Technical Note PT505

A/ROSE & MCP Card Q&As

This Technical Note contains a collection of archived Q&As relating to a specific topic--questions sent the Developer Support Center (DSC) along with answers from the DSC engineers. Current Q&A's can be found on the Macintosh Technical Q&A's web site.

[Oct 01 1990]

A/ROSE memory requirements

How much memory is required in addition to normal application requirements for uploading A/ROSE to an MCP card? My faceless background task was unsuccessful at booting the Apple TokenRing card until the background application partition was increased.

___

With the release of A/ROSE 1.1.8, the download code is stored as a resource, instead of being linked statically with an application. This change in download architecture allows A/ROSE to be more flexible in terms of compatibility with A/UX and future operating system architectures. As a result, additional heap space is required for the download code resource to be brought into memory. The `'dwnl'` resource is about 9K. The memory required is about 9K + `sizeof`(largest segment of the task). There are probably a few other small factors that may add an additional couple of K.

If you encounter problems in this area, you might segment your code. The A/ROSE stuff, after being segmented, has a maximum segment size of about 14 or 15K. So, if your code is small, you may still need about 25K to download. If your segments are greater than 15K, that will determine how much memory you need.

A/ROSE product description

Date Written: 1/4/93

Last reviewed: 6/14/93

What's A/ROSE? The extension is on my Duo, and I'm not sure what it is, or whether I need it or not.

___

A/ROSE is a system extension which allows communication between the Macintosh motherboard and Macintosh Coprocessor Platform (MCP)-based NuBus cards which run the Apple Real-Time Operating System. Translated to English, this means that specific Macintosh NuBus cards, such as the Apple TokenRing cards and the latest Apple Ethernet NB card, have their own 68000 chip on board. A/ROSE is the driver which allows the Macintosh Operating System to communicate with the smart card, plus A/ROSE contains the operating system used to get the smart card up and running, along with any task information associated with the card. The only reason for having this INIT installed is to support an MCP-based card. On the Duo, it could be that the DuoDock has an Ethernet NB card installed. When not docked, A/ROSE still loads, but isn't used by the Duo. Other cards that require A/ROSE include the two Apple TokenRing, the Serial NB, and the Apple Coax/TwinAx NB cards. Note that SNA*ps is a software product that may require A/ROSE in specific configurations.

If the DuoDock doesn't have one of the listed cards installed, or your Macintosh Duo is not used for SNA*ps, then you can remove the A/ROSE startup document.

For more details, read "Inside the Macintosh Coprocessor Platform and A/ROSE" in issue #4 of *develop* (October 1990).

A/ROSE & MCP development questions

Date Written: 12/11/90

Last reviewed: 6/14/93

What restrictions (power, speed, and so on) does the Macintosh Coprocessor Platform (MCP) impose on the hardware added to the platform? Can we use A/ROSE on a board that we design ourselves from scratch? If so, we have the following questions concerning the hardware requirements for a board running A/ROSE: Must the CPU running A/ROSE be a 68000 or could it be a 68020 or 68030? Can A/ROSE handle master transfers under NuBus control between the card and main board memory? Is the source code for A/ROSE (and specifically the InterCard Communication Manager) available if we needed to tailor that code to our system?

___

To the knowledge of the A/ROSE engineer, A/ROSE requires modification in order to permit master transfers under NuBus control between the card and main board memory. The engineer succeeded in designing a routine to perform the write operation from A/ROSE; however, the read operation proved to be a challenge. A solution to your problem would be to release the source code. However, at this time this isn't possible. There has been consideration on this subject but no final decision.

In response to your other questions, MCP doesn't impose any restrictions on the hardware. There are some card dependencies imposed by A/ROSE, described in the document, "A/ROSE Card Dependency," available on AppleLink.

I'm familiar with a couple of cards based on the MCP board. On one card in particular, the processor was swapped with a

68020 using A/ROSE.

gCommon addresses and downloading card-dependent routines

Date Written: 8/14/91

Last reviewed: 6/14/93

After upgrading to A/ROSE 1.1.2, when I download to my board, it doesn't call my board-specific routines in the resource `'hlta'` that I installed into A/ROSE Prep.

___

Check the gCommon area on the card at locations `gCardInit0`, `gCardInit1`, and `gCardInit2`. These are the addresses where the Download subroutine would have placed the card-dependent routines when placing the routines on the card. You can find these addresses in the :A/ROSE:includes:arose.a header file.

