Mica Working Design Document
Image Activation

Revision 1.2

8–February–1988

Issued by:

Lou Perazzoli

digital™

TABLE OF CONTENTS

FIGURES

Revision History

Date	Revision Number	Summary of Changes
30-May-1986	0.0	Initial entry. (Lou Perazzoli)
9-Sep-1986	0.1	Changed to reflect new improved map_file directive. (Lou Perazzoli)
25-Sep-1986	0.2	Incorporated review comments and changed name to image mapper. (Lou Perazzoli)
04-Nov-1987	1.0	Eliminated moldy P.TBD concepts such as environments, map_file directive, etc. and changed to reflect current linker and memory management designs. Add sections on autoloader and synchronizations. Added section on shareable image loading and initializations. (Lou Perazzoli)
25-Nov-1987	1.1	Minor edits to conform with object language and linker chapters. (Lou Perazzoli)
01-Feb-1988	1.2	Add section on thread local storage and modify autoloader design. (Lou Perazzoli)

CHAPTER 1

IMAGE ACTIVATION

1.1 Overview

The linker produces an executable image as the end product of program development. During process creation, the thread creating the process specifies the image to be executed by the new process. After the creation of the process and the initial process thread, the image file is mapped into the newly created address space. This mapping occurs in the context of the initial process thread.

Image mapping involves several steps that prepare the image for execution. The image activator opens the image file, thereby establishing a channel to obtain the necessary information to map the file. If the image does not already have an associated segment object, the image activator creates a segment object for the image, building prototype PTEs for the image file. The image activator maps the image into the user's address space, resolves certain address references, and establishes the debugger and traceback handlers.

1.1.1 Goals/Requirements

The Mica image activator has the following goals:

- All images are automatically and transparently shared among all users.
- Optimal performance is achieved by issuing a minimal number of disk reads to initially map the image and delaying most fixups by delaying the loading of shareable images.

1.1.2 Functional Description

1.1.2.1 Image Initialization

No special code exists in Mica to read images into memory for initial execution. Instead, the paging mechanism is used to "page" an image into memory. The image activator configures the process page tables to reflect all pages in the image file.

Mica performs the following steps to support image activation:

1. Opens the image file

   The image activator issues a read-only share open service on the image file. This service returns a channel ID to the file.

2. Creates a section

   The image activator calls the *exec$create_section* system service. The caller specifies the channel ID from the previous system service call, and a mapping type of *e$k_image_map*. This service returns a *section_id*.

3. Maps the section

   The image activator calls the *exec$map_section* system service, specifying the *section_id* returned from the *exec$create_section* system service. This service returns the starting and ending addresses that delimit the mapped image in the virtual address space.

4. Performs fixups

   The starting address identifies where the image's image header begins. The image activator examines the image header, and performs the necessary image fixup operations on the image.

5. Handles message sections

   If any message sections are present in the image, as indicated by the image header, the image activator calls the routine to add these message sections to the process. The nature and functions of this routine are described in Chapter 3, Status Codes and Messages.

6. Maps shareable images marked "activate immediately"

   The image activator examines the image header, and maps any shareable images which are marked "activate immediately". The image activator performs the external fixups for those shareable images once they are mapped. Note that this is a recursive call to the image activator.

7. Calls initialization procedures

   Once the image activator maps and fixes up the "activate immediately" images, it examines the image header, and calls any initialization procedures at their specified entry points. These initialization procedures provide the functionality of the LIB$INITIALIZE routine in VMS. The image activator does not guarantee the order between images of initialization procedure calls, but it does guarantee each procedure is called only once before the user executes any code within that shareable image.

8. Invokes image

   After the image activator has invoked all initialization procedures, it calls the image at its transfer address.

1.1.2.2 Image Exit

When an image returns to the image activator, the *exec$thread_exit* system service is issued, which begins thread termination. The *exec$thread_exit* system service simply calls each exit handler that has been declared by the thread, and then invokes the *exec$delete_thread* system service. If the process has other threads executing, those threads continue to execute normally. Image exit occurs when the last thread in the process exits and the mapping objects and section objects for the image are deleted during object container rundown.

1.1.2.3 Autoload Procedure

The autoload procedure operates similarly to the "activate immediately" method described above. The autoload procedure loads shareable images and resolves the external references when the reference is encountered, rather than at initial image activation time. This reduces the overhead of initial image activation, and maps shareable images only when they are actually required.

