by the putv and putp primitives to the buffer of the command line interpreter of the machine running the current segment. The vector contents will be acted upon when the the current segment ceases processing.

start

This primitive, which has no parameters passes control to an successor segment, switching the current processor and copying the machine data structures if this has been requested by the user.

c_clst.

This primitive checks whether the applications cluster of the current segment is initalised. If not a suitable error message is written to the console, and the segment is aborted.

i_seg.

This primitive checks whether the current segment is allowed to run within the current segment. If not, an error message is printed at the console, and the segment is aborted.

s_ch.

This primitive is used to indicate to an successor the identity of the segment that was its immediate ancestor.

i_hp.

This primitive sets various pointers within the segment to point into the common data structures. It enables an successor to be 'linked' to the data structures left by its ancestor.

enter.

This primitive performs data pointer allocation, and entry security checks for an SPS segment by calling relevant primitives within the communications library. The precise action taken is dependent on whether the next segment is to be run on the current machine, or on a remote processor.

exit.

This primitive deallocates the present segment and calls the next segment to be executed. The exact action taken is dependent upon whether or not the next segment to be run is to be executed by the current machine.

4.4 The Security Module.

The function of the security module is to limit the damage caused when for any reason, a segment tries to exchange data or status information with a segment of a cluster to which it does not belong. If such a transaction where allowed to continue unchecked, it may well bring the SPS environment down. Therefore each segment possesses an identity string which tells the security system which other segments it is allowed to communicate with. If the security module finds that a given segment does not belong to the currently active cluster, the segment is aborted, and control is passed back to its ancestor, which will generally be the segment manager. The security module also causes a segment to abort if an attempt is made to execute it before a suitable data structure has been set up for it. Thus, it is not possible to execute any of the members of a given cluster before that cluster has been STARTED, or after it has been STOPPED. It may be appreciated, on inspection of the communications primitives, that some SPS implementations include the security module with the communications module.

4.5 The User Interface Modules.

The user interface modules contain a number of primitives which are responsible for the input of information from the keyboard and its output to the console. In general, these interface modules tend to be dependent on the requiremnets of a given applications cluster, and define command loop drivers (CMD's) which are common to all or many of the segments of a given application. The CML which handles console sub-windowing for the GPROC application which runs under SPS is a typical member of the user interface modules.

4.6 Command loop driver (CML).

The SPS system is designed to be easy to use. Consequently, use is made of single key commands to communicate with SPS segments. The command loop driver is the software structure responsible for decoding user input of this nature. Typically, the CML is coded using a structure similar to the Pascal CASE ... of or BCPL/C SWITCHON structures, each key representing a possible CASE of the switching structure, and the default case giving rise to a suitable error message. As the SPS system has been developed some standardisation within CML's based on functionality has emerged. Thus using the BBC Microcomputer as the terminal device, softkey <f0> is taken to mean 'invoke help system' in any CML. In a simlar manner, <f1> is taken to mean 'abort the current CML'. Many complex segments will contain more than one CML. The conceptual structure of a typical CML configured for use with the BBC terminal device is shown in listing 1.

5.0 The Cluster Structure.

From the preceding section, the reader will be aware that the the conventions of SPS impose a definite structure on the segments. The SPS convention also imposes a structure on the segment clusters. The cluster illustrated in figure 4, is a group of SPS program segments which are co-operating to achieve a given objective. The cluster invariably contains a number of special purpose segments. These are

a) The MASTER.

b) The SEGMENT manager.

c) The START segment.

d) The STOP segment.

The MASTER segment is defined as that segment which may be regarded as being at the apex of a given clusters task hierachy. For example, in the GPROC application package which performs graphical data manipulations under the SPS or MSPS environment, the MASTER segment controls interactive graphical manipulations. All non-special purpose segments other than START, STOP and SEMAN (the segment manager) are known as SLAVES. Often the Master segment will be the segment which is most often in communication with the user via the console.

The SEGMENT manager (SEMAN) is the most important segment of any SPS or MSPS cluster. It is in effect the command line interpreter for the SPS operating system environment. In Serial SPS systems, SEMAN is used to call the next segment out of store and run it, and also to display various system parameters. However in MSPS the segment manager has many other functions. It can example select the machine on which the next segment will run, copy data structures between physical machines, and display the current disposition of all the physical machines running in the current MSPS environment. In short the Segment Manager performs the scheduling and control functions for MSPS environments. The segment manager is different from all the other segments in one important respect. important quality. Unlike them, it is not owned by any particular cluster. Any cluster running in any group of physical machines within the MSPS environment may execute it. This ensures that a standard set of basic operating system utilities are available to any cluster which may require them.

