The standard EMAS subsystem

G. E. Millard*, D. J. Rees†, and H. Whitfield‡

The user image of the Edinburgh Multi-Access System (EMAS) is provided by a subsystem. The structure of the standard subsystem and its relationship to the supervisor and director is described. A description is given of conventions introduced by the subsystem for the organisation of files and the standard facilities with which a user is provided.
(Received January 1974)

The Edinburgh Multi-Access System (EMAS) is a virtual memory system providing a general time-sharing service on an ICL 4-75 computer. Each foreground or background user runs programs in an independent virtual memory and each such user process has a share of the resources of the system (core, CPU, etc.). A central non-paged *supervisor* provides virtual memory support, basic device driving and scheduling. Non-time-critical supervisory functions are performed by a *director*, which occupies part of the virtual memory belonging to each user process. Director also provides basic file system and console communication services for each process.

When a user process is created director loads and enters a standard control package, referred to as the *subsystem*. The subsystem is responsible for the management of the virtual memory, with the exception of that part containing director, and for providing a suitable interface to users, all privileged operations being performed by calls on director. The subsystem introduces conventions for the organisation of files, in particular program files. These conventions facilitate exploitation of the virtual memory organisation. A simple relationship between a user's commands and the routines which they activate enable him to easily adapt his interface with the system.

An overview of the system and a description of the non-paged supervisor is presented by Whitfield and Wight (1973), and a description of director is given by Rees (1975).

This paper describes the standard subsystem which provides a comprehensive set of facilities for users in both foreground, i.e. interactive, and background mode. The command processor interprets each command as a call on an external routine with that name. The standard subsystem commands are implemented as a collection of external routines, to which a user may easily add. The mapping between routines and commands is such that routines written by a user may themselves invoke subsystem commands simply by calling the appropriate routines.

A general principle in the design of EMAS has been that no decision should be made at a more 'central' or privileged position than is strictly necessary. Thus while the maintenance of page and segment tables is performed by supervisor, director is responsible for checking access permission to files and updating a user's file index. The subsystem makes such decisions as how and where files are mapped onto the user's virtual memory and is responsible for the internal organisation of files.

The director implements a File System, in which each file is an unstructured sequence of bytes with a length which is a multiple of a page (4,096 bytes). Before using a file it must be mapped onto the user's virtual memory by a call on director. Access to the file then only requires references to virtual memory addresses, the peripheral transfers required being initiated by the paging mechanism in supervisor. All active areas of a user's virtual memory must be parts of files. Thus, for example, a user program's stack is a temporary file with characteristics such as length and access permission.

Explicit input/output requests for interactive devices are handled by director. Other slow peripherals, e.g. card readers and line printers, are handled by a system process called *demons*. Information is passed between user processes and demons by transferring the ownership of files.

Fig. 1 illustrates the conceptual organisation of EMAS processes. In practice the code of director and the standard subsystem is shared by all users.

Subsystem structure

The principal components of the subsystem are shown in **Fig. 2.** When a user logs onto the system or a background job is initiated on his behalf a process is created containing only director, which is entered. Director extracts from the user's file index the identifier of the *basefile*, which is a program file containing the principal components of the subsystem. This is loaded and control is transferred, in non-privileged mode, to the Basic Command Interpreter (BCI).

Commands are normally typed in by a user at his console and interpreted by the BCI. Every command has the same basic

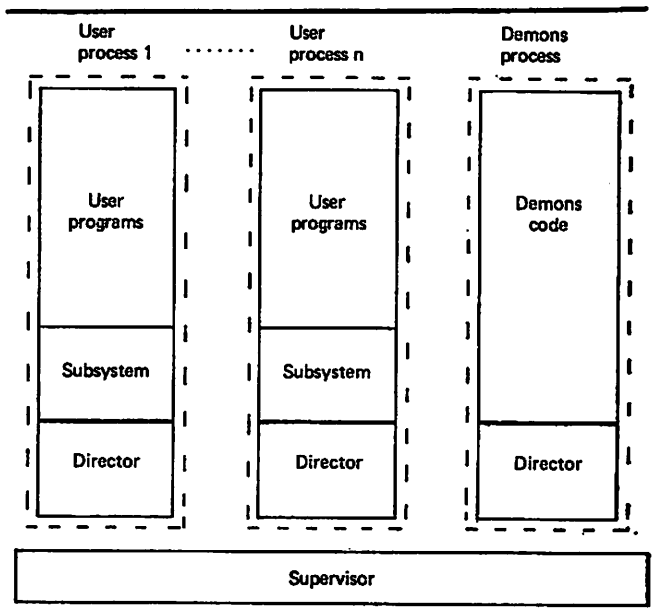


Fig. 1 Conceptual organisation of EMAS processes

*Edinburgh Regional Computing Centre, University of Edinburgh, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland.
†Department of Computer Science, University of Edinburgh, Scotland.
‡Department of Computer Science, University of Edinburgh, Scotland.
Present address: Mathematisch Instituut, Rijksuniversiteit te Groningen, Postbus 800, Groningen, The Netherlands.

