TECHNICAL MEMORANDUM

104

A memory manager for single and multi tasking applications

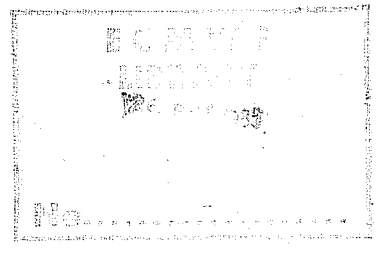
J.K. Gibson and D.W. Dent

Operations Department

August 1985

European Centre for Medium-Range Weather Forecasts
Europäisches Zentrum für mittelfristige Wettervorhersage
Centre européen pour les prévisions météorologiques à moyen

CONTENTS



1. INTRODUCTION

Computer programmes such as numerical weather prediction models often require large areas of main memory. Indeed, many such programmes cannot execute without the aid of an input/output system, with large areas of memory being continually refreshed from data held in on-line scratch space or backing store. To use the available memory efficiently, it is often necessary to re-use specific areas of memory for different purposes, over-writing one set of variables by another. Such practices are potentially dangerous, and must be organised with care. Memory maps must be produced, consulted, and updated as modifications are introduced.

Memory references to large shared areas of data become even more complex if a language of the FORTRAN type is used. Often it is necessary to resort to addressing displacements within a single, globally accessible array. Although displacement variables can be given meaningful names, the dimensional structure and direct relevance of well chosen array names and declarations are lost.

The ideas contained in the following paper were conceived in an attempt to provide a software solution to the memory management problem. They have been successfully implemented into the ECMWF operational weather prediction model using Cray Fortran, and could be adapted to any computer language which allows the use of based variables.

2. BASIC AIMS

The basic aims of a simple memory manager of the type under discussion are:

a) to enable the user to ALLOCATE arrays for use as WORK SPACE or LONG TERM STORAGE.

b) to enable LONG TERM STORAGE arrays to be LOCATED and used.

c) to enable WORK SPACE or LONG TERM STORAGE arrays to be returned when no longer required.

LONG TERM STORAGE is defined as array space required to be accessed by more than one routine. WORK SPACE is defined as array space required only within a single routine.

3. MECHANISM FOR ADDRESSING STORAGE

The memory manager is designed to supply pointers to based variables. In consequence, it is only suitable for use in association with languages which support such features (e.g. PL1, Cray Fortran, etc.) The concept of based variables is gradually becoming accepted as an important feature of high level computer languages. Simply stated, it enables the user to define a POINTER variable in association with an array or structure. The assignment of a valid memory address to the POINTER variable then defines the location of the corresponding array or structure in memory. Thus, in Cray Fortran,

```
DIMENSION  A(1000)
POINTER  (IB, B(1000))
IB=LOC(A)
```

has the effect of making array B equivalent to array A.

4. INTERFACE TO THE USER

Various implementations of the ideas contained in this paper will provide varying interfaces to the user. A sufficient interface, for the purpose of illustrating the following sections, would consist of five routines:

```
INILOC (KLENGTH1, KLENGTH2, KTASK, KSET)
ALLOCA (KPOINT, KSPACE, KNAME, KCODE)
ALLOCB (KPOINT, KSPACE, KNAME, KCODE)
LOCATE (KPOINT, KNAME, KCODE)
UNLOC  (KNAME, KCODE)
```

where

KLENGTH1 is the number of words of memory to be managed.
KPOINT is used to return a POINTER value to the user.
KSPACE is the array size in words.