The START segment is used to initalise the SPS/MSPS environment. It creates the data structures required by the rest of the system. In addition, the start segment may be customised so that in addition to the basic communications data structures required by the SPS/MSPS environment, it also creates the data structures required by the task which is to be performed. Prior to initalising the data structures in floppy disc based systems the start segment may also move some or all of the segments associated with the cluster that is to be run to a more rapidly accessible form of secondry storage such as RAM disc.

The STOP segment performs the opposite function to the start segment. It deallocates the data structures initalised by start and also delete all files in fast secondry storage. After executing STOP, START will have to executed once more, in order to re-create the environment in which the other (slave) segments will function correctly.

6.0 SPS/MSPS Data Structures.

Unlike many other operating environments, for example the Tripos System, or UNIX, the data structures associated with the MSPS system are simple. The basic SPS system requires only one data structure to function, while the MSPS system requires two. These data structures are the command/status table located in the segment processor memory, and the processor status table located in shared memory.

The processor status table, illustrated in figure 5, is stored in an area of common memory in the input output processor machine. In the prototype environment, implemented on a BBC Microcomputer system, the status table is located in the I/O machine which is accessible by all the slave machines. However, the MSPS system may be implemented to run on other parallel configurations in which the shared memory area may be resident elsewhere. In its most simple form, the status table is capable of storing two Boolean flags for each processor in the system. These flags indicate the processor status ($FF = busy, $00 = idle) and whether or not that processor currently owns the input/output channels ($FF = I/O owner, $00 = I/O non-owner). Thus, in this simple protocol, the segment that is running in the processor which is currently in communication with the console owns all of the I/O streams. All other segments have no access to the I/O devices unless they are specifically connected to the terminal by the segment manager. This means that segments running in the parrallel MSPS implementation are essentially stand alone entities. Even although they may co-operate on the completion of a given task, they must read all the data required to complete their sub-portion of the task before they are disconnected from the I/O streams. Experience with the a practical application, the GPROC graphic processing system [6], has shown that this is not such a severe limitation as the reader might think. The advantages to be derived from this simple approach are considerable. The interprocessor communication protocol is very simple to understand, and therefor to program. Problems of deadlock between processes running in the separate machines is virtually eliminated, and the system throughput is high,
even in 8 bit environments.

The command/status table, located in the segment processor memory, is a slightly more complex data structure. Its component parts are shown schematically in figure 6.

7.0 Conceptual hardware architecture.

The conceptual MSPS machine consists of a number of processing systems interconnected by some form of communication link. The MSPS system does not attempt to define the physical form of this communications link. This policy has been deliberately adopted, as it enables the MSPS environment to be implemented on a number of parrallel architectures. Examples of physical architectures in which the MSPS system may operate are diverse. The MSPS environment has been developed on the system illustrated schematically in figure 7. This is essentially a bus based architecture, with the Segment Procesor appearing in the memory map of the I/O processor which contains the processor status tables. However, if the low level communication software were written appropriately, the MSPS system could equally well be implemented on for example a Transputer [8] array, or some other appropriate parrallel hardware architecture. It should also be possible to implement the MSPS system on suitable network systems such as Ethernet [9], or The Cambridge Ring System.

8.0 References.

[1] Stoy J.E. and Strachey C,
OS6 - An experimental operating system for a small computer,
The Computer Journal,
Vol. 15, Nos 2 and 3, May and August 1972.

[2] Ritchie D.M. and Thompson K,
The Unix time sharing system,
Comm. ACM, Vol. 17, p365-375, July 1974.

[3] Richards M, Aylward A.R, Bond P, Evans R.D, and Knight B.J,
Tripos - A Portable Operating System for Mini-Computers,
Private communication, February 1979.

[4] De Cruz F,
Kermit User Guide,
Edn. 6, Revision 1,
Columbia University Centre for Computing Activities,
New York, 10027, June 1985.

[5] Shoch J.F and Hup J.A,
The 'Worm' - Early Experience With a Distributed Computation,
Comm. ACM, Vol. 25, No.3, p172-180, March 1982.