If these addresses are zero, that means either the Download subroutine failed to find the card-dependent routines when doing the downloading, or the Download subroutine is the wrong version and doesn't know about card-dependent routines, or the application doing the downloading was not linked with a recent enough version of Download-Lib.o and contains the old Download subroutine code. The actual code for doing downloading has been moved from the application to the A/ROSE Prep file and glue code is now in the Download-Lib.o file, which is what the application really calls. This glue code knows how to load the actual downloading code.

If these addresses are nonzero, A/ROSE, during its initialization, should call these routines. If A/ROSE is not calling them, check to see where the A/ROSE kernel is coming from. Was the A/ROSE kernel linked in with the code? In this case, you need to relink your code with a more recent version of the A/ROSE kernel.

A/ROSE NetCopy process description

Date Written: 8/30/91

Last reviewed: 6/14/93

How does the NetCopy `cVirtualToReal` routine work in the oscglue.a file of the Portable A/ROSE code? I'm trying to emulate the `NetCopy` command in my A/ROSE code. I want the copy routine to work without requiring the user to use the `LockRealArea` command on the Macintosh Operating System side, but this means that I have to worry about the addressing being passed in as being virtual. Is the A/ROSE kernel on the board providing the virtual-to-real mapping or does it just send a message back to A/ROSE Prep to do it and then use the results?

___

What `cVirtualToReal()` does is take as input, a transaction ID (TID), a virtual address and a size. It returns a NuBus address and a size. If the returned size is zero, `cVirtualToReal()` could not find a corresponding NuBus address for the virtual address. If the returned size is non-zero, this size is the size of the contiguous address range for the virtual address starting at the NuBus address.

This is how `cVirtualToReal()` works: `cVirtualToReal()` examines the TID to find out the slot (call it slot s) associated with the TID. `cVirtualToReal()` looks at the iccm communications area of slot s looking at `MapCnt`, `MapPtr`, and `MapChk`. (Please refer to iccmDefs.h or iccmDefs.a in the iccm communications area record ca_Rec.)

If `MapCnt` is zero, the virtual address is assumed to be the same as the NuBus address. In this case, `cVirtualToReal()` sets the returned NuBus address value to be that of the virtual address and returns the size the same as the size on input.

If `MapCnt` is nonzero, the slot either has an address scheme where the virtual address that the TID on slot s uses cannot be the same as the NuBus address (say the processor was a 68020 with no PMMU and required that local card addresses have zero for the high address byte) or the slot has something like VM (which is what happens on the main logic board). `cVirtualToReal()` will use the address map pointed to by MapPtr on slot s to try to resolve the virtual address to a NuBus address. MapPtr is a pointer to a linked list of struct AddrMap (defined in os.h or os.a). If `cVirtualToReal()` can resolve the virtual address by looking at the `AddrMap` entries, it will do so. If `cVirtualToReal()` cannot, it will return zero for the size. The `MapChk` and `MapCnt` variables in `ca_Rec` are used for concurrency. `cVirtualToReal()` will fetch a pointer from the `MapPtr` structure and then check to see if `MapChk` has changed. `cVirtualToReal()` knows that the pointers in `MapPtr` and the AddrMap entry have not changed if MapChk has not changed. `cVirtualToReal()` starts its search over at the beginning of the `MapPtr` list should `MapChk` change. `cVirtualToReal()` knows that it can start searching the `MapPtr` list only when MapCnt equals `MapChk`.

What `cLockRealArea()` does is take as input a virtual address and size and returns on output the NuBus address/length pairs associated with that virtual address and size. `cLockRealArea()` also returns a success/failure status indicating if `cLockRealArea()` successfully "locked" the pages in memory associated with the virtual address and size.

Please note: There is a `cLockRealArea()` on the main logic board used to lock memory on the main logic board. There is also a `cLockRealArea()` in the A/ROSE kernel on the NuBus card which WOULD lock memory on the NuBus card. The `cLockRealArea()` on the NuBus card acts as a no-op at this moment and is present for compatibility reasons.

Here's a description of how `cLockRealArea()` works: `cLockRealArea()` on a NuBus card acts as a no-op. `cLockRealArea()` on a NuBus card returns a NuBus address/length pair as one would expect but otherwise does nothing.