KNAME	is the array name (LONG TERM STORAGE) or the name of the user routine (WORK SPACE)
KCODE	is a code number in the range 1 to 98 (LONG TERM STORAGE) or 99 (WORK SPACE).
KSET	is a value to be used to preset the managed memory
KLENGTH2 KTASK	are parameters required because of extensions to multi-tasking (see Section 7)

INILOC is used to initialise the memory manager, and is called once only. ALLOCA is used to allocate array space. Arrays are identified to the memory manager by means of a name, KNAME, and a code, KCODE. It is thus possible to have several arrays with the same name, provided a different code is associated with each. This device has been found particularly useful where different spatial representations of a variable are required to be stored simultaneously. Thus, for instance, a code of 3 could be assigned to all variables in grid point space. A single name could be used for a variable stored in both grid point and spectral space, the memory manager being capable of distinguishing the required array space according to the supplied value of the code. If work space is required, ALLOCA is called using the name of the calling routine as KNAME, and a code of 99.

ALLOCB is used to allocate array space within the managed memory area in such a way that repeated calls to ALLOCB will result in a set of arrays occupying a contiguous area of memory.

LOCATE is used to locate an array previously allocated by ALLOCA or ALLOCB. Only LONG TERM arrays may be located. WORK SPACE is only allocated, as it is only addressable from the allocating routine.

UNLOC releases previously allocated space. Released spaces may be re-allocated by subsequent calls to ALLOCA. LONG TERM arrays are released singly, whereas a single call to UNLOC releases all areas of WORK SPACE corresponding to KNAME.

5. MANAGEMENT METHOD

5.1 Management tables

Management information is maintained in stack-like tabular form. For each allocated area, table entries record:

a) the name (KNAME)

b) the address (KPOINT)

c) the code (KCODE) (long term only)

d) the length (KSPACE)

Two sets of tables are maintained - one for LONG TERM space, the other for WORK SPACE. A stack-pointer is associated with each table, indicating the number of valid entries in each table at any time.

5.2 Allocation strategy - work space

Work space is always allocated from the next area of available space to that of the last table entry. No attempt is made to re-use released areas corresponding to entries within the table. As routines requiring work space are rarely deeply nested, this simple strategy is both sufficient and efficient.

5.3 Allocation strategy - long term space

Efficient use of memory dictates that long term storage must re-use released areas without resulting in memory fragmentation. A simple but effective strategy is:

a) re-use areas only if they are exactly the correct size.

b) do not combine adjacent released areas into larger areas.

c) if no suitable area can be re-used (i.e. no exact fit) then add a new table entry and allocate from the residual area of available space.

The key to the success of this strategy lies within the repetitive nature of most numerical computations, and the resolution dependence of many array lengths. In consequence, the sizes of array spaces requested are far from random, and often similar.

5.4 Release strategy - work space and long term space

When space is released, the "name" entries in the tables are replaced by blanks. If the space corresponding to the last table entry is released, the stack-pointer is decremented and the appropriate length added to the available space.

6. ALGORITHMS

6.1 General

The following algorithms were devised specifically for a Cray-1 computer with vector capabilities and a 64 bit word length. Characters are stored 8 per word, using 8 bits per character (ASCII). Names are assumed to be left justified, up to 8 characters in length. Despite these machine dependent features, some of the following algorithms are suited to other configurations.

6.2 Matching a single table entry

A specialised routine, MATCH, was written in Cray assembler language (CAL) to examine table entries 64 at a time using the vector facilities of the Cray-1 thus:

a) store value to be matched in a scalar register.
b) load 64 values from table into a vector register.
c) store vector difference in vector register.
d) set mask to ones for zero vector elements.
e) update counter and table address.
f) return to b) if mask is zero.
g) count leading zeros in non-zero mask and update counter.
h) return value of counter (zero if no match found).

This provides a means of locating the entry in a table matching a given value. The subscript of the first matching entry is returned.