format, the command name optionally followed, in brackets, by a string of characters. The command name is assumed to be the name of a routine, and the Program Loader is called to search through the current library structure in an attempt to locate the name of a program file containing that routine. If such a file exists it is loaded, the routine is entered and the string of characters is passed as a parameter. If the file is the basefile then the user has typed a standard subsystem command and the relevant routine is entered directly. This generalised mechanism enables a user to add a new command simply by compiling a routine and adding it to his library index.

Two packages play a major role in the functioning of the subsystem. The File Directory Package (FDP) uses the primitive file facilities of director to provide a convenient basic set of file facilities to the rest of the subsystem, and manages the virtual memory. The Environment Definition and Control Package (EDCP) provides an interface for the logical file handling requirements of programs and compilers, which it provides using FDP and other primitive facilities of director.

A contingency handler, essentially part of BCI, uses the signal facility in director to control the action of the subsystem following asynchronous or internally generated interrupts.

The components of the subsystem are discussed in more detail in the following sections.

The file system interface

Director maintains a file index for each user of the system and provides an orthogonal set of primitive file operations, These include file creation, extension and destruction, the setting of access permissions and the provision of information about files. Director is not concerned with the content of a file but only its external characteristics, i.e. its name, physical size (any multiple of a 4,096 byte page up to 1,024 pages) and access permission (which may be 'read only' or 'read and write', in each case 'shared' or 'unshared'). A user may only alter the external characteristics of a file owned by himself or obtain information about another user's file if he has been given specific access by that user.

Before a user can access a file it must be *connected* to the user's process, i.e. a mapping must be established between the file itself and the user's virtual memory. A file may be connected in more than one virtual memory, provided that appropriate shared access is permitted and that the mode of connection is the same in each case. Director provides a service which will attempt to connect a file at a point in the virtual memory determined by the subsystem.

Management of space within the virtual memory is a subsystem function and is concentrated in the File Directory Package. FDP maintains a map of the virtual memory and retains information concerning the size in pages and mode of access for all files which are connected to the user's virtual memory. This avoids the overhead in calling director whenever information is required concerning a currently connected file. All requests to director concerning files are routed through FDP which presents an interface to the rest of the subsystem which is more convenient to use than that provided by director. FDP also provides facilities to support the program loader and to set up and maintain a library index structure.

Internal file organisation

The subsystem provides for the handling of object program files and library index files, which are discussed later, and for two types of data file, oriented towards character and record manipulation.

Character files contain an unstructured sequence of characters with infixed control characters, principally 'newline' and 'newpage', and are normally read or generated as a character stream. Source programs are normally held in character files.