`cLockRealArea()` on the main logic board is much more interesting. `cLockRealArea()` on the main logic board tries to lock down the pages associated with the virtual address/size (assuming a PMMU is present and the appropriate a_line traps are present) calling either `_LockMemory` or `_LockMemoryContiguous`. Next, if `_LockMemory` or `_LockMemoryContiguous` works, `cLockRealArea()` tries to create an AddrMap entry to store the NuBus addresses associated with the virtual address. `cLockRealArea()` needs to call `_LockMemoryContiguous` on this AddrMap entry to lock it down as well. Finally, if `cLockRealArea()` has made it this far, `cLockRealArea()` will try to insert the AddrMap entry into the linked list pointed to by `MapPtr` in the iccm communications area ca_Rec. `cLockRealArea()` will change `MapChk`, insert the AddrMap entry, and then change MapCnt to be the same value as `MapChk`.

What `cNetCopy()` does is take as input a source TID/source address, destination TID/destination address, and size. `cNetCopy()` calls `cVirtualToReal()` to convert the source TID/source address and the destination TID/destination address into NuBus addresses. If `cVirtualToReal()` can convert the addresses, `cNetCopy()` will perform a blockmove() operation moving the data itself. If `cVirtualToReal()` cannot convert the addresses, `cNetCopy()` will send an A/ROSE IPC message to the NuBus slot (please note that the main logic board is NuBus slot zero) asking the iccm (please note that iccm is built into the A/ROSE driver on the main logic board) to perform the copy. Iccm will perform the copy and send a reply back to `cNetCopy()` when the copy is complete.

At this moment, the only slot that can have virtual addresses that `cVirtualToReal()` cannot convert to NuBus addresses are main logic board addresses. `cVirtualToReal()` can always resolve virtual addresses that exist on NuBus cards.

If the user on the Macintosh Operating System side does not use `LockRealArea()` and PMMU is believed to be active, an A/ROSE IPC message is sent from the NuBus card to the iccm portion of the A/ROSE driver on the main logic board to perform the copy. The iccm portion of the A/ROSE driver will send back a reply when it has performed the copy.

One unasked question is could the card request that memory on the main logic board be locked down from the card. The answer to this question is NO. The `LockRealArea()` routine on the main logic board that does the locking down of memory on the main logic board associates the memory that it is locking down with the TID of the main logic board process that is calling `LockRealArea()`.

NuBus '90 and MCP card timing

Date Written: 8/28/91

Last reviewed: 6/14/93

The Macintosh Coprocessor (MCP) card uses the NuBus 10 MHz clock for the system clock and the NuBus'90 clock is 20 MHz, so how can the cards continue to work? The 68000 isn't rated for 20 MHz. Is there a new pinout for NuBus'90 that provides both clocks?

___

The NuBus'90 CLK signal (pin C32) is 10 MHz (didn't change); the 20 MHz clock signal is the CLK2X signal, which is pin B24 (previously -5.2 Volts).

Dynamically downloaded A/ROSE task specifications

Date Written: 2/25/92

Last reviewed: 6/14/93

How does a dynamically downloaded A/ROSE task specify its stack size, heap size, execution priority?

___

You'll note from the description of the `DynamicDownload` function as described in the *Macintosh Coprocessor Platform Developer's Guide,* page 8-8, that a pointer to an `RSM_StartTask` parameter area is required. The caller must initialize the following fields:

```
st_parmblock->stack
st_parmblock->heap
st_parmblock->priority
```

The dynamic download process uses these fields of the RSM (Remote Start Manager) parameter block when initializing an A/ROSE task.

Dynamically downloaded A/ROSE task startup

Date Written: 2/25/92

Last reviewed: 6/14/93

When an A/ROSE task is downloaded dynamically, how and when is it started up?

___

The quick answer is that such a task is started before the download routine returns. A more complete answer can be made by perusing the Portable A/ROSE code available on the Developer CD. When a dynamic download call is processed, some preflighting work is performed by the download routine in the A/ROSE Prep file. A/ROSE first checks that a card capable of running A/ROSE exists in the designated slot.

A/ROSE then checks via the Remote System Manager (RSM) to determine if there is sufficient memory on the target board to process the download. The memory is allocated via RSM for the new task, and a copy of the `ST_PB` is made into the user area, for processing the download. The register blocks are then initialized. A message is sent to RSM to perform the `RSM_StartTask` procedure. When RSM processes the message, a call is made to NetCopy to copy the ST_PB record onto the card. There's some more preflighting of the `ST_PB`, which results in a call to the `StartTask` Instruction Trap. `StartTask`, as shown in the file ostrap.a from Portable A/ROSE, ensures that the call did not originate from an interrupt. The task table is searched for an available entry. When one is found, memory is allocated for the process, heap, and stack if specified. Some additional processing is performed to set up the task. Finally, the task is linked into a priority table and the task becomes "live."