6.3 Matching two table entries simultaneously

The routine MATCH described above was extended to match two table entries to two given values simultaneously. The resulting routine, MATCH2, uses the following algorithm.

a) store value to be matched in Table 1 in scalar register 1 (S1)

b) store value to be matched in Table 2 in scalar register 2 (S2)

c) load 64 values from Table 1 into vector register 1 (V1)

d) load 64 values from Table 2 into vector register 2 (V2)

e) obtain vector difference S1-V1 in vector register 3 (V3)

f) set mask to ones for V3 zero elements (VM)

g) update counter, Table 1 address, and Table 2 address

h) store VM in scalar register 3 (S3)

i) return to c) if S3 is zero

j) count leading zeros in mask (A3)

k) transfer corresponding element from V2 into scalar register 5 (S5)

l) compare S5 with S2 and update counter

m) return to c) if not equal

n) return value of counter

This requires the second Table to be consulted only for segments where matching values have been found in Table 1. Where no matching values exist in Table 1, a zero value is returned; in such cases Table 2 is not examined, and the only additional cost is that incurred in updating Table 2's address in g) above.

6.4 Hashing names

Even when vector matching routines are available, searching Tables can be expensive. In consequence it was considered desirable to provide a hashing technique based on 8 character names and the integer code. The algorithm used is:

a) obtain exclusive OR (XOR) of KNAME with KNAME left shifted 11 bits
b) XOR a) with KNAME left shifted 21
c) XOR b) with KNAME left shifted 31
d) XOR c) with KNAME left shifted 41
e) return d) right shifted 53 with zero fill and merged with KCODE.

The result is an 11 bit value formed by combining parts of 7 characters of KNAME and including a contribution from KCODE.

Using this technique, more than 90% of the names used in ECMWF's operational numerical forecast are hashed to unique values in the range 0 to 2047.

6.5 Allocation of long term space

A hash table of length 2048 is initiated with all values equal to 1. The algorithm for allocation of long term space is then:

a) hash KNAME, and extract value stored at this entry of hash table (IPOS)

b) check entries in management tables at position IPOS against KNAME and KCODE. If correct entry located, return KPOINT from tables to user.

c) otherwise use 6.3 above to find table entries matching both KNAME and KCODE. If matched, return KPOINT from tables to user, and update hash table.

d) If no existing entry can be found, use 6.3 above to find table entries with blank name and matching length (KSPACE). If matched, return KPOINT from tables to user, and update hash table.

e) If no returned space of correct length can be found allocate space from the residual area, adding a new table entry to the top of the table stack, and up-dating the hash table.

6.6 Location of long term space

Previously allocated space is located as follows:

a) hash KNAME and extract hash table entry (IPOS)

b) check entry IPOS in management tables against KNAME and KCODE. If correct, return KPOINT to user.

c) otherwise use 6.3 above to match both KNAME and KODE. Return the matched entry for KPOINT to the user.

6.7 Release of long term space

Previously allocated space is released as follows:

a) hash KNAME and extract hash table entry, IPOS

b) check entry IPOS management tables. If not correct, match KNAME and KCODE using 6.3 above.

c) change KNAME in table entry to blank.

d) if IPOS is at the top of the table stack, collapse the stack, returning space to the residual area, until a non-blank entry appears at the top of the stack of names.

6.8 Memory distribution

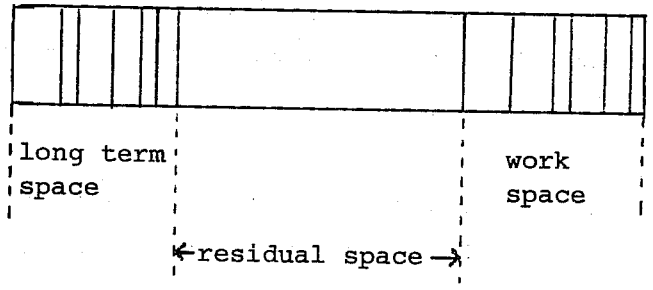


Fig.1 Memory distribution

Fig.1 illustrates one method of organising the memory space to be managed. Long term space is allocated from low order memory, work space from high order memory, with a moveable partition of residual space separating the two. This provides the flexibility of allowing all space not allocated for long term storage to be used as work space, and vice versa.