1.1.2.4 Installation of Images

The Install Utility serves two purposes. It provides for the installation of a shareable image:

- Within the shareable image space
- With the WRITE attribute

The Install Utility creates a section object for the image, which causes a segment object to be built. The segment object has a "system channel" to the image which implies that the image is effectively installed "opened".

1.1.2.5 Images Within Shareable Image Space

When a shareable image is installed in the shareable image space by use of the /BASE qualifier, the Install Utility opens the image file, and creates a segment causing prototype PTEs to be built. The segment object for the shareable image contains the base address for the image within the shareable address space. When the shareable image is loaded, it is mapped at the specified address. If the image cannot be mapped at the specified address due to addressing conflicts, an error is returned, and the shareable image is not mapped.

When the shareable image is installed no fixups, internal or external, are performed. Note, however, that since no external fixups are performed, any referenced shareable images are treated just like referenced external images. This allows later versions of shareable images to be installed at different base addresses while the system is running, and the latest image is properly loaded.

Images installed in shareable image space may reference other images though use of the autoload capability or the activate immediately capability. These referenced images do not need to reside in shareable image space.

1.1.3 Issues to be Resolved

- Exact detail of message section addition. This is dependent on the design of message sections and the definition of the routines.

1.2 Image Activation

1.2.1 Thread Startup

Once the process and the initial thread have been created with a minimal address context (user stack, process control region, thread control region, last chance condition handler), the initial thread startup routine is invoked in user mode. The initial thread startup routine exists in the system portion of the address space. Its sole purpose is to map the shareable image, *mica$fm_share*, that performs the actual user-mode thread startup.

A section exists for the *mica$fm_share* shareable image in a known system container. To map the image an object name translation is performed on the image name in the system container. The resulting section ID is mapped and its base address in shareable image space.

The *mica$fm_share* shareable image contains the image activator, the image fixup, the global autoloader, and the support routines to start the initial image.

After mapping the *mica$fm_share* shareable image, the initial thread mapping procedure examines the image header to locate the transfer vector offset for the *imact$initialize* procedure. The image activator then calls the *imact$initialize* procedure, passing the base of the system service vector page as an argument. The base of the system service vector page is required to allow various system services to be called before the system service shareable image has been loaded.

The *imact$initialize* procedure maps the section for the system services, and opens the specified image file as execute only.

If the open service on the image file succeeds, the *exec$create_section* service is called to create the structures necessary to share the file among multiple address spaces.

The *exec$create_section* service performs an object name lookup to determine if the specified image file currently has an associated segment object. Note that all segment objects exist in the same system container. If the segment indicates that the file is mapped as a data file rather than an image file, the *exec$create_section* service returns an error indicating that the file is mapped in an incompatible state. If the segment object specifying image mapping currently exists for the image file, a section object is created which refers to the segment object. If the segment object exists, the image activator continues the process of image activation by mapping the section (See Section 1.2.3). Otherwise, memory management software must create the segment (See Section 1.2.2).

1.2.2 Segment Does Not Exist for Image File

If a segment object does not currently exist for the image, the *exec$create_section* system service creates and initializes a segment object for the image. A system channel is created to the specified file, and the pointer to the channel object is stored in the segment object.

The segment object header contains, among other items, the system channel pointer of the file, the number of pages in the segment, the required size of the user stack, and the mapping type (in this case, *mm$c_image_map*).

1.2.2.1 Image Header

The first sixteen virtual blocks of the image are read into memory and examined. If the image is less than 16 blocks, an error status is returned in the IOSB, but the read still completes, delivering all the blocks in the image. The image header (which is at least one block) contains an item describing the number of 64-Kbyte pages required to map this image. The number of 64-Kbyte pages determines the allocation required for the segment object.

1.2.2.1.1 Image Section Descriptors

The analysis of the image header continues by examining the image section descriptors (ISD). The image section descriptors describe the layout for creating the prototype PTEs for the image. Each image section is aligned by the linker on a virtual disk block number (VBN). An image section descriptor starts on a 64-Kbyte virtual boundary. The image section is either demand zero, or it contains the number of VBNs in the section, and the starting VBN number of the section. The ISD also contains the page protection for the section. See Chapter 29, Linker for more details.

The image activator determines the format and the protection attributes of the prototype PTE from the information in the ISD. Example 1–1 describes how the protection is set.