[6] O'Neill M.A,
An Outline of Work Completed on an Advanced On-Line Graphical Data Processing System,
Private Communication, October 1986.

[7] Cockerell, P,
ISO Pascal on the BBC Microcomputer,
Pub. Acornsoft, 1984.

[8] Newport J.R,
The Inmos Transputer,
32 Bit Microprocessors,
Ed. H.J. Mitchell, Pub. Collins.

[9] Metcalfe R.M and Boggs D.R,
Ethernet: Distributed Packet Switching for Local Computer Networks,
Comm. ACM, Vol. 19, No.7, p304-404, July 1976.

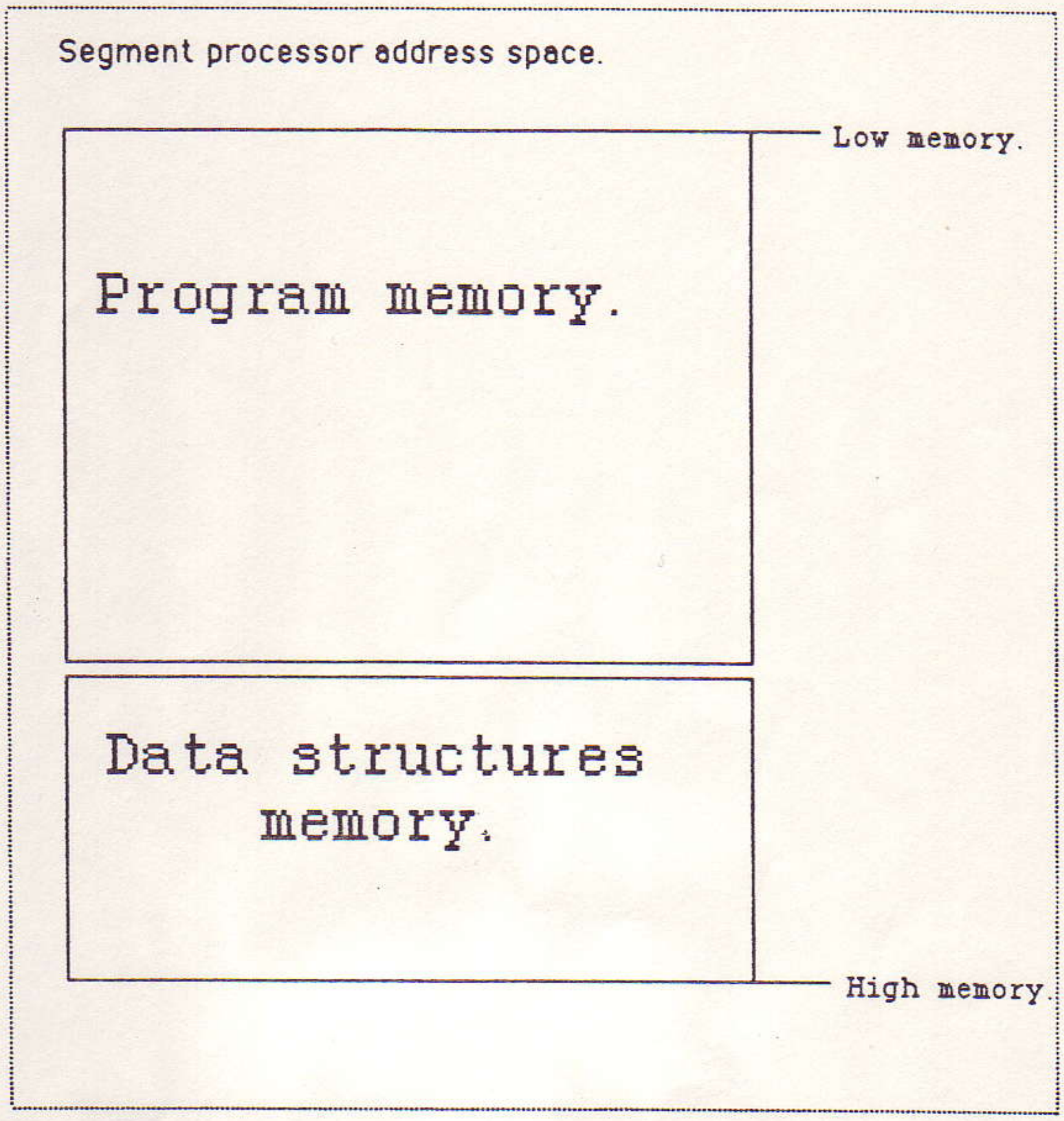


Figure 1. Showing the division of memory between program and data structures.

