

# Memory Allocation Page

*Diagnostic like Page K*

Tue, Jul 20, 2004

The IRM 68K-based systems include a page application, normally assigned to Page K, that shows what allocated memory blocks are in existence, updated at 15 Hz. But the vxWorks-based PowerPC system has so far lacked a similar diagnostic. An initial step has been provided for that system, however, that writes a record into a data stream when a block is allocated or freed. (See the note, *Memory Allocation Diagnostic*, Feb 4, 2003, for a description of this interim implementation.) The new plan is to write a page application that can monitor what is written into the data stream and build a list of allocated blocks similar to that in the IRM. Because it is based on a data stream, it is possible to view allocated memory blocks in another node, not merely in the local node. This note is written to evolve a design for the new Page K.

## *Preview of plan*

Just to get started, assume that we can make a 15 Hz data request for the contents of the most recent records written into the ALLOCLOG data stream. (If this is in another node, it will first be necessary to look up the index of the data stream in the DSTRM table, which can be done via the generic name lookup supported by listype 55.) Process these records to maintain a data structure that includes all of the currently allocated blocks, at least those known to be created since the request was started. (If it is desirable to maintain such a structure that reaches back to the time of reboot, it would be necessary to implement the same processing logic within a local application that would become active immediately upon reboot and remain active indefinitely.)

Here is the format of the 16-byte data stream records:

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
rAddr	4	Base address of allocated block
rSize	2	Block size
rFlag	1	Flags: Ax = allocation, Fx = free, x = flags
rType	1	Memory block type#
rTime	8	Date, time of allocation or free call, in usual BCD format.

How shall the data stream records be processed to build a structure that is maintained in increasing order of memory block address? Consider the idea of keeping a linked list of active block references. Imagine an array of records with the following structure:

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
aFwd	4	Ptr to next record in sequence of block address
aInfo	4	Extra information: LA name, requesting node#
aAddr	4	Base address of allocated block
aSize	2	Block size
aFlag	1	Flags from data stream record
aType	1	Memory block type#

Assume there is a variable called aHead that is a ptr to the head of this linked list, the record that contains the lowest block address in its aAddr field. In case there are no

allocated blocks known since the initial request was made, `aHead` will be `NULL`.

Assume another variable called `fHead` contains a ptr to a linked list of free records in the same linked list. Initially, `fHead` points to the start of the linked list, and every `aFwd` field points to the next entry in sequence, with the last record having `NULL` in that field.

Initially, all records are free, and the free list is of maximum length. Consider that the constant `aRecMax` is the total number of records available in the array. As blocks are allocated and freed over time, the physical order of records in the allocated linked list and in the free list may change. But one can easily traverse the list in numeric order of `aAddr`, since that is the purpose for this structure. The idea here is that the array provides a place to keep the block addresses that are active, and the records are logically maintained in sequential order of increasing `aAddr`.

### *Processing allocation records*

How do we process allocation records found in the data stream? If an allocation record containing the block address `rAddr` is found, insert it into the array by calling the function `ARecInsert`. What does `ARecInsert` do? Search through the linked list to find a record, if any, that has an `aAddr` field higher than the `rAddr` to be inserted. If one is found, logically insert the new record just before that one, removing a free record from the free list for the purpose. Removal from the free list can be done by taking the record pointed to by `fHead`. The `aFwd` field of that free record is copied into `fHead`.

If no higher `aAddr` field is found, compared with the new `rAddr`, append the new record at the end. Make the `aFwd` field of the last record point to the new record obtained from the free list, and make the `aFwd` field of the new record point to `NULL`.

### *Processing free records*

What if we encounter a free block record in the data stream? Find the record with a matching `aAddr`, if any, delete it from the linked list, and return it to the free list. Copy the `aFwd` field of the matching record to the `aFwd` field of the previous record. Add the newly freed entry to the free list at the head by copying `fHead` to its `aFwd` field and setting `fHead` to point to the newly freed record. The next time we need a free record, it will be this one, unless additional records are freed before that time.