7. EXTENSION TO A MULTI-TASKING ENVIRONMENT

7.1 Multi-tasking space management

When modifying an application to take advantage of a multi-processor computer, various multi-tasking strategies may be employed. The memory manager is particularly useful when the chosen strategy shares the work to be done by segmenting the data and performing the same calculations in several tasks simultaneously. This requires access to different data areas in different tasks.

Since the code being executed is common to all tasks, it is convenient to allow the same names to be used for data areas, irrespective of the locations actually being referenced. This may be accomplished in an application using the memory manager by making the code parameter (KCODE) dependent on a task identifier. Thus if the first task is identified as task 0 and subsequent tasks as 1, 2, 3 etc. :

```
ITASK  =  IQTASK( )
ICODE = 10 + ITASK
CALL  ALLOCA (IPT1, ILEN, 'XXX', ICODE)
CALL  LOCATE (IPT2, 'YYY', ICODE)
```

where IQTASK is a function which retrieves the task number. Here, the newly allocated space will have the memory manager name 'XXX' regardless of how many tasks are executed. There will nevertheless be a unique area for each task, distinguished by the code values 10, 11,...etc.

Similarly, when LOCATEing previously allocated space, the area required for a specific task is obtained by establishing the correct value for ICODE.

7.2 Managing space for several tasks

When the memory manager is used in a multi-tasking environment, an important difference compared to single-tasking is the indeterminate order with which allocation requests are made. Hence the layout of allocated areas in memory becomes indeterminate leading to wasted space. (See Section 5.3).

To overcome this problem, separate areas of memory are managed for each task. The initialising routine INILOC is told what the maximum number of co-existing tasks will be. Since many allocation requests are made from the root or parent task (task 0) the manager allows for the space belonging to task 0 to be a different size than for other tasks. Hence, the INILOC parameter KLENGTH1 defines the size of managed memory for task 0, and KLENGTH2 defines the size of managed memory for all other tasks.

7.3 Memory distribution

The memory space is now organised as shown in Fig.2 (compare with Fig.1)

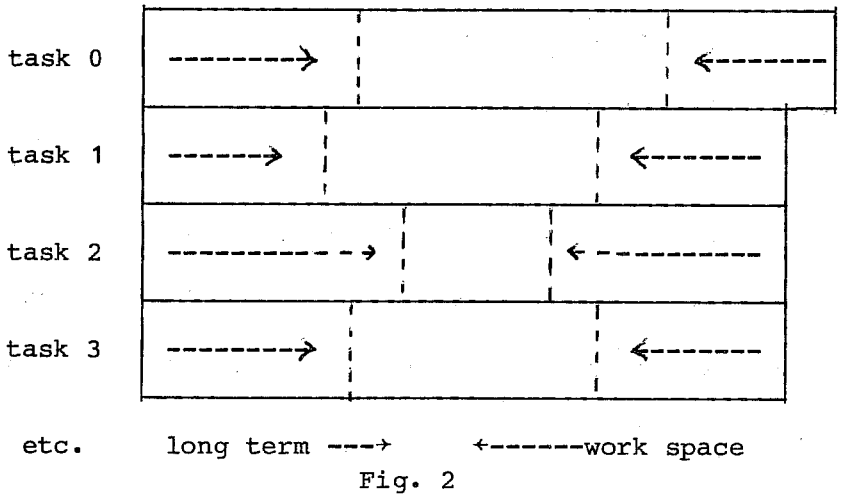


Fig. 2

there is a separate set of management tables (see 5.1) to describe the space for each task.

7.4 Choice of task space

A multi-tasking application has the freedom to choose which set of tables to use, and hence which area to allocate from. Normally, this choice is made automatically, in that a request from task N will be satisfied from the tables for that task. However, the user can override this decision by supplying the table number as an additional parameter (see Appendix).