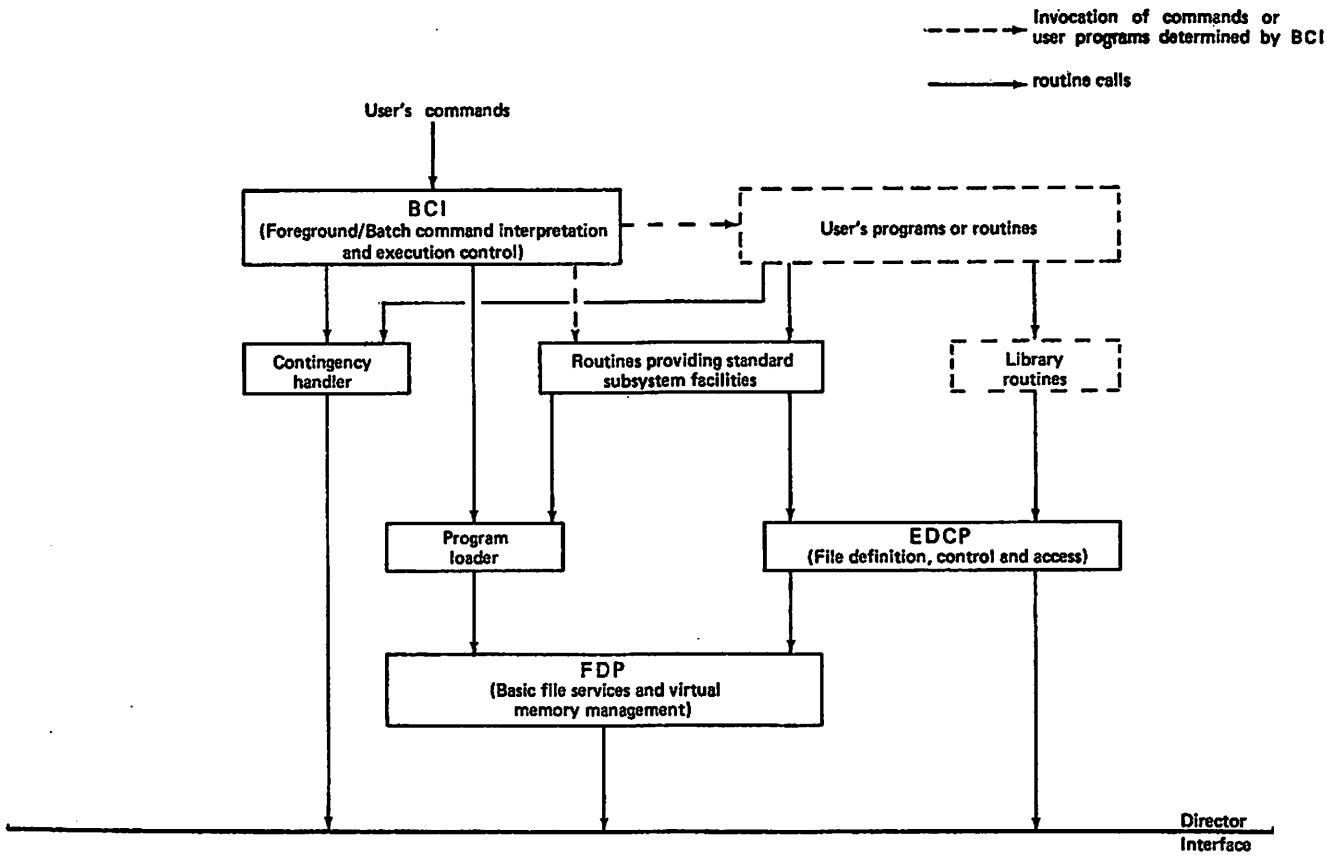


Fig. 2 Principal components of the subsystem

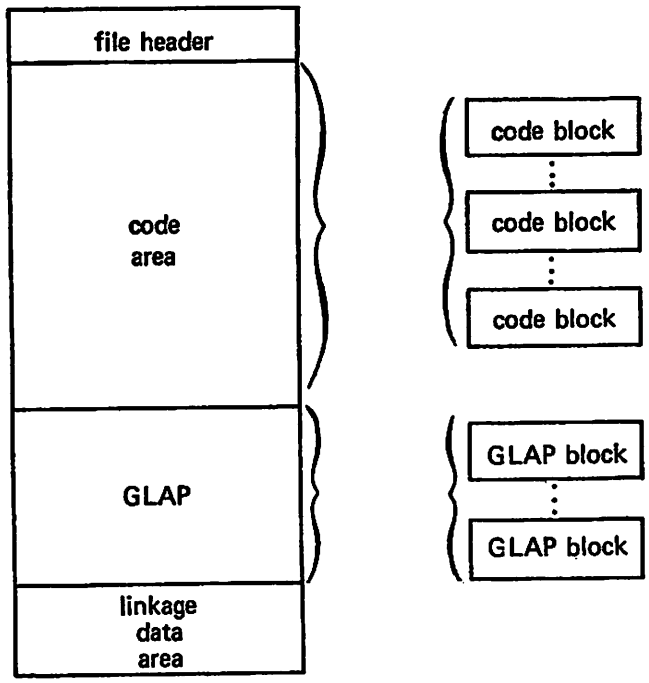


Fig. 3 Program file format

Record files may contain character or binary information which is written and read using the sequential and direct access facilities provided either through library routines (as with the local language IMP (Stephens, 1974)) or as part of the language (as is the case with FORTRAN). It is also possible to map a user-defined data structure directly onto a file. This is particularly useful for textual manipulation in the case of character files.

As has been mentioned earlier any information regarding the content of a file is the responsibility of the subsystem and is held within the file. Except in the case of object files, which are discussed below, the first four words of each file have a standard content as follows:

word 0 current length of information in the file
word 1 start of information relative to the start of the file
word 2 current created length of the file
word 3 file type.

In the case of record files further words are reserved for the record size, record format and maximum permitted file length. Fixed length records are recorded contiguously and locating a direct access record involves only a simple virtual memory address calculation. Variable length records are separated by forward and backward length indicators, thus enabling efficient implementation of back-space operations.