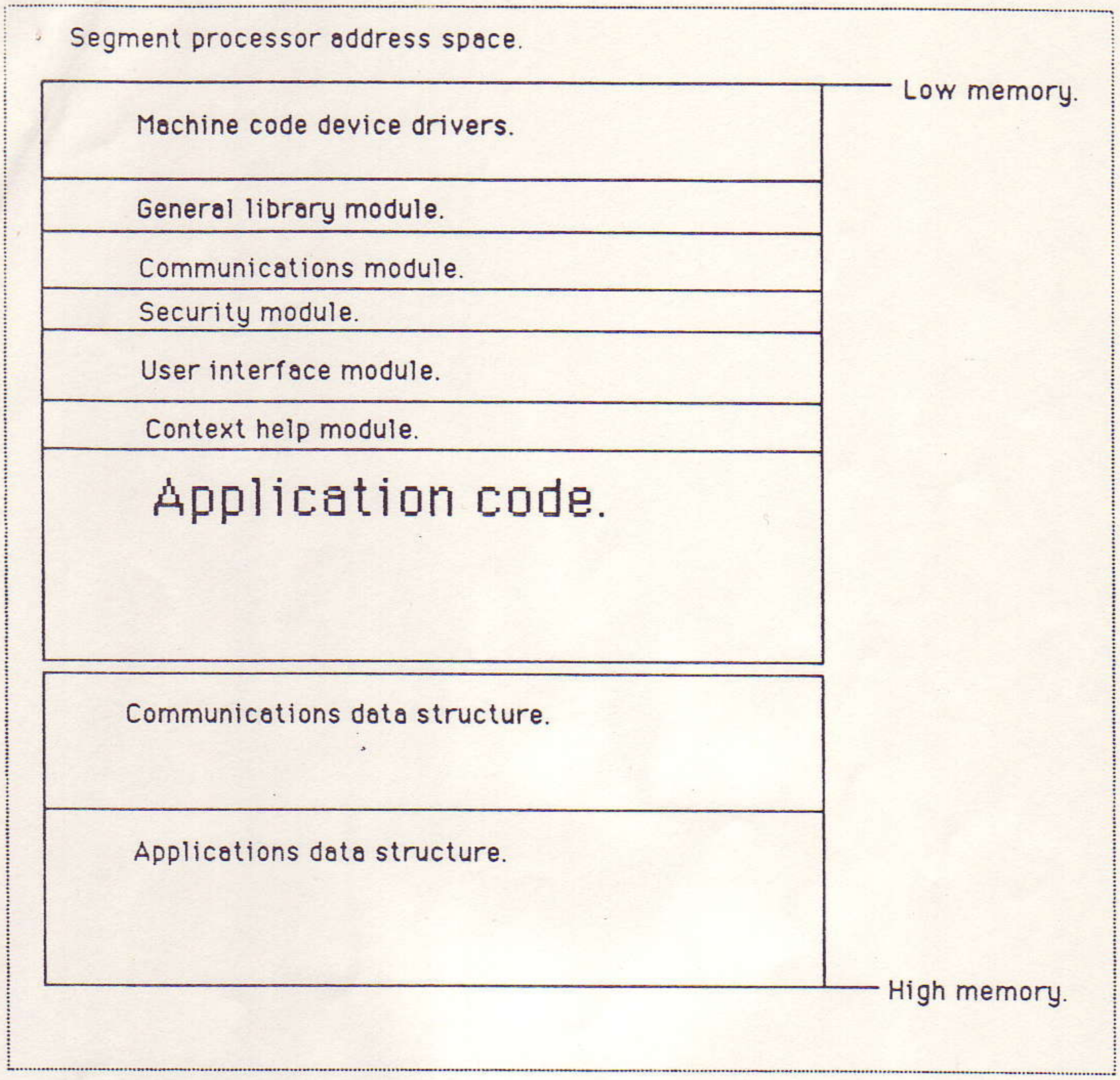