Example 1–1: Prototype PTE Protection Attributes

Flags in ISD			Settings in Prototype PTE					
READ	WRITE	EXECUTE	READ	WRITE	FOR	FOW	FOE	COM
0	0	0	0	0	0	1	0	1
0	0	1	1	0	1	1	0	1
0	1	0	Invalid in PRISM					
0	1	1	Invalid in PRISM					
1	0	0	1	0	0	1	1	1
1	0	1	1	0	0	1	0	1
1	1	0	1	1	0	1	1	*
1	1	1	Invalid in an image file					

* Set if MODIMG$IMAGE_SECTION_DESCRIPTOR.COPY_ON_MODIFY is true.

The *copy_on_modify* flag is set on all nonwritable pages. This allows the protection on a nonwritable page to be changed to writable, causing the materialization of a private page before the actual write. Also, fault on write is enabled on all pages even if those pages are not writable. This allows the protection to be changed to writable without having to also enable fault on write. For ISDs which are read/write and not copy on modify, fault on write is enabled to maintain the modified state of the page.

When a writable image section descriptor is encountered without *copy_on_modify* set, the file system is queried to see if the user has opened the file for write access. If the file has not been opened for write access, the *exec$create_section* service fails indicating the file was not opened for write access in the status value. If the file was opened for write access, the *exec$create_section* proceeds. Only a shareable image may have image sections without *copy_on_modify* set.

Once the stack descriptor has been found, the size of the stack is recorded in the segment object.

Once the segment object has been initialized, the section object is created in the process-private container that refers to the segment object.

At this time, the section object and segment object for the file have been created. All local image sections have been processed and the prototype PTEs have been created for the segment. It is interesting to notice that at this time all prototype PTEs either point to a subsection, are no access, or are demand zero.

1.2.3 Mapping the Image into Virtual Address Space

The *exec$map_section* service is issued to map the section into the user's address space. This causes the creation of a mapping object in the process-private object container. The mapping object contains a reference pointer to the section object and the virtual address limits where the section was mapped.

The image activator attempts to map images at their based address to eliminate internal fixups.

If the initial user stack is less than the stack size specified in the segment object, the *exec$expand_stack* service is called to produce a stack of the proper size.

1.2.3.2 Image Fixup

An image section contains the fixup information necessary to resolve any internal image addresses. Once the *exec$map_section* code completes, the image fixup is done in user mode by examining the fixup information, and adding the difference between the real and the base address of the image to each fixup location.

If necessary, the protection of the page is changed to writable in order to allow fixup information to be written to the page. By examining the fixup information, an argument list for the Set Protection service is created which sets the protection of all necessary pages, that is, pages which reside in nonwritable program sections, to write enable. The fixups are then calculated and written to the appropriate locations. The protection on the pages is changed back to write disable.

1.2.3.3 Thread Local Storage Fixups

The shareable image *mica$fm_share* contains the current sum of all thread local storage regions in each loaded image. When an image is loaded via the activate immediate mechanism or the autoloader, the image being loaded is given the current sum, and the sum is updated by adding the number of thread local storage regions found in the image being loaded to the current sum.

The accessing and modification of the sum is performed under synchronization to prevent multiple threads from storing or updating the sum simultaneously.

1.2.3.3 Activate Immediately Shareable Images

After completing internal fixups, the image activator examines the image header to locate any "activate immediately" shareable images. If found, the image activator issues a call to an object service to determine if a mapping object exists with the name of the image desired. If the image has been mapped previously, the virtual addresses of where it is mapped may be obtained from the mapping object. The virtual address of the shareable image is used to perform the external fixups. Note that the linker determines whether a shareable image is autoloaded or "activate immediately".

If an "activate immediately" image is not currently mapped, the shareable image synchronization primitives described in a later section are used to ensure that only one copy of the shareable image is mapped, fixed up, and initialized.

As each shareable image is mapped and fixups performed as necessary, any initialization procedures for that shareable image are invoked. As "activate immediately" image operations are performed, external fixups are performed in the invoking image.

1.2.3.4 Debugger

The debugger is invoked as an activate immediately shareable image which has an initialization procedure. The debugger's initialization procedure checks to see if the debugger should be invoked and if so does whatever operations are necessary to create a debugging environment.