Program files and the program loader

A design objective of EMAS was that the code of programs should be protected from accidental or deliberate overwriting and that code should be shared between users whenever possible, with consequent savings of core, drum and disc space.

The standard program file format is shown in **Fig. 3.** Program files are shareable and while being executed are normally connected in virtual memory in read or read-shared mode. The first four words constitute a 'file header'. This provides the length of the file and the start of the Code area, the GLAP (General Linkage Area Pattern) and Linkage data area, each relative to the start of the file. The code area must be invariant, and usually comprises executable code, constants and diagnostic tables.

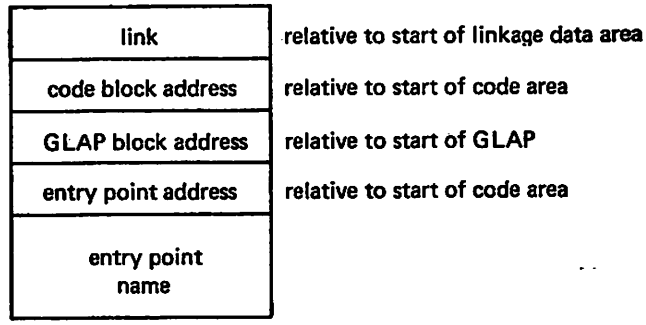
The GLAP contains the information needed for linking this program file to any file it may require, including other program files. When a program is loaded the actual program file is left unaltered and the necessary changes to establish external linkage are made to a copy of the GLAP, referred to as the GLA (General Linkage Area). This approach was suggested by Arden, Galler, O'Brien and Westervelt (1966). Any initialised data to be used by the program may be set up in the GLAP. A particular program file may contain several separately compiled routines, each of which will have a block of code and a block of GLAP within the appropriate areas of the file.

Information required by the Program Loader is provided in linked lists, the heads of which are stored at the front of the linkage data area.

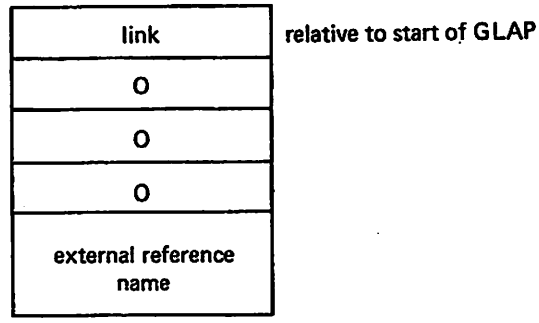
List 1 is a list of entry points at which the program file may be entered. These may be a main program or externally accessible routines. The list itself is contained in the linkage data area. Each item in the list has the format shown in **Fig. 4**(*a*).

List 2 is a list of external references to be satisfied by the Loader before the program file is entered. The list items are set up in the GLAP by the compiler which creates the file, with the format shown in **Fig. 4**(*b*). The loader replaces the three zero-filled words with the virtual memory addresses of the start of the code block, the start of the GLA block and the entry point address respectively of the program file satisfying the reference.

List 3 is a list of external references to be satisfied dynamically (i.e. at the time that an external routine is called). The compilers set up items in this list in the same way as for list 2, but the loader action is different. The three zero words are filled with the addresses of the list item itself, an environment descriptor for the loader (i.e. a register set and the program counter), and the entry point address of a dynamic load sequence. If the



(a) Entry point list item



(b) External reference list item

Fig. 4

program uses this information to attempt to enter the external routine it will enter the dynamic load sequence.

A fourth list of data entry points defines initialised data areas in the GLAP which may be referenced by other routines (e.g. FORTRAN COMMON areas), and a fifth list defines external data references to be satisfied by the loader before the program file is entered.

Run-time form

When the loading process is complete the code and GLA areas of a program are in different segments of the virtual memory. The code is in one or more segments which are connected in read mode, while the GLA is in a write unshared file. The same physical copy of the code may be in use by several other users who have the program loaded at the same time, though not necessarily at the same virtual memory address. It can be seen that the shareable part of a program file must not contain any 'absolute' virtual memory addresses. With the exception of the external references satisfied by the loader all virtual memory addresses have to be established by the program at run-time.

Calling sequence

Before entry to an external routine, the three addresses, code block, GLA block and entry point, inserted into the external reference item for the called routine by the loader, are loaded into three (consecutive) registers. One of the general purpose registers contains the address of the start of the free space on a common stack which all programs are constrained to use for the entry sequence to external routines. The calling routine may copy its registers on to the stack in the 16 words immediately ahead of the stack free space pointer. These are restored on return by the called routine. Any parameters to the call are set up beyond the register save area according to standard conventions. Control is then transferred to the entry point address with the return address in another register. In practice this sequence usually requires only three hardware instructions. In the case of a routine which is to be loaded dynamically this entry sequence, together with the loader action defined above, results in entry to the dynamic load sequence. This sequence finds the program file containing the routine (if it exists), loads it, and overwrites the relevant three words in the GLA of the calling routine with the addresses of the code block, the GLA block and the entry point of the called routine. Control is transferred to the entry point. The over-writing of the three words enables the program to use the same code sequence to jump directly to the external routine on any subsequent call.

