Section 4
EXECUTION ENVIRONMENT

In this section: See page

Introduction.. 4-3
Task Groups and Tasks... 4-3
Multiprogramming.. 4-4
Task Step Control... 4-4
Multitasking.. 4-5
Lead Task... 4-6
Application Design Benefits of Task Group Use................. 4-6
Intertask Communication....................................... 4-6
System Control of Task Groups................................. 4-7
Generating Task Groups and Tasks.............................. 4-8
Characteristics of Task Groups and Tasks...................... 4-10
Task Group Identification..................................... 4-11
Memory Management and Protection.............................. 4-12
Segmentation.. 4-12
Segmentation With Basic Memory Management Unit................ 4-12
Segmentation With Extended Memory Management Unit............. 4-13
Segment Ring Protection....................................... 4-13
Memory Pools.. 4-14
Sharing Memory Pools.. 4-14
Memory Pool Attributes.. 4-15
Protection.. 4-15
Containment... 4-15
Privilege... 4-15
Serial Usage.. 4-16
Ring Access Rights.. 4-16
System Pool... 4-16
Swap Pools.. 4-17
Independent Pools... 4-19
Selecting Memory Pool Types................................... 4-20
Memory Pool Layout.. 4-21
Fixed System Area... 4-21

In this section (cont): See page

Bound Unit Characteristics.. 4-22
General Bound Unit Characteristics................................ 4-22
Shareable Bound Units... 4-23
Shareable Bound Units In Swap Pools............................... 4-23
Shareable Bound Units In Independent Pools........................ 4-23
Globally Shareable Bound Units.................................... 4-24
Shareable Bound Units And Executive Extensions.................... 4-24
Bound Unit Search Rules... 4-25
Bound Unit Overlays... 4-26
Nonfloatable And Floatable Overlays............................... 4-26
Overlay Areas... 4-30
Bound Unit Allocation... 4-33
Memory Allocation... 4-33
Memory Deallocation... 4-34
Swap Pool Task Address Space...................................... 4-35
Bound Unit.. 4-35
User Stack Area... 4-35
Dynamically Created Segments...................................... 4-35
Group Work Space.. 4-36
Group System Space.. 4-36
System Global Space... 4-36
System Representation of Task Address Space....................... 4-36
Task Address Space In System With Basic Memory Management Unit.. 4-37
Task Address Space In System With Extended Memory Management Unit.. 4-39

Summary

This section describes the MOD 400 execution environment. It discusses task groups and tasks, memory management, segmented demand paging, memory pools, and bound units.

INTRODUCTION

System control of user applications and system functions is accomplished within the framework of the task group. A task group consists of a set of related tasks. The most simple case of a task is the execution of code produced by one compilation or assembly of a source program (after the code is linked and loaded).

A task group is both the owner of system resources and the context in which system control of tasking is accomplished. A task can be characterized as the execution of a sequence of instructions that has a starting point and an ending point, and performs some identifiable function. It is the unit of execution of the Executive, and its execution must be requested through the Executive software.

The source language from which task code is derived can be any of the languages supported by the Executive. Source code is compiled (or assembled) and linked to form bound units consisting of a root and zero or more overlays. (Refer to "Bound Unit Characteristics" later in this section for more information.)

TASK GROUPS AND TASKS

You can configure a system dedicated to interactive applications or to a combination of interactive and absentee applications. This flexibility of configuration is based on the concept of the task group as the owner of the system resources it requires for execution.

By defining more than one application task group to run concurrently, you are utilizing multiprogramming. You can step through an application in sequence by causing tasks in the group to be executed one at a time, or you can multitask an application by causing tasks within the group to be executed concurrently.

Multiprogramming

Since multiple applications can be loaded in memory at the same time, contending for system resources, the system builder must define an environment for each application so that the application knows the limits of its resources. This defined environment is called a task group, and its domain includes one or more tasks, a memory pool, files, peripheral devices, and priority levels.

By defining the total system environment to consist of more than one task group, the system builder divides up the resources so that more than one application can run concurrently. To do this the system builder divides the memory not occupied by the Executive into one or more user memory pools. Users assign task groups to memory pools at group creation time. Task code of a task group is loaded into this task group's pool; the task obtains dynamic memory from that pool.

Task Step Control

By using the resources of one task group repetitively, you can run an application as a sequence of job or program steps. To do this, you invoke the Spawn Group command to create a task group that uses the Command Processor (whose function is to process system-level commands). You can enter commands through task groups whose lead task is the Menu Processor or the Command Processor.