1.2.4 Loading of Shareable Images

The following pseudo-code describes the steps to ensure that two threads do not attempt to load the same shareable image at the same time.

```
! General shareable image synchronization.
disable ASTs
status = exec$create_event (name = shareable_image_name)
IF status == collision THEN
    status = exec$wait_any (collided event_id)

    ! Shareable image is loaded and initialized when wait
    ! completes. Note wait could return an invalid ID error.

END IF

status = exec$translate_object_name (
        object_id = returned_mapping_id,
        object_type = mapping_object,
        object_name = shareable_image_name
        )

IF status == name_does_not_exist THEN

    ! Image has not been loaded. Load it.

    open_file (shareable_image)
    create section (shareable_image_channel)
    map section (shareable_image_section_id)
    fixup shareable image
    activate immediates
    call initialization procedures

ELSEIF status == success THEN

    ! Shareable image is loaded and initialized.

    exec$get_mapping_info (
        object_id = returned_mapping_id,
        item = base_address
        )
ELSE
    restore AST state
    ! Unexpected error - raise condition.
END IF

store the base_address
status = exec$set_event (event_id)         !ignore any errors
status = exec$delete_object_id (event_id)  !ignore any errors
restore AST state
```

1.2.5 Autoloading of Shareable Images

Automatic loading of shareable images is the act of loading a shareable image only when a procedure within that shareable image is invoked. This allows the overhead of loading the shareable image and performing fixups and initialization routines to be deferred until the shareable image is actually required.

When the first call to a procedure within a shareable image is made, that image is automatically loaded. All other calls to procedures within that shareable image from the image making the call are fixed to directly call the loaded shareable image. Note that this involves changing the protection of read-only memory to read-write, and then changing it back again.

1.2.5.1 Linkage Pair

A *linkage pair* consists of two longwords. The first contains a pointer to the invocation descriptor for the procedure and the second contains the address of the code for the procedure. The following instructions are generated to call the procedure:

```
LDQ     routine(Rx),R10   ;load R10 with invoc. desc and r11 with code address
JSR     R11,(R11)         ;call the procedure
```

Note that routines which access the linkage pair must always be accessed with a LDQ instruction. This prevent synchronization problems when the linkage pair is being modified.

For procedures which reside in automatically loaded shareable images the first longword of the linkage pair contains the address of the autoload vector and the second longword contains the address of the transfer code. The linkage pair and autoload vector reside in the linkage section, which is read-only.

1.2.5.2 Autoload Vector

The autoload vector is the data structure that maintains the information about the automatic linkage to a routine in another shareable image. The autoload vector consists of 5 longwords, and is quadword aligned. The first longword is the address of the transfer code, the second longword is the address of the entry descriptor for the transfer code, the third longword contains the address of the autoload vector, the fourth longword contains the address of the autoloader code, and the fifth longword contains the address of the image descriptor for this shareable image. The third and fourth longwords are changed after the image is loaded to contain the address of the routine's real invocation descriptor and entry point. This allows subsequent calls to the routine using the autoload vector to work (which might happen in an optimizing compiler).

For each procedure called within a autoloaded shareable image a autoload vector exists. The autoload vector is created by the linker in the image's header, which is read-only memory. When the desired image has been loaded and all fixups and initializations performed, the autoload vector is modified. The address of the autoload vector field and the pointer to the local autoload code are changed to be the real routines linkage pair using a STQ instruction.

1.2.5.3 Transfer code

The transfer code consists of the following 3 instructions:

```
LDQ     8(R10),R4         ;load address of autoloader code
OR      R4,R0,R10         ;load R10 with address of autoload vector
JSR     R0,(R5)           ;jump to the local autoloader
```

Since the autoload vector itself is fixed up during automatic image loading, the transfer code on subsequent calls transfers the caller directly to the called routine. The transfer code is generated by the linker in a special psect that allows execution.

1.2.6 Image Descriptor

The image descriptor consists of the image name, the self-relative base address for the current image, and an array of longwords consisting relative pointers to each linkage pair which references procedure in the shareable image described by this image descriptor. Note that the image descriptor is created by the linker in the image header in read-only memory.