The actual sequence used in the standard EMAS external routine entry is given in Appendix 1.

Library index structure

A library index, which is a one page file, contains entry names and their associated file names, and also a list of pointers to other library indices. The library facilities allow a user to create library indices referencing his own files and link them, together with other users' library indices, in any chosen structure (Fig. 5).

The purpose of the library index structure is to enable the program loader to associate any external reference with a particular file. Thereafter further information must be obtained from the file itself. Each reference is satisfied, if possible, from the index at the head of the structure or, failing that, from the list of libraries appended to that index in the order in which they were appended, and so on recursively. In the example the order of attempting to satisfy a reference would be Lib 1, Lib 11, Lib 111, Lib 12.

The present subsystem uses a dummy index as the head of its structure and appends to it the current user-nominated library structure and MANAGR.BASELIB. MANAGR is the user name of the system manager, who owns the principal files

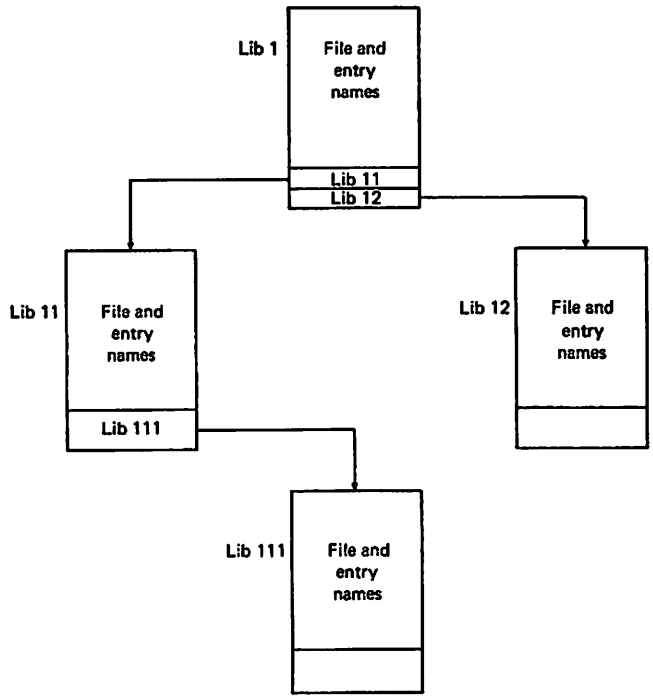


Fig. 5 Typical library index structure

shared by all users. MANAGR.BASELIB is a library index containing the entries of MANAGR.BASEFILE and to which other generally available libraries, for example those containing standard mathematical subroutines, are appended. MANAGR.BASEFILE is the program file containing most of the subsystem components, including the routines supporting the commands provided by the subsystem. This structure enables the user to redefine subsystem commands if he wishes by including routines with the appropriate names in his library index. The user is free to nominate separate library structures for each problem he may be working on.

The routines required for a particular program do not need to be included in the same program file as the main program block, but may be held in separate files provided that they are included in the current library index structure when the program is run. This is a great convenience during program development for the interactive user who gains from the ability to recompile small sections of his program. A command is provided for linking such files to form one program file when development is complete. This could result in improved performance because of the reduction in active core pages which may be achieved.

Environment definition

When a program is run it is necessary that a mapping is defined between logical files used by the program and actual files, whether these are permanent files, temporary files to be transmitted later to slow devices or an interactive device. Programs run in foreground mode have certain logical files mapped by default onto the initiating terminal. These may be over-ridden and others defined by a subsystem command. Such mappings are retained for the rest of the terminal session or until they are deleted or redefined.

Mappings may also be established through job control language for a job submitted in background mode or, as with all other subsystem commands, by a call on the corresponding external routine from within a user's program.

The Environment Definition and Control Package maintains tables defining the relationship between logical and physical files. These tables record relevant characteristics, e.g. the

record size for a direct access file, or the maximum permitted file size and, when the file is open, relevant virtual memory addresses.

EDCP also includes a set of interfacing routines which ensure that programs and compilers need not be aware of the device with which a file is associated, or the physical representation of data within a file.

Logical files may be mapped onto

(*a*) files, either permanent or temporary, held in the file system

(*b*) magnetic tape

(*c*) one or more terminals logged into the process

(*d*) other slow peripherals.