One method of sequencing application steps is to have this spawned group issue a Spawn Task command for each task to be executed. This command causes a task to be loaded, executed, and then deleted. Provided the Command Processor is instructed to wait for completion of each spawned task, the tasks in the group can be executed in sequence. For example:

```
ST 1 -EFN REP_DATA -WAIT
```

(Spawn task to gather report data and wait for it to complete)

```
ST 1 -EFN PR_RPT -WAIT
```

(Spawn task to print report and wait for it to complete)

Multitasking

A variation of step control can be used to attain multitasking within one task group. Consider the situation in which the Command Processor is the lead task and reads a file containing Spawn Task commands. The Command Processor does not wait for the execution of the individual tasks; rather it continues to spawn tasks until it reads an end-of-file or &Q directive. The spawned tasks are loaded and run concurrently in this task group, contending among themselves for the resources belonging to the group. For example:

ST 1 -EFN REP_DATA

(Spawn task to gather report data)

ST 1 -EFN PR_RPT

(Spawn task to print report)

Synchronization

This method can be used only if a synchronization mechanism such as a semaphore is employed to ensure that PR_RPT does not run until REP_DATA has finished (refer to "Semaphores" in Section 5 for further information).

In a multiprocessor system it is possible that the PR_RPT program will be started before REP_DATA has finished gathering its data. In any system, each task is given a certain amount of time to execute, after which it must wait for some event. If the task exceeds this amount of time, the system schedules it to resume after other tasks. It is possible that REP_DATA could be stopped before it has finished collecting the data and that PR_RPT could be started. For reasons such as these, a synchronization mechanism is a necessity.

Lead Task

The Command Processor must be the lead task of an absentee task group so that it can read the EC file containing the desired commands. However, the Command Processor does not have to be the lead task of an interactive task group. An application consisting of one task could execute in a task group whose lead task is the application task.

If the application requires step control or multitasking and you do not need to use commands for control, you can generate a task group whose lead task contains the Assembly language system service macrocalls whose functions are analogous to the Create Group, Create Task, Spawn Group, and Spawn Task commands.

These situations are illustrative and do not exhaust the various ways in which you can control program execution.

Application Design Benefits of Task Group Use

Designing an application around a task group provides intertask communication and Executive control of multiple unrelated task groups.

Intertask Communication

The tasks in a task group execute asynchronously under control of the Executive. Tasks within a group can use control structures supplied with each task request for intertask communication.

Asynchronous tasks provide effective software response to information received from real-time external sources, such as communications or process control systems. Usually, the task (a line protocol handler) that is activated to handle the interrupt from the external source has a higher priority and a shorter execution time than the task that processes the information. The task that responds to the interrupt will use the Executive to request the execution of the processing task, supplying along with the request a control structure containing a pointer to the new information to be processed. The Executive responds to the request by activating the requested task or by queuing the request if other requests for the execution of the task are still pending.

Communications applications can use a high priority task to respond to data interrupts and determine which processing task should handle the data. This higher priority task uses the system to queue requests for the processing task, thereby accommodating peak-load conditions in which data is received faster than it can be processed.

In a process control system, the real-time clock might provide the interrupt that causes the higher priority task to scan and update temperature, thickness, or raw material level sensors that monitor the physical status of the process. This information is passed to a processing task with a lower priority that determines the necessary adjustments based on the new data. A third task, having a priority between the other two, could be requested to make whatever changes are required (for example, to change the flow rate of material entering the process by closing a valve).

System Control of Task Groups

Job/Step Sequencing

System control of an application based on the use of multiple task groups is important for several reasons. First, these applications can be thought of as consisting of multiple unrelated "jobs" (task groups) made up of one or more "job steps" (tasks). The sequence of task execution can be controlled by the system (Command Processor) as it processes synchronously supplied commands instead of responding only to externally supplied interrupts. The next "step" is started only when the previous step terminates. (You must ensure that the steps will be carried out in order.)

Resource Use

If any one set of tasks does not fully use the available processing time, the system can make more efficient use of resources by rotating their use on the basis of interrupts and priority level assignments.

Job Independence

The use of independent task groups that are subject to system control prevents one task group from adversely affecting another. If an error occurs in one task group, this group can be aborted while the others continue to execute.