Figure 1–1: Structures Before Autoload Has Occurred

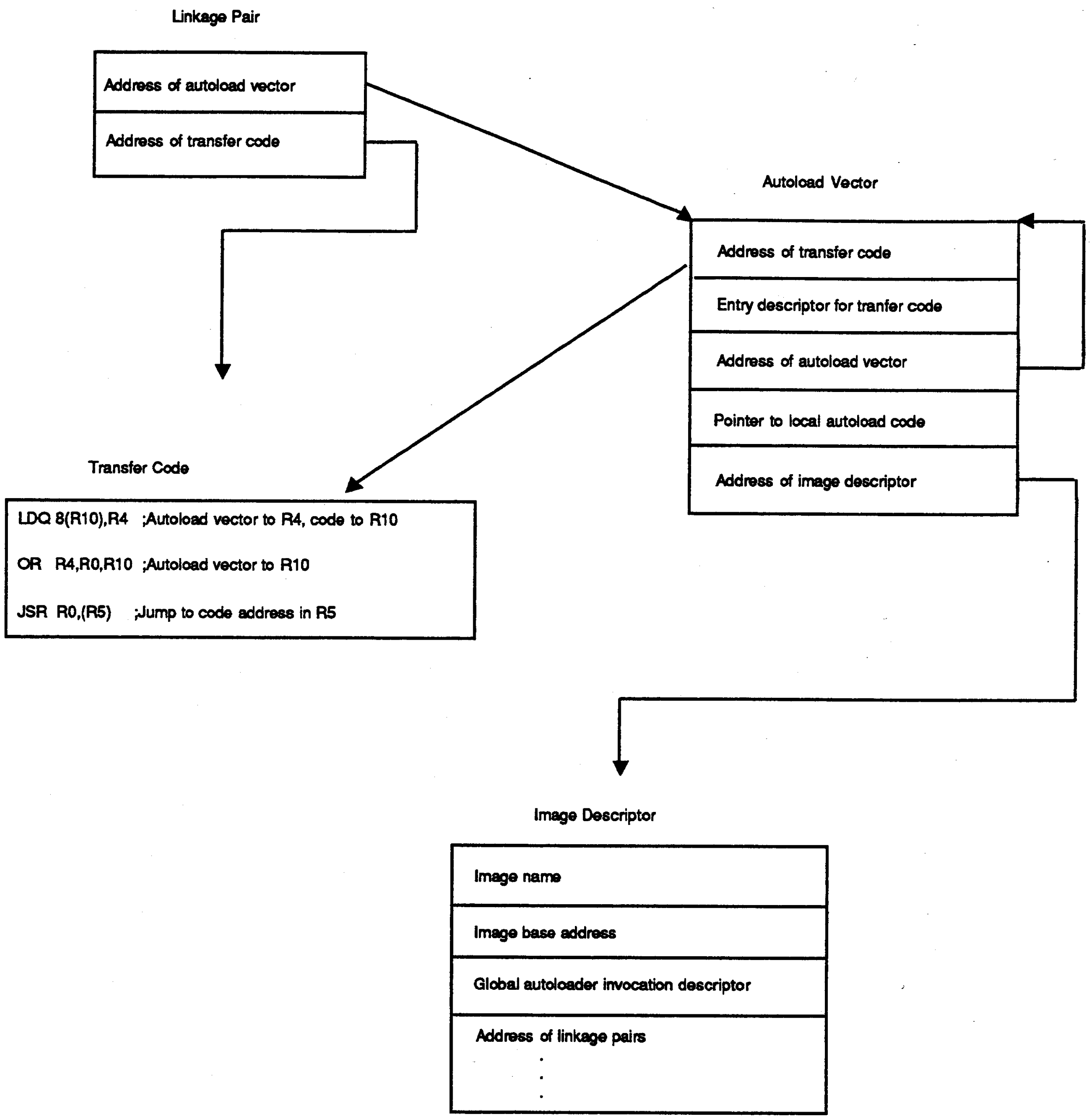
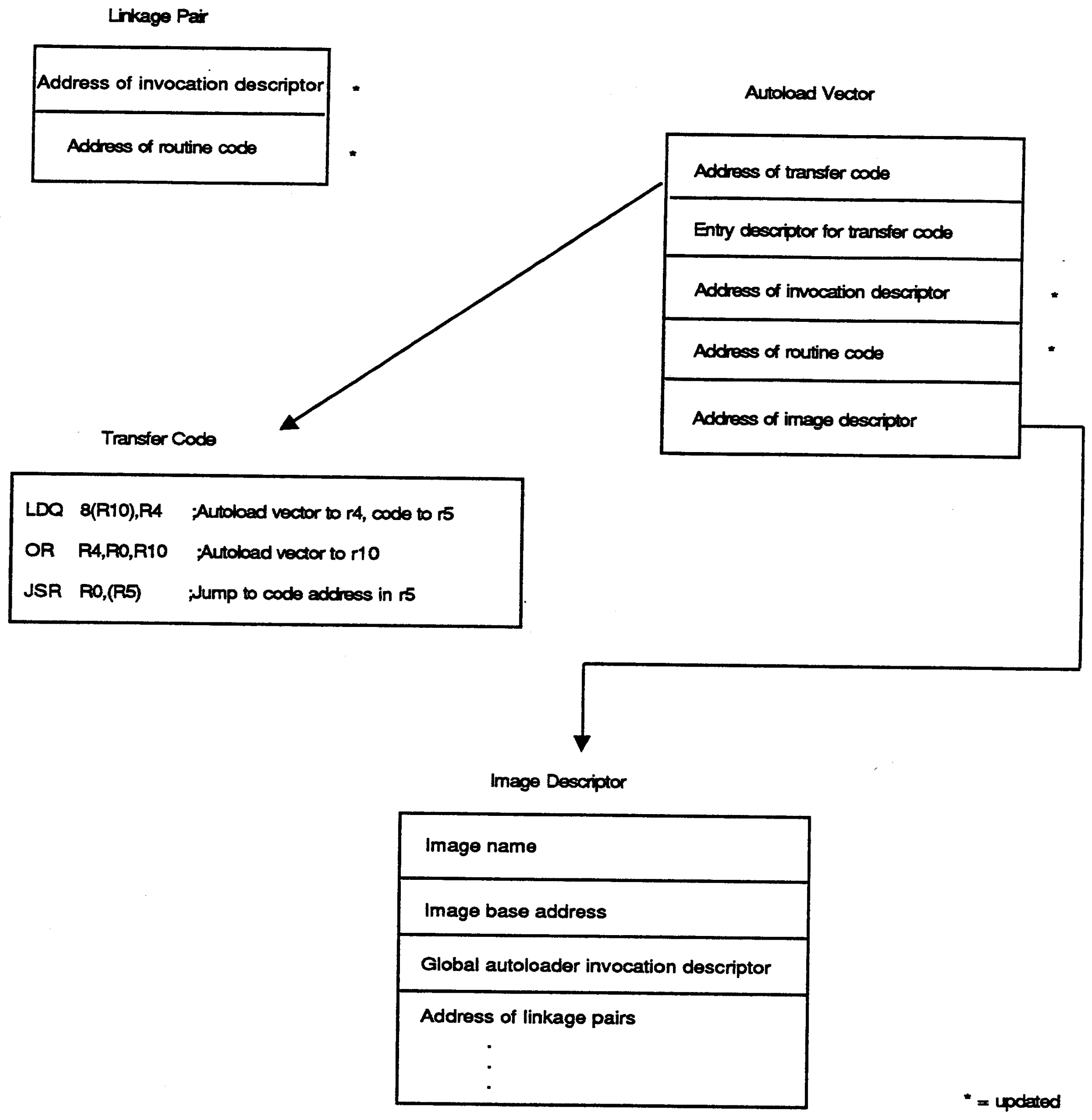