The handling of files in the file system has already been discussed.

The system supports a set of magnetic tape primitives through which the subsystem provides facilities, in background mode only, for sequential access to large data files (in excess of 1 megabyte) and for program and data interchange. The use of magnetic tape for archive and backup storage is fully discussed by Wight (1975).

The user's terminal may be used to interact with a program. A facility is provided by the director which enables programs to nominate text to be issued as a prompt when input is required. If the user anticipates data requirements by 'typing ahead' the prompt is suppressed. Any terminal logging into an active process will only be recognised by the subsystem if the initiating user is prepared to map a logical file onto it.

Slow devices other than interactive terminals, e.g. card readers and line printers, are handled by the special system process, demons, which deals with files as follows. Input files are submitted as batch jobs which are handled by demons. New files are created in the file system and records of these are entered in the relevant users' file indices. Output files are generated as temporary files in the file system. At the end of the current foreground command, or batch job, the file is transferred to demons with a request to output it using the appropriate mode and device type.

Contingency handling

A mechanism is provided by director whereby a user process may recover from program failures or other contingencies. The user program may specify an area of his virtual memory as a 'save area' and nominate an environment, i.e. a set of register values and a program address. If a contingency occurs director stores the contingency type, program address and register values at the time of failure in the 'save area' and transfers control to the nominated environment.

If a number of contingency traps are nominated, director stacks the information. For program failures the most recently stacked definition is used, but for asynchronous interrupts the oldest stacked definition, or 'outer level', is used. Facilities are also available for unstacking definitions, for inducing an artificial contingency at either the current or outer level of the stacked definitions, and for resuming the process with a nominated environment and program address (usually that at which the last interrupt occurred).

The BCI establishes the outer level contingency trap. As well as trapping all asynchronous interrupts this also acts as a 'backstop' to the trap which is set on each program entry. Program generated interrupts and forced contingencies unstack the contingency trap which is activated, while asynchronous interrupts do not.

In the case of program errors, such as overflow, address error, etc. control is normally transferred directly to a diagnostic routine which has sufficient information to provide source language diagnostics.

An asynchronous interrupt may be caused by the following:

(*a*) message from the machine operator

(*b*) local CPU time limit exceeded

(*c*) user console interrupt

(*d*) hardware malfunction affecting this process (e.g. disc transfer failure).

Following one of these interrupts control is transferred to the BCI. If a semaphore is set indicating that a critical part of the subsystem is active (e.g. modifying tables describing the content of the virtual memory) then that activity is allowed to continue until the semaphore is cleared. Action is then taken appropriate to the interrupt. In the case of (*a*) the subsystem normally sends the message to the user's console and then resumes from the point of interruption. Following (*b*) a contingency is forced at the 'current level' to provide diagnostics for the user.

When a user interrupts from his terminal he normally types a single character identifier to request specific subsystem action. This may be a request to abort the current command, with or without diagnostics, to monitor the progress of the current command, or to logout the user's terminal and allow execution of the current command to continue in background mode. Multi-character interrupt identifiers are noted for subsequent inspection by a user program. This is the means by which terminals logging in to an active process can make their presence known.

User facilities

A summary of the most frequently used subsystem commands is given in Appendix 2.

The principal compilers available are for IMP (Stephens, 1974) and FORTRAN IV (Millard, 1971). In each case great emphasis has been placed on the provision of good diagnostics, particularly at run-time. These include unassigned variable checks, array bound checks and, after an error has been detected, traceback through each currently active routine. This traceback provides, at each stage, line sequence numbers and a list of actual source identifiers with their current values. A program may contain routines written in either language, subject to the use of a common subset of parameter types in cross-language calls. A possibly unique feature of the implementation is that the routine traceback is fully effective for mixed-language programs. For developed programs the diagnostic facilities may be inhibited, partially or totally, by a command, PARM, specifying compilation options. These options will apply to all subsequent compilations in the current terminal session or until a further call on PARM. The compilers are invoked by a command of the form

IMP (source file name, object file name, listing file name).

The listing file name may be omitted, in which case a standard file, 'SS≠LIST', is used, or it could be '.LP', in which case a temporary file is created and passed to demons for printing when compilation is complete. In each case the terminal is given either a 'compilation successful' message or a summary of program faults.

A powerful context editor is available. The command has the form

EDIT(filename)

if a new file is to be created or an old one updated, or the form

EDIT(old filename, new filename)

if the original file is to be retained. The editor includes facilities for issuing repetitive commands, inserting the content of one file within another, and moving text around within a file. Since the file being edited is wholly contained within the virtual memory, moving the 'current pointer' backwards, to the top, or to the bottom of a file involve trivial operations.