As documented, the download process returns the TID of the task, if successful. It should be noted that while the task has been started, the task still needs time to perform initialization and any Name Manager registration before a `LookupTask` call is made.

The code available on the Portable A/ROSE disk is a good source for learning the details on this function.

FreeMem and FreeMsg don't set pointers to zero

Date Written: 2/25/92

Last reviewed: 6/14/93

Do the A/ROSE functions FreeMem and FreeMsg set the pointers to zero or does an A/ROSE task calling these functions have to do so explicitly?

___

FreeMem and FreeMsg do not set the pointers to zero upon return. Your process must do so explicitly if you plan to check whether a message block is valid.

Net_Register_Task same as Register_Task(..., World_Visible)

Date Written: 2/25/92

Last reviewed: 6/14/93

What's the difference between the A/ROSE Net_Register_Task and Register_Task(..., World_Visible)?

___

There's no difference between the two calls. When A/ROSE was first being developed, a design specification under investigation was to have NetCopy perform as its name suggests - to copy data across the network. This feature proved difficult to implement. The functions became limited to copying data between the motherboard and installed NuBus cards. To transmit A/ROSE messages across the network, refer to Chapter 11 in the *Macintosh Coprocessor Platform Programmer's Guide* for a description of the Forwarder mechanism and a sample program demonstrating this capability.

A/ROSE mCode value isn't incremented for unknown messages

Date Written: 2/25/92

Last reviewed: 6/14/93

With an unknown A/ROSE message, is mCode incremented by 1 before sending it back to the sender (assuming, of course, the current task is not the sender)?

___

For unknown messages, the mCode value is not incremented. The message code is simply OR'd with 0x8000.

Reference:

Macintosh Coprocessor Platform Programmer's Guide, page 3-15

Setting A/ROSE mCode or mStatus on receiving a message

Date Written: 2/25/92

Last reviewed: 6/14/93

Should an A/ROSE server-like task (a task that loops waiting for messages, but doesn't originate messages) switch on mCode or mStatus when a message arrives? In other words, which should be checked first when getting a message--mCode or mStatus? In addition, if mCode is a known code, but the status is nonzero, should the message be discarded (ignored)?

___

Your program could certainly do either; however, it would seem more normal to switch on mCode first, then take a look at the status. This style corresponds to the layout of the Programmer's Guide and would make the code easier to follow by someone else.

As for discarding the message, the question posed is not clear. If mStatus is nonzero, some error has occurred and needs to be dealt with. However, other processing could be performed with the same message block allocated. For example, you might have a loop that queries the existence of the ICCM on each card. If an error is returned, the task would know that there is no Macintosh Coprocessor Platform card in the designated slot, or that A/ROSE has not been initialized on the installed card. The task would increment the slot counter variable and continue to query the next card. When the loop is finished, a FreeMsg call would be made.

NetCopy & mDataPtr buffer address if A/ROSE IsLocal TRUE

Date Written: 2/25/92

Last reviewed: 6/14/93

If IsLocal returns TRUE, can a task avoid calling NetCopy and access the mDataPtr buffer address directly and at will?

___

How about a yes/no answer. If IsLocal returns TRUE, then your process can bypass the NetCopy procedure; however, you cannot access the mDataPtr buffer address directly without first checking that you are dealing with a physical address on the main logic unit. For example, your process may be running on a Macintosh IIci or IIsi, which have "interesting" memory maps where the virtual addresses do not correspond to the physical ones. In such cases, refer to the Technote "Coping with VM and Memory Mappings."

A/ROSE LockRealArea call

Date Written: 2/25/92

Last reviewed: 6/14/93

When does the A/ROSE LockRealArea call really need to be used? Is it compatible with virtual memory and System 7.0?

___

LockRealArea is provided as a Prep routine and is used by routines on the motherboard to prevent System 7 virtual memory from paging an important buffer out from under a process like NetCopy. LockRealArea was designed for compatibility with virtual memory and System 7.0.

`LockRealArea` performs two functions. First, it makes it possible for a `NetCopy` call made by any NuBus card task to access the main logic board memory. Secondly, it calls either the Macintosh Operating System virtual memory dispatch trap `_LockMemory` if `LockRealArea` was called with a physical address/length pair count greater than one, or `_LockMemoryContiguous` if LockRealArea was called with a physical address/length pair count equal to one.

`_LockMemory` and `_LockMemoryContiguous` in turn perform two functions. Both requests ensure that the virtual memory being locked is placed in physical memory and is not pageable. Secondly, both requests make the physical pages of the physical memory noncacheable. This second attribute takes care of the "stale" data cache problem that can occur with 68030 and 68040 CPUs.