Generating Task Groups and Tasks

The system provides tasking facilities regardless of the source code in which the application is written. Once generated, all tasks are subject to the same system controls, whether written in COBOL, FORTRAN, BASIC, Pascal, C, Ada, or Assembly language.

Some languages (such as COBOL and BASIC) do not provide for tasking as part of the programming language's capabilities. In these cases, the generation of tasks consisting of code written in those languages is done through commands.

Although tasks written in languages such as Assembly language and FORTRAN can be generated at the control language level, these languages have a facility for generating task groups and tasks without recourse to commands. Assembly language programs use system service macrocalls; FORTRAN programs use tasking routines.

From the overall system viewpoint, the actions of the control language in the generation of task groups and tasks are much more visible than the same capabilities in Assembly language and will be considered next.

Generation Commands

As shown in Table 4-1, commands submitted by the operator and commands submitted by other users share some of the task group generation functions and also perform unique functions. The control commands are divided into three groups:

1. Commands that perform the same function whether submitted by the operator or another user.
2. Commands that can be entered only by the operator.
3. Commands contained within the content of an existing task group request.

Table 4-1. Task Group and Task Functions Possible From Interactive and Absentee Modes

Function	User Commands		Operator Commands	
	Interactive	Absentee	Interactive	Absentee
Create Group	Yes	No	Yes	Yes
Enter Group Request	Yes	Yes	Yes	Yes
Delete Group	Yes	No	Yes	Yes
Abort Group	Yes	No	Yes	Yes
Spawn Group	Yes	No	Yes	Yes
Bye	Yes	Yes	No	No
Suspend Group	Only operator commands exist for these functions.		Yes	Yes
Activate Group			Yes	Yes
Abort Group Request			Yes	Yes
Create Group Request Queue			Yes	Yes
Create Task	Yes	Yes	Only user commands exist for these functions.	
Delete Task	Yes	Yes		
Enter Task Request	Yes	Yes		
Spawn Task	Yes	Yes		

NOTE: The Command Processor executes in interactive and/or absentee mode.

Characteristics of Task Groups and Tasks

Task groups and individual tasks can be originated in either of two ways: by creation or by spawning. The choice depends on application design considerations as well as the intended functions.

There are important differences between tasks (and task groups) that are generated by a create function and those originated by a spawn function. Created task groups and tasks are permanent; they remain available in memory until explicitly removed. Spawned task groups and tasks are transitory; they perform a function and disappear.

Created task groups and tasks are passive; they must be explicitly requested to execute in order to perform their intended function. Spawned task groups and tasks cannot be requested. The spawning of a task group or task is equivalent to a create-request-delete sequence of control language commands. In a spawn operation, the task group or task is defined, provided with system resources and control structures, executes, terminates, and has its resources deallocated, all in one continuous process.

Task code may cause extensive action in its own behalf, as when application task code requests a system service or the execution of another task while awaiting the completion of the requested task. Each task that requests another supplies the address of a control structure through which the issuing task and the requested task can communicate, and which the Executive uses to coordinate task processing.

Task Group Identification

Each task group has a unique identifier. Honeywell Bull supplied system task group identifiers begin with a $, as shown below:

Task Group ID	Function
$H	Honeywell Bull-supplied user task group
$L	Listener
$P	Deferred print
$S	System task group

Group-Id

The identifier for a user task group in the Create Group or Spawn Group command is a 2-character name that should not have the dollar sign ($) as its first character. The identifier (or group-id) can be indicated or implied in commands to designate what task group is to be acted upon. The operator can include the task group identifier when responding to operator terminal messages from the task group.

MEMORY MANAGEMENT AND PROTECTION

The system (hardware and software) provides a memory management and protection facility that performs the following functions:

- Allocates memory to guarantee each task group (user) its own address space.
- Protects multiple users from each other and the system from the users.

The hardware used to provide memory management and protection is called a memory management unit. The type of memory management unit varies according to the kind of processor. DPS 6 systems use either the Basic Memory Management Unit (BMMU) or the Extended Memory Management Unit (EMMU). Each of these memory management units is based on the concept of segmentation.

Segmentation

The memory management unit maps a segmented address space onto physical memory. The unit of memory allocation is a segment. A segment is a variably sized area of memory that usually consists of a logical entity such as a procedure. The system memory management and protection facility treats all addresses generated by the central processor as segment-relative addresses. It maps the segment-relative addresses through the memory management unit to absolute physical addresses.

