Pete Maclean
July 21, 1972

Mass Storage Utilisation in LITTLE.

This paper explores the possibilities for the ways in which LITTLE might use mass storage, and describes three particular systems for mass storage transput. The first and most fundamental one is the "Buffer I/O Scheme" and is designed after FTN'S Buffer in and Buffer out. The other two systems are built on top of this —— one is a random file scheme, the other an automatic paging system.

Buffer I/O represents transput at its lowest level, in which all other input/output could be implemented —— including the regular READ/WRITE facilities.

My aim is to describe what I consider a good system for a general systems programming language, rather than propose an exact set of additions to LITTLE. In particular the example I/O statements given are to indicate necessary arguments rather than to suggest syntactical additions to LITTLE.

In researching for this my biggest problem was one of nomenclature. People accustomed to different machines and different systems use confusing terms relating to files and mass storage. So I will now present a brief glossary to define the terms I will use later on, giving the interpretation of them from my own point of view.

Logical Entities.

"FILE" —— my interpretation of a file is a generalisation of an array —— that is, an 'array' whose 'elements' are of arbitrary size —— and which is stored in some physical medium.

"Record" — A file is partitioned into records (its "elements"). Records are terminated by record marks. (Only one level of record marks is assumed).

"Tape" — a tape is a "multi-file file". It is partitioned into files by file marks and terminated by an information mark.

Physical Entities.

"M.S.U." — a mass storage unit e.g. disc, drum.

"P.R.U." — a physical record unit. Any M.S.U. is assumed to be partitioned into p.r.u.'s — the major significarce of which is that a p.r.u. is the only unit of information which can be transferred and from central memory.

"Magtape" — refers to a reel of magnetic tape as distinct from a logical tape. A reel of magtape is an m.s.u. which is assumed to have physical marks denoting end-of-tape and beginning-of-tape.

"Recall" — a m.s.u. transfer may be executed with or without recall. When done with recall, control is not returned to the caller until the data transmission is completed — otherwise control is returned as soon as the transfer initialisation has been processed. This corresponds to automatic recall on the 6600, and nothing corresponding to periodic recall is assumed.

Buffer I/O.

The main features of Buffer I/O are:

(1) One call causes the transfer of one p.r.u. of information to or from mass storage.

(2) Data transfer may be done with or without recall.

(3) This should represent in any system, I/O at its basic and fastest level.

"Buffer In" — This should cure a lot of headaches for people writing I/O routines. A call to Buffer In causes one p.r.u. to be transferred from a m.s.u. into central memory. Necessary arguments are:

pru — address of p.r.u. to be transferred — includes address of m.s.u. containing the p.r.u.

F — specifies whether read is binary or coded.

buff — array into which information is written. This must be large enough to accomodate largest p.r.u. that could be accessed.

rcl — specifies recall option.

n — returned as number of words transferred.

"Buffer Out" — causes a p.r.u. to be transferred from an array in central memory to a m.s.u. Necessary arguments are pru, F buff, rcl as above, plus:

n — no. of words to be transferred if p.r.u. size is variable (e.g. with some magtapes). If p.r.u. size is fixed then this parameter will be ignored.

Two further items are demanded to test the status of a m.s.u. these are exemplified by two functions ETX and EOT:—

ETX (msu) — tests if a data transfer has been completed — it must be called after every transfer. see below-

EOT (msu) — applies only to magtape and tests for an end-of-tape mark.

msu — specifies a m.s.u.

N.B. If recall is not used then any access of the buffer area before ETX may void the process

Parity errors and the like:

In some systems an unrecoverable parity error will cause an immediate abort — otherwise it should show up when ETX is called. This should also return information about file, record and information marks(if these are defined physically) — for instance ETX might return:

Ø — if transfer complete no. mark found.
1 — if record mark found
2 — if file mark found
3 — if information mark found
4 — if parity error
5 — if other error — e.g. invalid parameter

In the event of a parity error Buffer I/O might be programmed to retry the transfer some numbers of times.

Random File I/O

The features and uses of this scheme are:

(1) It is designed around the logical record as a unit of information.
(2) It is very versatile and can be used in segmentation and overlays as well as more conventional ways for random files.
(3) In order to access a random file the user must provide an index for it — i.e. an array with dimension at least as big as the largest number of records in the file, and size to accomodate any m.s.u. address. The user may access records by name or number. If by number, the number refers to an element of the index. If by name, the user must supply another array of the same dimension, and of size to accomodate the longest name (on which there need be no restriction). In either case, although arrays are supplied by the user, they are maintained by the random file I/O system.

(4) This system may be thought of as a logical derivation of Buffer I/O. That is a random file may be considered a virtual m.s.u. — and the records may be virtually thought of as existing contiguously since they do in the index — however no assumptions can be made about how they are stored physically.