Unregistering A/ROSE task names

Date Written: 2/25/92

Last reviewed: 6/14/93

When an A/ROSE task quits, is it more efficient to unregister that task's names with the Name Manager, or just let the Name Manager take care of it automatically?

___

For tasks running on cards, A/ROSE will automatically unregister all names associated with a given task when either StopTask is explicitly called or your tasks main() [as specified in the StartTask call] is returned from.

For tasks running on the motherboard, A/ROSE will *not* automatically unregister the names. This is because the task is an application, driver, VBL, etc., first, and an A/ROSE task second. In all cases it isn't clear when the task is finished. It's the task's responsibility to unregister names associated with tasks running on the motherboard.

In regard to efficiency, the answer is that it is more efficient for the task to allow the Name Manager to unregister the names for it. The Unregister function will be called in all cases at termination (even if you explicitly unregister), so a duplicate function call can be avoided. Again, this *only* applies to tasks on the card! On the motherboard you must *explicitly* call `Unregister`.

A/ROSE task interrupts

Date Written: 2/25/92

Last reviewed: 6/14/93

What can or cannot be done at interrupt time in an A/ROSE task or an A/ROSE process running on the motherboard?

___

On the motherboard, you must refrain from doing anything that results in memory movement or allocation. In addition, you should not call `KillReceive` from an interrupt routine. A race condition could result --the `KillReceive` may not get done or done at the wrong time.

On the card, you should not call `StartTask`, `StopTask`, or `Receive` from an interrupt routine. Please note that `RegisterTask`, `LookupTask`, and `printf` call `Receive` in their code, and as such, cannot be done from an interrupt routine.

Here are some additional caveats that you should be aware of with regard to ISRs (Interrupt Service Routines):

On the card, you probably do not want to call `GetTID`. If you call `GetTID` during interrupt time, you'll get the `TID` of the currently executing A/ROSE task, which is probably *not* the `TID` of the A/ROSE task which installed your interrupt routine. It is best for your task which installs your interrupt routine to save the TID in a memory location for your interrupt routine.

On the card, you should note that if you do a `GetMsg` request, the `mFrom` field in the A/ROSE message will not be what you expect. You might expect the `mFrom` field to be your task which installed the interrupt routine. The `mFrom` field will be set to what `GetTID()` would return.

One final caveat and this one is hard to explain. This caveat concerns performance. To achieve a fast level of real-time latency, the card's primitives, like `Send` and `GetMsg` and `FreeMsg`, try to disable interrupts for as short a time as possible. However, if an interrupt routine calls `Send`, `Send` *cannot* lower the interrupt level below whatever called it. For example, suppose an interrupt routine at interrupt level 2 calls `Send`; `Send` *cannot* lower its interrupt level below 2 to zero; an interrupt at level 1 or another interrupt at level 2 might be pending, causing confusion for the interrupt routine. It's best for the interrupt routine to lower its interrupt level when safe before calling `Send`.

Should the interrupt routine need an A/ROSE message buffer, it's best for the task that installed the interrupt routine to preallocate message buffers by calling `GetMsg`, refill the supply as needed, and place pointers to these message buffers in an area for the interrupt routine to use. Normally, the interrupt routine will want to send an A/ROSE IPC message to the task to wake it up.

DTS recommends the following logic for such a wakeup scenario:

1. Have the task do a `GetMsg`, set up this message so the mTo field is that of this task and this message has a task-defined `mCode`, and save the address of the message buffer in a variable called `messagePointer`. This effectively creates an A/ROSE IPC message that you will recognize as the wakeup message from the interrupt routine.
2. Have the task install the interrupt routine.
3. When the interrupt routine is invoked, have it do whatever is necessary at its interrupt level. Then, have the interrupt routine get the value from `messagePointer` into a register (like D1) and zero `messagePointer`. Have the interrupt routine lower its interrupt level to the previous interrupt level found on the stack. If D1 is zero, you're done. Exit the interrupt routine by doing `JMP PostRTE`. If D1 is nonzero, D1 contains the address of the wakeup message; call `Send` to send the A/ROSE IPC message to wake up the task.
4. When the task receives the wakeup message, which it recognizes by the `mCode`, the task can store the address of the wakeup message in `messagePointer`. Then, the task can look for information left by the interrupt routine on what the task should do. The task must be prepared to find multiple things to do or nothing to do.

Back to top

Downloadables

Acrobat version of this Note (K). Download