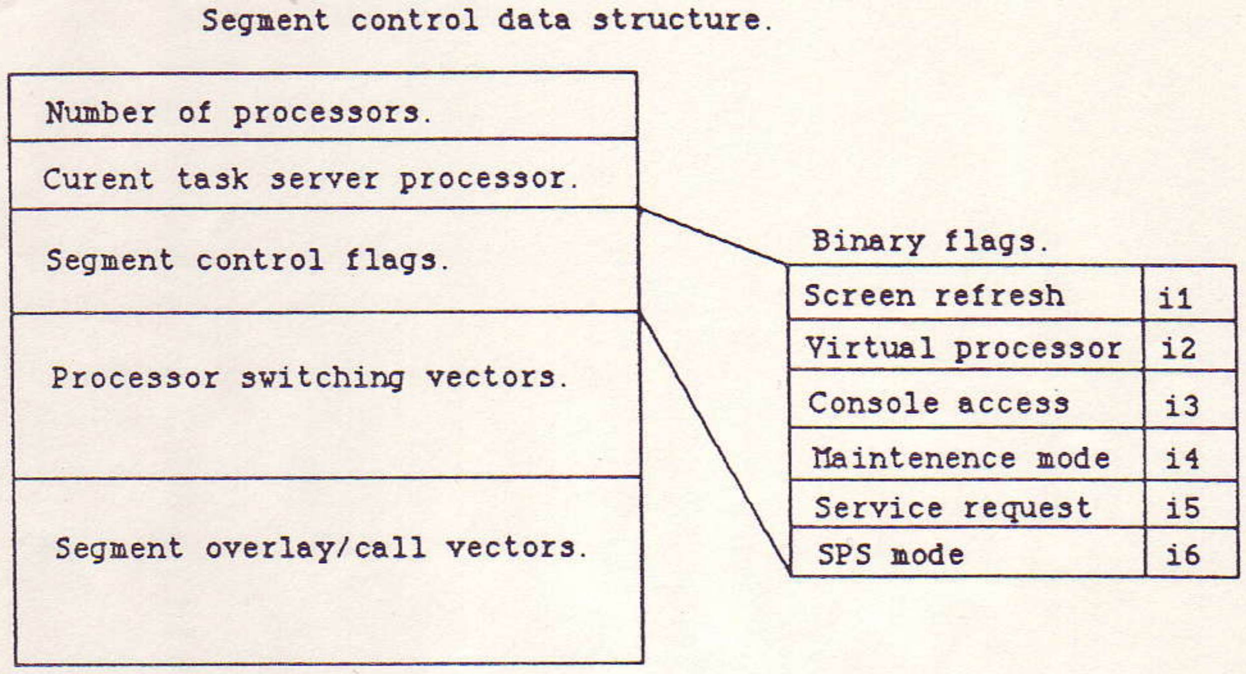