Four calls are needed to use random files — one to open, close, read and write. For example:

open <u>fs</u>, <u>index</u>, <u>L</u> , <u>names</u>
read <u>fs</u>, <u>rs</u>, <u>n</u> , <u>wksp</u>
write <u>fs</u>, <u>rs</u>, <u>n</u>, <u>wksp</u>
close <u>fs</u>

arguments are:

<u>fs</u> — file specifier e.g. filename, tape number, fet address.
<u>index</u> — index array
<u>L</u> — dimension of index array
<u>names</u> — name of names array <u>if</u> accessing is to be done by name.
<u>rs</u> — record specifier.i.e. name or number
<u>n</u> — no. of words to be transferred
<u>wksp</u> — user's workspace

Open will open the random file and set up the index parameter — if the file is already open this serves to associate a new or extended index with it.

Read will transfer n words to the user's workspace starting from the first word of the record specified.

Write will write n words from the user's workspace to the record specified. If the record already exists a write will replace it— if it does not exist it will be added to the index, provided the index is not full.

Close writes the index to a special record on the file and closes it.

I have tried to think of alternative ways to access random files, and othershave been suggested, but none has come up that will fit well into this system. The prime aim has been to invent a way not requiring an index. The trouble is that although it would be possible to find anygiven record without an index, it cannot be expected that any such method will always be efficient. One must consider the worst case — which would probably be with the file distributed over random p.r.u.'s on an m.s.u. chained in one direction, like a list.

Paging System.

The paging system is the most sophisticated part of this package. Its main features are:

(1) It is highly automatic.
(2) It allows optional optimisation parameters.
(3) It is primarily designed to handle the paging of large arrays.

The minimal necessary code to cause an array to be paged is exemplified by:

Page arrayname

There are certain pieces of information which could be made available to the pager to improve efficiency in certain cases. In other cases, greater efficiency will result from the pager defaulting parameters to best suit the system. My idea is that such information may be made available to the pager at 3 levels, from which it will select the highest. These levels are:

3. The programmer.
2. The compiler/optimiser.
1. Default set.

In particular this allows the programmer to optionally specify extra paging parameters and calls if he thinks they will help. Such items include:

(1). Page size. The "reference string" (i.e. the sequence of elements referenced) of an array, may suggest specifying a special page size — i.e. if the reference string has some special non-serial pattern. For a serially-referenced array one is always best to leave it for the pager.
e.g. Page A, Pagesize = 3840;
However the pagesize must represent a number of words that is a power of 2, not less than 64. This helps all around — common fixed p.r.u. sizes are powers of 2 — housekeeping is much easier — and page frames can be optimally arranged according to the "Buddy System". It would probably often be the case that in any program all pagesizes would be the same, making things even simpler, but it would be too much to demand this.

(2). Paging m.s.u. If more than one m.s.u. is available the user may want to specify a particular one. By default the fastest device would normally be chosen.

(3). Rewrite. When a page is referenced for a write a flag associated with it is set saying that this page has been altered and when it is to be replaced it must be written back to disc. Then if it has been referenced only for read, it will not be rewritten to disc and some time will be saved. It would help if this information could come from the user or optimiser.

(4). Anticipatory fetches. The pager keeps a coefficient repesenting the randomness of the reference string to each paged array. This is called the "page reference string coefficient" (prsc) and is such that — $-1 \leq \text{prsc} \leq 1$, and

prsc = ∅ indicates referencing is random.
prsc = 1 indicates fetch of page i always followed by page $i+1$
prsc = - 1 indicates fetch of page i always followed by i-1.

then when a fetch of one page is demanded, the pager may actually fetch several consecutive pages, depending of availability of page frames and the prsc.

Anticipatory fetches can be enhanced by explicit commands to fetch pages placed suitably before references

e.g. Fetch Page (A(I))

meaning fetch the page of A containing element I of paged array A.

Also the programmer might be given the opportunity to suggest an initial value for the prsc.

(5). Release. A call may be inserted by programmer or optimiser following a section of code accessing a paged array saying this array will not be referenced again (or at least not for a time) if appropriate. In this case the page frames devoted to this array may be freed.

A Problem.

In order that this package be implemented on the 6600 it is essential that one very basic addition be made to the LITTLE language. In order to communicate with the operating system (regardless of which, including an imaginary one also written in LITTLE) some measure of absolute addressing must be available. The immediate need is to reference absolute location 1 (absolute within the field length that is). Externally this is trivial. I suggest the addition of some token meaning absolute zero —— this is enough to address any bit within the field length absolutely. Call this token Ω , then to store in RA + 1 for example one requires only

.F. 61, 60, Ω = ...

Then Ω really denotes the bit string that is your F.L.

I intend to follow this paper with one devoted to the paging system in detail, and the algorithms it uses. Also consideration is called for on the subjects of interrupts, M.S.U. addressing, and transput to other devices than M.S.U.'S