### *Efficient methods*

What efficiencies can reduce the amount of scanning through the linked list that is required? By examining the list of new records sampled from the data stream, if we find an allocation record followed by a freeing record with the same `rAddr` field, we can forget about inserting the allocation record into the linked list. This logic can be added to the processing of each allocation record. Before doing the insert, scan from that point in the buffer of new records to find a match on the `rAddr` field. If there is a match, skip the insert and erase the matching entry, so it will not be interpreted as a freeing record later. If there is not a match, go ahead and perform the insert as described above.

We can retain a `aRecPtr` to the latest record in the linked list that was processed. When scanning to find a place for an insert, or to perform a deletion, first check the `rAddr` against the `aAddr` of the record pointed to by `aRecPtr`. If `rAddr` is greater, start the scan

after this record; if it is less, start the scan at the beginning (`aHead`). If it is equal, and the matching record is to be freed, we already have the matching entry. Set `aRecPtr` to point to the last record inserted, or the record just before a newly deleted record.

### *Foreign node limitations*

Since we have within each data stream record the value of the block type, we can interpret what kind of block it is. But we cannot so easily find out what LA may have created it. The current version of Page K for the IRM can do so because it only displays memory blocks in the local node, where the `LATBL` is conveniently accessible. (We can also show a block allocated to hold executable code, since the `CODES` table is also quite conveniently accessible. But for the `vxWorks` case, such allocated executable blocks are handled by `vxWorks`, so we really don't know about them and they will not appear in the data stream records.) We cannot show the requesting node for a given allocated request block either, since we cannot easily access the contents of the request block. All of this means that we can do best in the case that the node queried is the local node. We may therefore want to special case it. If the target node is the local node, we can scan through `LATBL`, and we can look inside an allocated request block.

In either case, this new program cannot show the free blocks of memory. The reason is that `vxWorks` allocates memory for purposes for which we have no direct concern, so just because there appears to be a gap between two successive allocated blocks does not mean that the gap is free memory. Also, we do not know the size of the maximum contiguous free block under `vxWorks`, so the display will have to omit that info as well. If one uses `telnet` access to enter the `memShow` command, the number of allocated blocks might be 1100, typically, far more than are likely to be blocks that we allocated, perhaps by an order of magnitude.

What diagnostics might we include for this page application? Besides the execution time each 15 Hz cycle execution, we can keep a count of the number of data stream records captured each 15 Hz cycle, the current total `aRecActive` of active blocks in the linked list, the number of free entries available ( $= \text{aRecMax} - \text{aRecActive}$ ), error counts of the number of illegal block type codes encountered, the number of times a block to be inserted was already in the linked list, and the number of times the linked list was full.

### *Additional feature*

If it were useful to maintain a simple array of currently allocated blocks, one could easily be built by traversing the linked list and copying the information into the array. If this is cheap (in execution time) to do, it may be useful in making it that much easier to monitor allocated blocks in other nodes.

Since the time-of-day is included in the data stream records, we have the possibility to show it on the new Page K (`PAGEMBLK`) display. One would need to figure out how to fit it in, and also allocated another field in the array elements to house it.

### *Implementation for IRM*

With the design described in this note, it is clear that it does not mix well with the analog in the IRM. If one used the new Page K to target an IRM, we would not get any response, because there is no such data stream as `ALLOCLLOG` written to by IRM

system code. What would it take to do so?

All allocations and freeings of memory blocks would have to be corralled so that we could be sure to write a data stream record for each occurrence. The code at the lowest level is normally `AllocP`, which itself calls `Alloc`, a register-based entry point that actually does the pSOS `ALLOC_SEG` trap. It allocates from Region 1 memory, or from Region 2 memory if that fails. Similarly, the `FreeP` routine calls `Liber`, another register-based call that makes the `FREE_SEG` trap. As for the applications, most of them are Pascal programs that either call `Alloc`, which actually finds its way to `AllocP` in the system code via a special trap, or `New()`, which is implemented directly by code in `LASysLib`. It is too bad that the support for `New()` did not also find its way to `AllocP` in the system code, because it would make the job of capturing the application calls much easier. As it is, we might modify `LASysLib` to do it the right way, but we would also have to build new versions of all the applications, and install them, in order to get the updated library routine. Still, it may be worthwhile as a long term goal.