Execution of a user program may be achieved by the command:

RUN (filename)

where filename refers to an object program file containing a main program block. It is not necessary that this file contains all the routines required to run the program provided they are entered in the current library index structure. Use of the dynamic linkage feature considerably reduces the overhead in loading large programs in which only a part is active on any single run, e.g. comprehensive statistical packages or programs under development. As discussed earlier it is not essential that a user program has a main program block, as it may be entered at any external routine simply by typing that routine name as a command.

A user may create a file containing a sequence of foreground commands and ask that this be obeyed either immediately or by submitting it as a background job for later execution.

The foreground facilities are complemented by a batch processing provision. The job control language definition which has been implemented is a subset of IBM 360/370 OS job control language which has been supported by the Regional Computing Centre on each of its generally available batch operating systems since 1969. Facilities are provided for IMP and FORTRAN compilations, running programs, editing and creating files containing information submitted on cards or paper tape in character or binary form. Jobs are submitted either on cards or paper tape or by preparing a file using the interactive editor and giving a command to inform the system process demons, which maintains a list of all batch jobs outstanding.

Any batch job which has a need to create or extend any permanent files for a user must be run in that user's process. It must therefore be run at a time when the user is not logged on at a terminal, and in practice such jobs are usually run overnight. The user may request that a job is run as soon as possible, implying that he is prepared for the inconvenience of not being able to log on to his process while his job is being executed.

Alternatively, jobs which require no access to permanent files other than generally permitted program files or subroutine libraries are run in special 'batch' processes of which at least one is normally run in parallel with daytime interactive use of the system.

Standards for parameter passing and the layout of data objects

As mentioned earlier a particular feature of the EMAS subsystem is the ability to cross call between routines generated by different language compilers and to obtain mixed language diagnostics. This can be done without formality and without intermediate conversion routines.

To do this efficiently it is necessary to have standards for the layout of data objects and particularly for arrays. The FORTRAN layout for arrays was chosen because FORTRAN specifies how they are to be stored and IMP does not. It is undesirable for language specifications to include layout conventions as this restricts the system programmer's choice—ALGOL 60, for example, makes no such specification and could therefore be included in the above scheme. A language specifying storage in a way conflicting with FORTRAN would make it necessary to pass accessing routines or descriptors as well as the data.

Development of new subsystems

As has been mentioned a user's file index contains the identifier of the basefile, i.e. the program file which is loaded by the director when a process is started. This file is usually MANAGR.BASEFILE, which is a program file permitted to all users in read-shared mode and contains most of the code of the standard subsystem (some components, e.g. compilers, are held in separate files which are loaded when required).

The subsystem includes a command which sends a message to director nominating a new basefile identifier to be entered into the user's file index. Thus it is possible, by setting a new basefile, logging off, and logging in again, to test a new subsystem without disturbing normal system use. Substituting a new, proven, subsystem for all users is done by replacing MANAGR.BASEFILE with the new version. This must be done by the system manager when no other users are on the system (MANAGR runs on a private basefile).

The components of the subsystem which are in separate files, i.e. not in MANAGR.BASEFILE, are tested simply by creating new files, inserting them in the developer's library index and issuing the relevant commands which will use the new components in preference to the 'standard' ones belonging to MANAGR.

The subsystem is written entirely in IMP, in common with the rest of the EMAS software. With the benefit of a high level language with excellent diagnostics it is possible to identify an error in the subsystem, correct it and try it again in the space of a few minutes.

Acknowledgement

The authors acknowledge the part played by P. D. Stephens in the development of subsystem standards.

Appendix 1

The use of registers in the EMAS standard subsystem external routine entry sequence is as follows:

R0-R3	Scratch registers—not saved
R4-R14	Environment registers—saved and restored
R15	Return address

The following have special uses:

R11	Stack free space pointer
R12	Code base register
R13	GLA base register
R14	Entry Point Address register

The following are typical call and return sequences:

CALLING ROUTINE

	Store parameters (if any) beyond the save area
STM 4, 14, 16(11)	Store R4-R14 in the save area on the stack
LM 12, 14, EPREF(13)	Load code, GLA and Entry Point addresses of the called routine from an entry point reference in the GLA of the calling program.
BALR 15, 14	Enter the called routine leaving the return address in R15.