Figure 2. Showing the conceptual structure of an SPS segment.



The task server is that physical processor which is currently running the segment manager on an MSPS implementation.

Figure 3. Showing the principal SPS/MSPS intersegment communication datastructures.

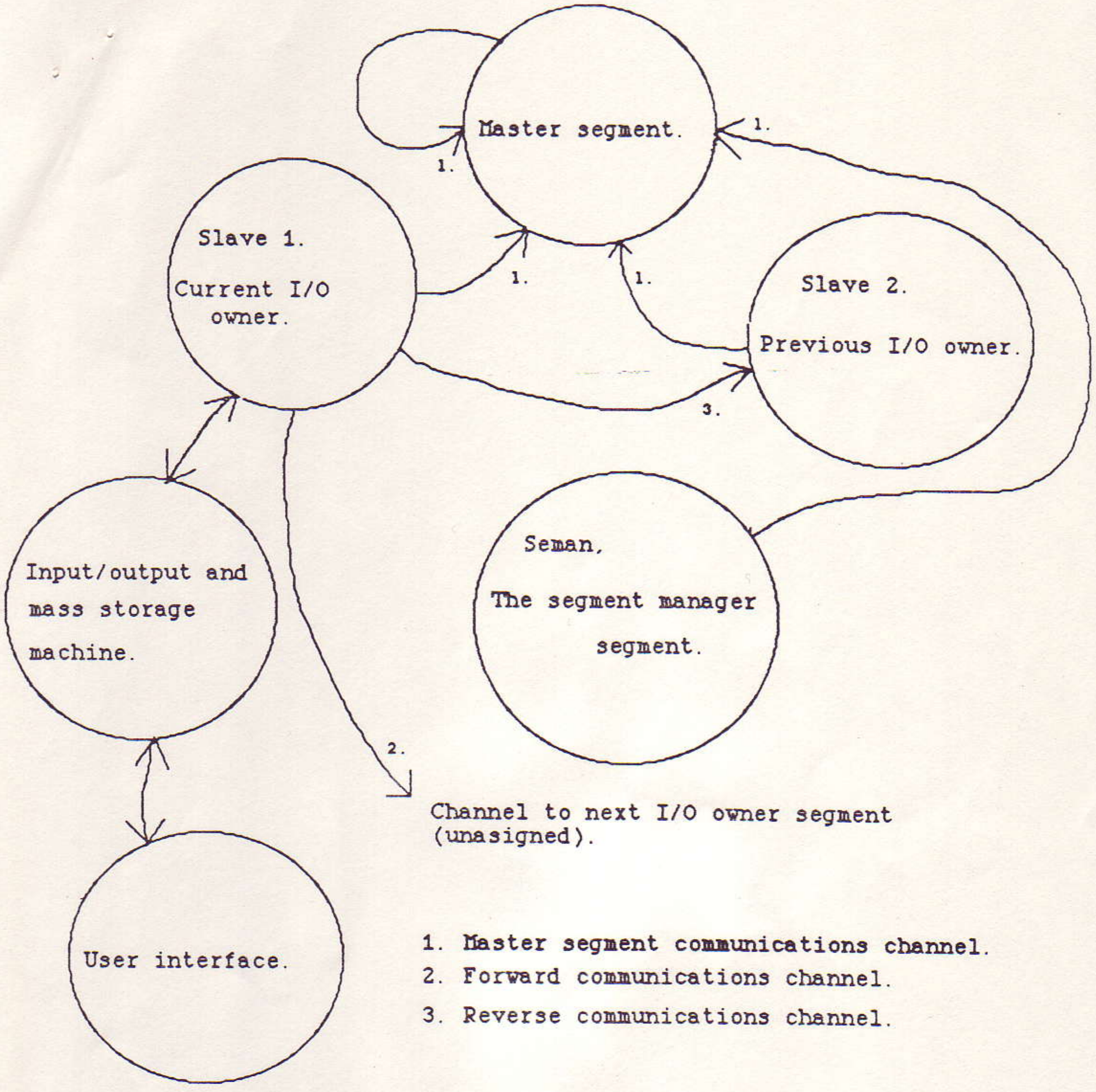

Master segment.
1.
1.
Slave 1.
Current I/O owner.
1.
1.
Slave 2.
Previous I/O owner.
3.
Input/output and mass storage machine.
Seman,
The segment manager segment.
2.
Channel to next I/O owner segment (unasigned).
User interface.
1. Master segment communications channel.
2. Forward communications channel.
3. Reverse communications channel.

Figure 4. Showing a schematic of a typical, active SPS application cluster.

To cluster.

Globally accessible memory segment.