Segment Size — No segment can be less than 512 bytes in length. Segment size is always a multiple of 512 bytes.

Segmentation With Basic Memory Management Unit

The BMMU supports up to 31 segments, 16 of which can be up to 8K bytes (K=1024) in size and 15 of which can be up to 128K bytes in size. The segments that can be up to 8K bytes are called "small segments"; those that can be up 128K bytes are called "large segments." The 16 small segments are numbered from 0.0 through 0.F; the 15 large segments are numbered from 1 through F. All small segments, and often some large segments, are reserved for system use; the actual number reserved is established at system generation.

The BMMU provides a total of 2 million bytes of segmented address space for each task.

Each segment is described by a 4-byte segment descriptor that contains the segment's starting physical address, its length (in units of 512 bytes), and its access rights for each ring (refer to "Segment Ring Protection" below).

Although you can assign any of the large segments to a bound unit when it is linked, the availability of a segment depends on the system configuration. Therefore, most applications simply let the system assign segment numbers. The identity of the segments available to you should be obtained from the system administrator.

Segmentation With Extended Memory Management Unit

The EMMU supports up to 256 segments, each of which can be up to 128K bytes in size. The segments are numbered from 00 through FF. Segments 00 through 7F are reserved for system use; segments 80 through FF are available for user tasks.

The EMMU provides a total of 32 million bytes of segmented address space for each task.

Each segment is described by a 2-byte segment descriptor that contains the segment's starting physical address, its length (in units of 512 bytes), and its access rights for each ring.

Segment Ring Protection

Access to memory segments is controlled through the memory management unit. The memory management unit assigns each executing task to a ring of privilege. (Rings may be thought of as concentric circles, like a target. The innermost circle, ring 0, has the most privilege.) During the linking of a bound unit, you can assign access attributes to each bound unit to indicate whether a task executing in a particular ring of privilege can read, write, and/or execute in the code or data segment of the bound unit. However, it is recommended that you use the system defaults.

Task Execution

System tasks execute in ring 0 (privileged state). User tasks can execute in rings 2 and 3. Ring 0 is most privileged, and ring 3 is least privileged. The ring in which a user task executes is defined by the type of memory pool to which the task has been assigned (refer to "Ring Access Rights" later in this section).

Access Checking

Every attempted access to a segment is checked for legitimacy. The system compares the ring number of the executing task with the access attributes of the segment to be accessed. An access violation trap occurs if a user application attempts to access one of its segments without having the proper access rights.

MEMORY POOLS

At system startup the Configuration Load Manager (CLM) reads a file of directives, sets up memory pools from the supplied specifications, and indicates to the Loader what system and user-written software is to be resident for the life of the system. On the DPS 6/22, the Autoconfigurator creates the file. On other systems, the system builder can use the line editor to create the file.

After the system has been in operation for a while, experience may show the desirability of reconfiguring the pool sizes so that they meet user requirements more precisely. The system builder can hand-tailor the MEMPOOL and SWAPPOOL directives in the CLM file using the line editor. Refer to the *System Building and Administration* manual for information on the CLM directives.

The system supports the following types of memory pools:

- System pool - Contains the system task group and all globally shared elements. There is only one system pool per system.
- Swap pool - Provides an environment in which segments can be swapped out to disk to make room for competing users, and in which the memory requirements of individual users do not have to be predetermined. Systems with EMMUs should have all of user memory as one swap pool. Systems with BMMUs should have all of user memory as multiple swap pools.
- Independent pool - Provides an environment suitable for applications with well-defined memory requirements, all of whose tasks must must be in memory at the same time. There can be multiple independent pools in the system.

Swap and independent pools allow applications running on systems with a BMMU to access more than 2 million bytes of physical memory.

If multiple swap or independent pools are configured, Listener can optionally ensure an even distribution of task groups among memory pools by assigning each new user to the memory pool with the fewest task groups.

Sharing Memory Pools

Swap and independent pools will be shared if users assign more than one task group to the same pool. As the tasks execute, they contend for the same memory space. Tasks running in an independent pool should be designed so that they can be suspended or take some alternative action when no additional memory is available.

Memory Pool Attributes

Memory pools can have one or more of the following attributes: protection, containment, privilege, serial-usage, and ring access rights.

Protection

When a memory pool has the protection attribute, it cannot be written into by a task running in another pool. The Executive uses the memory management unit to prevent all write intrusions by foreign tasks. A task attempting to write into a protected pool receives an error notification from the Executive.