STATE OF THE STACK AFTER THE ABOVE CALLING SEQUENCE

	save area	Parameters (if any)
0 1 2 3 4	14	
R11 16(11)		64(11)

CALLED ROUTINE

ST 15, 60(11)	It is usual to save the return address
LR local base, 11	on the stack
:	
LM 4, 15, 16 (local base)	Restore R4-R14 and pick up the return address. This ensures that R11 has been reset to its entry value if the called routine has changed it (which it generally has).
BR 15	Return to the calling routine

It should be observed that with an IBM 360 type order code the loading of the code base register is not strictly necessary, as

the called routine can set its own code base using a BALR 12, 0 instruction, and subtracting some fixed offset. However, the present sequence is more general and more efficient. The restoring of the registers from the save area could take place on return to the calling routine with a LM 4, 14, 16(11) instruction, but the present scheme is more efficient because the called routine usually stores the return address in memory to make R15 available for other uses and the present scheme reloads R15 at negligible cost. Because we have chosen not to save R0-R3 and R15 the corresponding positions in the save area can be used for other purposes (e.g. a dynamic or static environment chain in an ALGOL type language). However it is obvious that this is by no means necessary and a more general scheme could use the whole save area.

Appendix 2 Principal EMAS commands

ACCEPT	Accept the transfer of ownership of a file OFFERed by another owner, possibly renaming it.
ALERT	List information on the current state of the system.
APPENDLIB	Add a library to the current library index structure.
ARCHIVE	Nominate file(s) to be moved from on-line disc storage to magnetic tape.
CHERISH	Mark file(s) for which daily back-up copies are required. A copy is made only if the file has been written to in the preceding twenty-four hours.
CLEAR	Clear selectively or totally the file definitions established by DEFINE.
COPYFILE	Make a copy of a file. If the file being copied does not belong to the user then he must have been granted access permission to it.
CPULIMIT	Modify the cpu time allowed for each individual command.
DDLIST	List the current file definitions set up by DEFINE.
DEFINE	Set up a relationship between a logical data channel number and a file. The filename may take the form of a mnemonic to indicate particular device destinations, e.g. .LP implies a temporary file which is to be sent to a line printer when it is closed. The maximum permitted size of a file and its internal format may be specified, but for the majority of users the defaults are adequate.
DELETEJOB	Remove a job from the queue of DETACHed jobs.
DELIVER	Note new delivery information to be added to the top of all printed output.
DESTROY	Destroy file(s) owned by the current user.
DETACH	Submit a job to be run in background mode.
EDIT	A context editor.
FILEANAL	Provide the status and type of a file, together with information relevant to its type. For example, a list of entries and references is produced for an object file.
FINDFILE	Used to obtain information about files held in archive storage.
FINDJOB	Enquire the status of a job which has been DETACHed.
FLIST	List the files belonging to the user and resident in the on-line file system.
FORTE	Call the FORTRAN compiler.
HAZARD	Specify file(s) for which back-up is no longer required.
HELP	Provide general or specific assistance in the use of subsystem commands.
IMP	Call the IMP compiler.
INSERTFILE	Record the name of an object file and the entries associated with it in the current library index.
LINK	Consolidate a number of object program files.
LIST	A character or data file may be listed on the user's console or any suitable output device, either remote or central.
METER	Provide information on the user's consumption of resources.
OBEYFILE	Execute a sequence of foreground commands contained in a file.
OFFER	Offer a file for transfer to another user, who must ACCEPT it to complete the transfer.
PARM	Nominate compiler options.
PASSWORD	Change the foreground and/or background passwords.
PERMITFILE	Change the access permission of a file.
REMOVELIB	Delete a library reference from the current library index structure.
RESTORE	Request that a file be restored from archive storage to the on-line system.
RUN	Run a compiled program.
STOP	The process is stopped and the console disconnected.
USERLIB	Nominate a file to be used as the head of the current library index structure. If the nominated file does not exist a new file is created.
USERS	Enquire how many users are on the system.

References

ARDEN, B. W., GALLER, B. A., O'BRIEN, T. C., and WESTERVELT, F. H. (1966). Program and Addressing Structure in a Time-Sharing Environment, *JACM*, Vol. 13, pp. 1-16.

MILLARD, G. E. (1971). The Edinburgh FORTRAN compiler and its environment. *Proc. SEAS XVI*, Pisa, pp. 318-327.

REES, D. J. (1975). The EMAS Director, *The Computer Journal*, Vol. 18, No. 2, pp. 122-130.

STEPHENS, P. D. (1974). The IMP language and compiler. *The Computer Journal*, Vol. 17, No. 3, pp. 216-223.

WHITFIELD, H., and WIGHT, A. S. (1973). The Edinburgh Multi-Access System. *The Computer Journal*, Vol. 16, No. 4, pp. 331-346.

WIGHT, A. S. (1975). The EMAS Archiving Program. *The Computer Journal*, Vol. 18, No. 2, pp. 131-134.