When tables are being searched (e.g. by LOCATE) the searching mechanism will examine all sets of tables if the required entry is not found in the first choice. This is obviously inefficient and may be prevented by supplying the table number as an additional parameter.

8. IMPLEMENTATION USING THE CRAY HEAP MANAGER

8.1 Initialising

The memory manager as described in Section 7 may be used for both single and multi-tasking applications.

The memory to be managed is obtained by routine INILOC from the Cray Heap Manager by means of a call to HPALLOC. This requires that appropriate parameters are supplied to the loader as follows:

```
LDR,....., MM=INIT:INC, STK=STINIT:SINC, ...
```

where INIT is an initial heap allocation which may be increased as often as necessary in pieces of size INC. The stack parameter STINIT should be such that INIT > KTASKS * STINIT where KTASKS is the number of concurrent tasks in the application. The HPALLOC request also includes a small amount of space for tables.

Similar parameters exist when using the segmenting loader SEGLDR.

8.2 Adding to existing space

The memory manager includes a routine ADDMEM to change the size of an existing managed space. This is implemented using the heap manager call HPCLMOVE. This routine will move the existing space to a new location if necessary before appending to it. It is very important to appreciate that any array pointers which exist in the application code will be invalidated by this operation (i.e. it is necessary to reLOCATE). Since the original allocated space is unused after the ADDMEM operation, this is likely to result in inefficient use of memory and is therefore not recommended unless the original allocation is small.

9. CONCLUDING REMARKS

Memory management software using the above ideas has been found to be both flexible and efficient. A version used by ECMWF's numerical weather prediction model uses about 5% of the CPU time of the code; in addition about 1% is added in traceback overheads to allow comprehensive error facilities to provide useful information when errors are detected.

Memory management software does not relieve the user of the task of planning the use of memory space in a sensible manner. It does provide a means of using space efficiently, and allows the user to choose sensible array names and structures. The capability of allocating several areas of storage in a contiguous block facilitates the buffering of data for input/output processes. Flexibility is enhanced because extra variables can be introduced into the code without disturbing a rigidly structured memory configuration.

Good diagnostic and trace facilities are important. It has been found that the provision of switchable trace facilities has been especially useful in program development. The ability to call a trace routine to print a map of

the managed space in critical areas assists the user to plan, and often points to ways in which problems may be overcome.

A final, but important feature worthy of comment is the usefulness of based variables as a means of reducing programming errors. In an environment where all POINTER variables are set as a result of calls to memory management routines the likelihood of using nonassigned data, or over-writing previously assigned data incorrectly, is reduced. If a POINTER variable has not been set before reference is made to a based variable, a fatal error usually results. Over-writing in a managed system takes the form of allocating space which must first have been released. The resulting increased confidence in the integrity of the code is a considerable benefit.

APPENDIX A

The subroutines and parameter lists supported by the memory manager are listed here. [ ] indicates optional parameters.

```
ADDMEM (KALEN1 [,KALEN2])

ALLOCA (KPT,KLEN,KNAME,KCODE [,KTABLE])

ALLOCB (KPT,KLEN,KNAME,KCODE [,KTABLE])

ALLOCC (KVPT,KVLEN,KVNAME,KCODE,KNUM [,KTABLE])

FREEALL

INILOC (KLENGTH1,KLENGTH2,KTASKS,KSET)

LOCATE (KPT,KNAME,KCODE [,KTABLE])

LTRSTAK (KLTRACE)

MMGETL (KTLEN [,KTABLE])

MMGETN (KNAME,KLOC [,KTABLE])

MMLIST (KNAME,KLEN,KCODE)

MMPUTN (KNAME,KLOC [,KTABLE])

MMSWAPN (KLOC1, KLOC2 [,KTABLE])

OFFTR

ONTR (KTRACE,KTRLIM)

PUSHMEM [(KTABLE)]

QLEFT (KMLEN [,KTABLE])

QMEM (KMLEN)

RENAM (KNAME1,KCODE1,KNAME2,KCODE2[,KTABLE])

RLSEALL

UNLOC (KNAME,KCODE[,KTABLE])
```