Suppose we have the data stream being written to by all the relevant parts of the IRM system code. Then a new Page K that works for the PowerPC should also be able to target the IRM nodes in the same way. Also, we should be able to write a new version of Page K for execution on the IRM. But the new one does not work exactly like the old (present) one, which also shows free spaces, since all allocated memory is understood by system code. The present one is called `PAGEMBLK`. If we wish to keep both versions available for the IRM, we would need a different name for it, such as `PAGEMEMB`.

### ***Post-implementation***

A new `PAGEMEMB` was written for the PowerPC along the lines described herein. At first, it failed to work when the target node specified was other than the local node. This was due to a bug in the generic name search support in the `ReqDGenP` system module. But it works ok for targeting the local node. Only allocated blocks are displayed, since we have no easy way to know about free blocks. This display shows only blocks that we allocate, not any that VxWorks allocates for its own use. Operating in test node0590, after quite some time, the list is about 8 blocks long only. Execution time is typically about 80 microsec, with occasional excursions beyond 1 ms.

A possible improvement, besides fixing the bug, is to show “friendly node numbers” for the requesting node of a request block. This may involve maintaining a cache of translated node numbers.

Sometimes this display that is updated at 15 Hz can be hard to follow visually, so a listing option is added that allows for a snapshot listing of allocated blocks.

Finally, the latest version includes friendly node#s, although they are only included for the case of monitoring the local node. This is not thought to be a severe restriction, as one can use Page G to reach any node and run the program there.

The final display format is as follows:

```

K MEMORY BLOCKS    07/13/04 1330
NODE<0590> *L<0509>    6 OFFS=  0
  ADDRESS SIZE TY DESCR  RNOD AGE
008E6588 0040 09 AHDRMSG    0
008E65D0 0050 09 AHDRMSG   13
008E6628 0030 01 LOCREQ   39
008E6660 0030 01 LOCREQ    0
008E6698 0018 0E INTPTRS   13
008E66B8 0048 0C ACREQ   AB86  13

```

The target node is followed by a listing node# to which a snapshot of the allocated blocks can be sent. It is initiated by a keyboard interrupt in the area of the listing node field. A count of the total #entries in the linked list follows, with the display window offset at the end of the line. Each line of the display shows the block address, its size, its type# in hex, some text description for that type#, the requesting (friendly) node# for a local request block case, and the age of the block in seconds. The age makes it clearer which blocks relate to each other. For a RETDAT request, for example, 3 blocks might be needed to support the request, all of them allocated at the same time. The largest age in the list is likely to be the request block that supports the collection of the data stream data from which the linked list and display are produced. This also gives a context for the significance of the other age values.

The meaning of a friendly node# is shown in this example by the value 0xAB86. The upper 4 bits value of A is meant to denote the Acnet protocol port#. (A value of C indicates the Classic protocol port#.) The low 12 bits indicate the Acnet trunk# and node# in hex. The trunk# is 11, and the node# is 134. To support this RETDAT request that has been active for 13 seconds, 3 blocks were created. One is the main request block; the second is the internal ptrs block; and the last one is the reply message block.

### Summary

This note describes the evolution of a new page application designed to show allocated memory blocks in a PowerPC-based node, based upon the existence of the ALLOCLOG data stream in the target node that holds records of each memory block that is allocated or freed during system operation. As a page application, it only shows memory blocks that are allocated since the "click" that initiated the activity. An option is provided to generate a snapshot listing of the blocks while the display is actively updating at 15 Hz. For the case of monitoring the local node memory block activity, the requesting node# is displayed for those blocks identified as Classic or Acnet protocol request blocks. If a local application is started up in the local node, the display indicates the name of the LA associated with the static memory block it allocates upon start-up. The age of each block in seconds is displayed to allow easy association of related blocks. The page application is written to run in both 68K and PowerPC systems. Until support for the ALLOCLOG data stream is implemented for IRMs, however, only PowerPC nodes can be targeted for display of allocated memory blocks.

### Addendum

Modifying the application to maintain the linked list in order of block age, not base address, makes it even more obvious which blocks might relate to each other. This was done in the final version of both MEMB and PAGEMEMB.