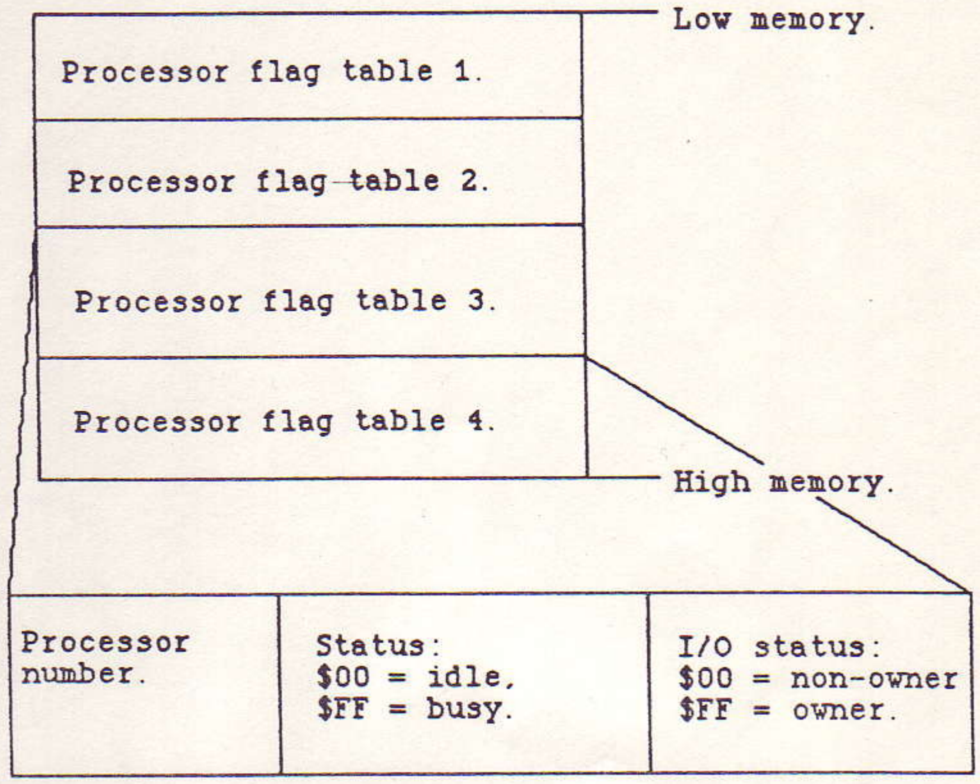
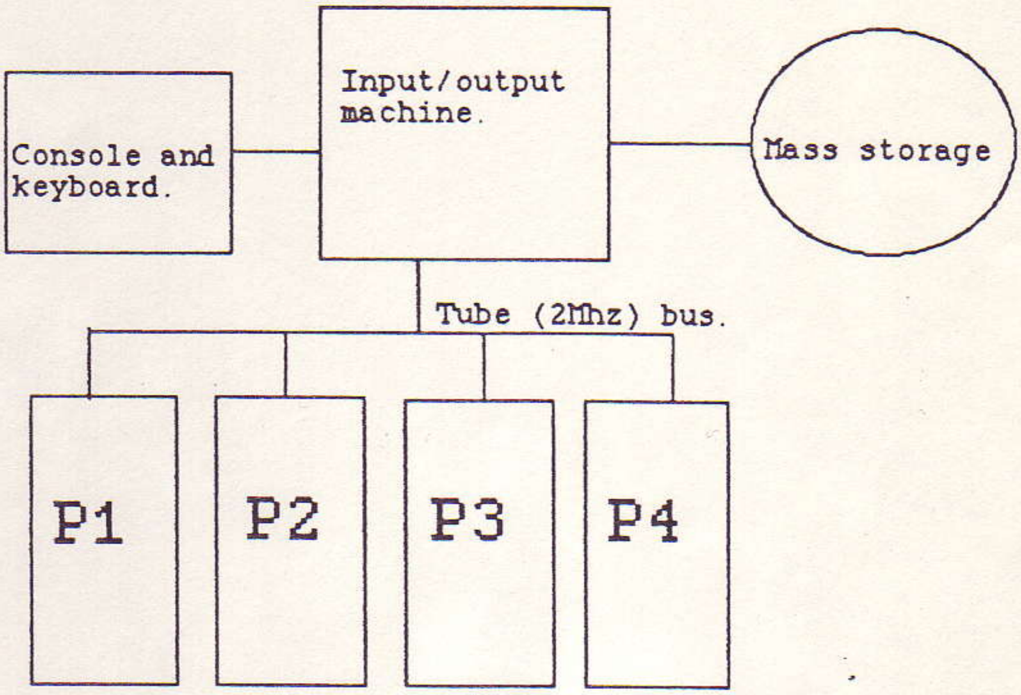


Figure 5. Showing a typical entry in the MSPS processor status table in global memory, which may be accessed by all physical processors.



P1 to P4, the sub-task processors, are either 4Mhz 65C02 units, equipped with 64K of RAM, or 8Mhz 68000 units equipped with up to 512K of RAM.

Figure 7A Schematic of the development environment for SPS and MSPS. This system is essentially a modified BBC Microcomputer System.