Linkage Pair
Address of autoload vector
Address of transfer code
Autoload Vector
Address of transfer code
Entry descriptor for tranfer code
Address of autoload vector
Pointer to local autoload code
Address of image descriptor
Transfer Code
LDQ 8(R10),R4 ;Autoload vector to R4, code to R10
OR R4,R0,R10 ;Autoload vector to R10
JSR R0,(R5) ;Jump to code address in R5
Image Descriptor
Image name
Image base address
Global autoloader invocation descriptor
Address of linkage pairs

Figure 1–2: Structures After Autoload Has Occurred


Linkage Pair
Address of invocation descriptor *
Address of routine code *
Autoload Vector
Address of transfer code
Entry descriptor for transfer code
Address of invocation descriptor *
Address of routine code *
Address of image descriptor
Transfer Code
LDQ 8(R10),R4 ;Autoload vector to r4, code to r5
OR R4,R0,R10 ;Autoload vector to r10
JSR R0,(R5) ;Jump to code address in r5
Image Descriptor
Image name
Image base address
Global autoloader invocation descriptor
Address of linkage pairs
* = updated

1.2.6.1 Autoloader

The autoloader consists of two parts, a local portion generated by the linker and a global portion which resides in the shareable image *mica$fm_share*.

The local portion does the following:

- Checks to ensure that R10 contains the address of the autoload vector. This is accomplished by comparing R10 to 8(R10). If they are unequal, then the image is already loaded and may be called by executing the transfer code instructions.
- Saves all scratch registers except R4 and R5.

- Loads R10 with invocation descriptor, and loads R11 with the code address of the global autoload routine.
- Calls the global autoload routine.
- Upon return, restores the saved registers and executes the transfer code sequence to call the newly loaded routine.

The global autoloader does the following:

- Disables ASTs.
- Performs synchronization to ensure that multiple threads are not attempting to load the same shareable image simultaneously. See Section 1.2.4 for details.
- Checks to ensure that the autoload has not already occurred. This is done by comparing R10 with 8(R10). If they are not equal, 8(R10) contains the invocation descriptor address and 12(R10) contains the code address.
- Maps in the shareable image if it is not already mapped.
- Performs fixups on the shareable image.
- Invokes the shareable image's initialization procedures.
- Performs synchronization to ensure that multiple threads are not attempting to update the shareable image's linkage pair area simultaneously.
- Sets the protection of the linkage pair areas to read/write and updating all elements represented in the image descriptor's array of longword pairs. As linkage pairs are updated, the autoload vector which the invocation descriptor refers to is also updated.
- Restores AST state.
- Returns to the local autoload procedure.

After the autoloader has executed, the linkage pair contains the actual address of the invocation descriptor and the code, and the autoload vector contains the actual address of the invocation descriptor in the third longword and the actual address of the code in the fourth longword.

\This section will track any changes in the Prism calling standard.\

1.2.7 Autoloading System Services

At system initialization some pages of system space are allocated as kernel entry pages (user read, fault on execute, kernel entry point fields are set). The starting address of these pages is stored in the global variable *e$system_services_base*.

Also, at system initialization, the *exec$create_section* service is called to create a section for the system services shareable image. This image is mapped into shareable image space, thus creating a segment for the subsequent references. In order to fixup the vectors, the address found in *e$system_services_base* is added to each offset. Thus, when a JSR is issued to a system service, the destination address of the JSR is within the system service vector page.

1.2.8 Image Startup

Once the fixup operation has completed and all "activate immediately" shareable images have been loaded and initialized, the thread startup procedure, still running in user mode, locates the transfer address to be called.

The transfer address is called with the standard argument list. This transfer address is the entry point of the user image.

\ Exact details of the argument list are TBD. \

1.2.9 Merged Image Activation

The image activator supports merged image activation to allow the emulation of the *lib$find_image_symbol* routine. Note that VMS compatible routines for *$imgact* or *imgfix* is not provided since they are not documented in the VMS system services manual.

1.2.10 Installation of Images

The Install Utility serves two purposes. It allows:

- Installation of a shareable image within the shareable image space
- INSTALL /WRITE functionality

All images which are installed have a segment object. Since the segment object contains a channel to the specified image file, the image is effectively installed "opened".

1.2.10.1 Images Within Shareable Image Space

When a shareable image is installed in the shareable image space by use of the /BASE qualifier, the image file is opened and prototype PTEs are built. The segment object for the shareable image contains the base address for the image within the shareable address space. When the shareable image is mapped, it is mapped at the BASE address specified. If the image cannot be mapped at the specified address space, due to addressing conflicts, an error is returned and the shareable image is not mapped.

When the shareable image is installed no fixups are performed. Internal fixups are not performed because of the complex nature in forcing the fixups back to the prototype PTEs. However, by linking the shareable image as based and installing the shareable image at its linked based address, no internal fixups are required.

External fixups are not performed to allow later versions of referenced shareable images to be installed (at different base addresses) while the system is running, and the latest image is autoloaded.

It is not possible for the same address space to have two different versions of the same shareable image loaded. This problem is avoided because the synchronization rules followed when shareable images are loaded.

1.2.11 Image Mapping into System Space

During system initialization and normal system operations shareable images need to be loaded into system space. Such shareable images could be function processors, object service routines or user loadable system services, or in the case of system initialization, components of the executive.

The image activator is subsetted to create a system image loader which provides this functionality. The system image loader loads and binds shareable images with the executive.

\There also needs to be a executive service to allow system management to invoke the system loader.\