The parameter meanings are as follows:

KALEN1 change in length of task 0 space

KALEN2 change in length of space for tasks other than 0

KCODE	Integer code in the range 1 to 99
KCODE1	" "
KCODE2	" "
KLEN	length of allocated array
KLENGTH1	size of space required for task 0
KLENGTH2	size of space required for tasks other than 0
KLTRACE	TRUE to switch on calling trace. FALSE to switch off
KMLEN	amount of remaining space
KNAME	array name (1 to 8 characters)
KNAME1	" "
KNAME2	" "
KNUM	number of arrays to be simultaneously allocated
KPT	address of allocated array
KSET	initial value for managed memory
KTABLE	table number to be used (optional parameter)
KTASKS	number of tasks required to use the memory manager
KTLEN	table length
KTRACE	maximum number of table entries to be printed
KTRLIM	print limit before tracing is switched off
KVLEN	vector of length KNUM containing KLEN values for each array
KVNAME	vector containing KNAME for each array
KVPT	vector containing KPT for each array

APPENDIX B

Description of memory manager routines:

ADDMEM (KALEN1 [,KALEN2])

change size of managed space (increase or decrease)

ALLOCA (KPT,KLEN,KNAME,KCODE [,KTABLE])

allocate space for an array with given name and code.

KCODE = 1 to 98 for long term storage

= 99 for work space.

ALLOCB (KPT,KLEN,KNAME,KCODE [,KTABLE])

as for ALLOCA but long term storage only

Consecutive calls are allocated contiguous memory

No reuse of released space except when at top of area

ALLOCC (KVPT,KVLEN,KVNAME,KCODE,KNUM [,KTABLE])

as for ALLOCB but KNUM arrays are allocated simultaneously

FREEALL

released managed space to system. Must be followed by INILOC before any new space can be allocated.

INILOC (KLENGTH1,KLENGTH2,KTASKS,KSET)

obtain memory to be managed and preset it to KSET

KLENGTH1 words are obtained for table 0

KLENGTH2 words are obtained for table 1 ... (KTASKS-1)

LOCATE (KPT,KNAME,KCODE [,KTABLE])

locate a previously allocated array

LTRSTAK (KLTRACE)

switch on or off the subroutine calling trace

MMGETL (KTLEN [,KTABLE])

return the current length of the memory manager table

MMGETN (KNAME,KLOC [,KTABLE])

return the name held in position KLOC of the name table

MMLIST (KNAME,KLEN,KCODE)
print contents of tables for debugging purposes

MMPUTN (KNAME,KLOC [,KTABLE])
insert KNAME into position KLOC of the name table

MMSWAPN (KLOC1, KLOC2 [,KTABLE])
swap entries in positions KLOC1 and KLOC2

OFFTR
switch off debugging tracing

ONTR (KTRACE, KTRLIM)
switch on debug tracing and establish print limits.

PUSHMEM [(KTABLE)]
compress space used by memory manager to eliminate unused portions. Applies either to all tables or to a specific table. Modifies addresses and therefore requires arrays to be re-located.

QLEFT (KMLEN [,KTABLE])
returns in KMLEN the residual space available for subsequent allocations.

QMEM (KMLEN)
returns the residual space available to the job, i.e. space not already in use by memory manager or operating system.

RENAM (KNAME1,KCODE1,KNAME2,KCODE2 [,KTABLE])
renames the array identified by KNAME1 and KCODE1 to the new identification of KNAME2 and KCODE2

RLSEALL
frees (UNLOCs) all arrays known to the memory manager space remains available to the manager

UNLOC (KNAME, KCODE[,KTABLE])
releases previously allocated space
If KCODE = 99 all arrays named KNAME are released.
Otherwise, a single long term array is released.