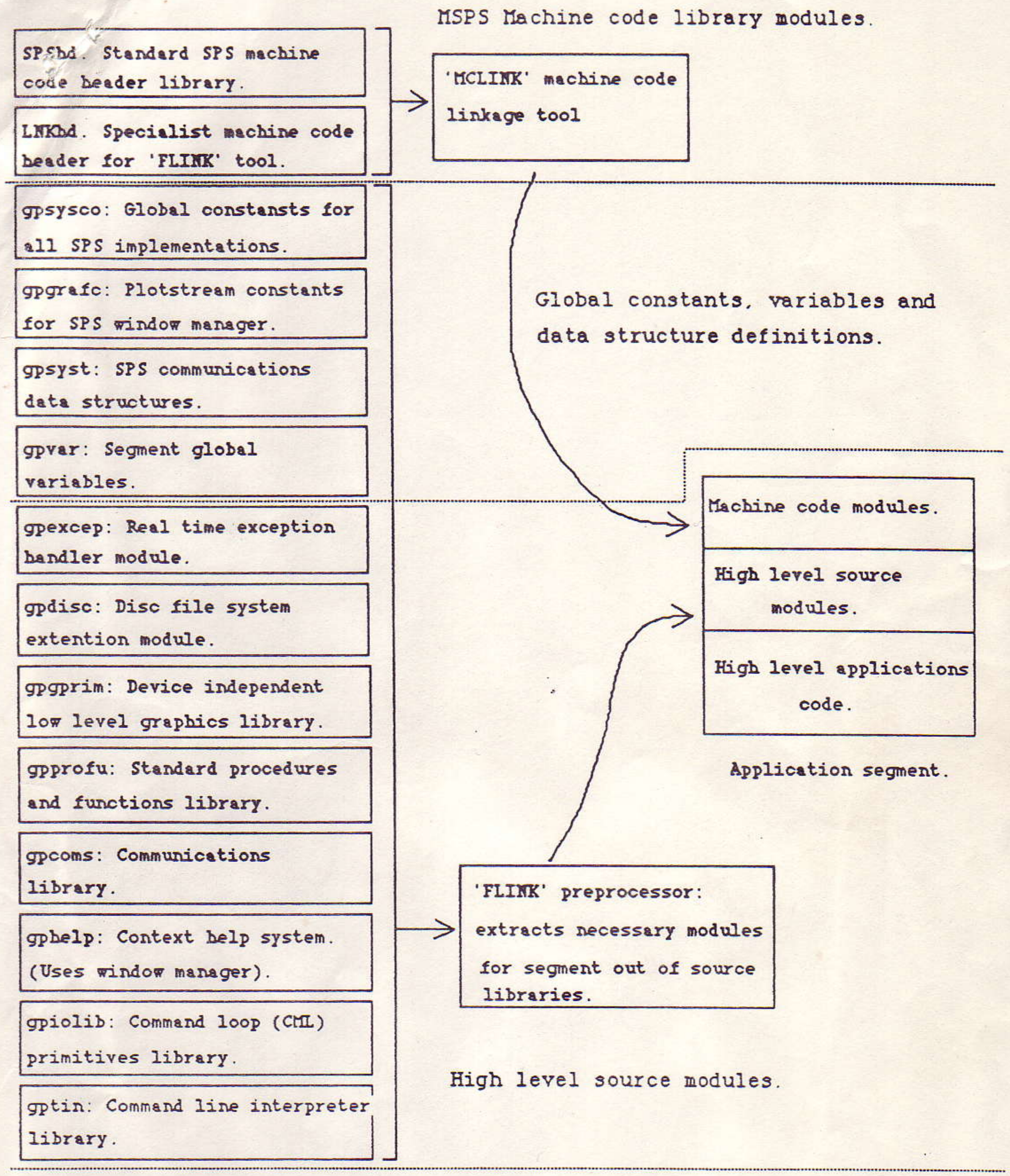

MSPS Machine code library modules.
SPShd. Standard SPS machine code header library.
LNKhd. Specialist machine code header for 'FLINK' tool.
'MCLINK' machine code linkage tool
gpsysco: Global constansts for all SPS implementations.
gpgrafc: Plotstream constants for SPS window manager.
gpsyst: SPS communications data structures.
gpvar: Segment global variables.
Global constants, variables and data structure definitions.
Machine code modules.
High level source modules.
High level applications code.
Application segment.
gpexcep: Real time exception handler module.
gpdisc: Disc file system extention module.
gpgprim: Device independent low level graphics library.
gpprofu: Standard procedures and functions library.
gpcoms: Communications library.
gphelp: Context help system. (Uses window manager).
gpiolib: Command loop (CML) primitives library.
gptin: Command line interpreter library.
'FLINK' preprocessor: extracts necessary modules for segment out of source libraries.
High level source modules.

Figure 8. Showing the organisation of the SPS/MSPS libraries, and the role of the segment assembly tools, 'MCLINK' [10] and 'FLINK' [11].