Protection applies to memory pools and not to task groups. Groups sharing a a memory pool are protected from each other only in the swap pool. Tasks within a group are not protected from each other.

All pools are automatically generated as protected; this attribute cannot be changed.

Containment

When a memory pool has the containment attribute, tasks running in the pool cannot write outside the pool area. The Executive uses the memory management unit to prevent all tasks from writing outside the pool. A task attempting to write outside of a contained pool receives an error notification from the Executive.

The system pool cannot be contained. Swap and independent pools are automatically generated as contained; this attribute cannot be changed.

Privilege

When a memory pool has the privilege attribute, any task running in that pool can execute privileged instructions. The following Assembly language instructions are privileged:

```
ASD     IO      LEV     WDTF
CNFG    IOH     RTCF    WDTN
HLT     IOLD    RTCN
```

If the pool does not have the privilege attribute, any task attempting to execute one of the above instructions will trap.

The system pool is always privileged; this attribute cannot be changed. Swap and independent pools are unprivileged; this attribute can be changed in the CLM MEMPOOL directive.

Serial Usage

When a memory pool has the serial-usage attribute, it can be used by only one task group at a time.

The system and swap pools are generated without the serial-usage attribute; this specification cannot be changed. Independent pools can be specified in the CLM MEMPOOL directive as having the serial-usage attribute.

Ring Access Rights

Each type of memory pool is automatically assigned a ring access designation. There are four rings, numbered 0 through 3. Rings 0 and 1 are privileged rings; rings 2 and 3 are unprivileged. Tasks acquire the ring attribute of the pool to which their task group is assigned. The pools and their associated rings are:

Pool	Ring
System	0
Swap	3
Independent	1 or 2

Independent pools have ring 1 access if they are privileged and ring 2 access if they are not privileged.

Accessing Memory

Ring access is used with segment ring protection to determine the ability of a task to access memory (refer to "Segment Ring Protection" earlier in this section). A task whose ring access is 3 can only access memory protected at ring 3. A task whose ring access is 2 can only access memory protected at rings 2 and 3. A task whose ring access is 1 can access memory protected at rings 1, 2, and 3. A task whose ring access is 0 can access memory protected at rings 0, 1, 2, and 3.

The following paragraphs describe each type of memory pool in greater detail.

System Pool

The system pool contains the system task group ($S), certain file control structures, and system elements that are to be shared. Its maximum size is 4 million bytes. The system pool is protected, privileged, and not contained. Tasks running in this pool have ring 0 access.

User Groups

User task groups cannot be created in the system pool. User tasks cannot execute in the system pool with ring 0 access.

System Group

The system task group cannot be aborted or suspended. Since it never terminates, it cannot be requested. The system group always has read and write access to all of memory. It handles all system dialog (including operator commands) through the CLM-designated operator terminal.

System Elements

The file control structures in the system pool are the File Description Blocks (FDBs) and the buffers for shareable files. Other system elements in this pool include:

- Current function invoked by an operator command.
- Extended Trap Save Areas (TSAs) needed during processing.
- Control blocks for all tasks (TCBs) and task groups (GCBs).
- Globally shareable bound units.
- File System directory and file definition blocks.
- Public buffer pools.
- Memory control blocks for swap pool segments.

Swap Pools

A swap pool is a privileged, protected, and contained pool in which segments can be swapped out to disk in order to make physical memory available to competing users. Swap pool memory management can move segments to physical memory in order to eliminate fragmentation and consolidate available memory space.

Swap pools support both interactive and absentee processing.

Size

The size of a swap pool, plus the size of the Executive and its structures, cannot exceed 2 million bytes in a system having a BMMU and 16 million bytes in a system having an EMMU. On systems with a BMMU, the system builder can configure multiple swap pools. On systems with an EMMU, all of user memory should be one swap pool.

Virtual View

A swap pool is a separate virtual view of the system. That is, each task in a swap pool can view that portion of the pool relating to itself and can view all of the global system space. Tasks running in the swap pool have ring 3 access.

Swapped Out The system acquires space for a given segment from whatever portion of free swap pool memory is available. If not enough space is available for the needed segment, the system attempts to obtain memory by swapping out lower priority tasks in the same pool that are waiting on an event. If this action does not produce enough memory, the requesting task is swapped out until sufficient space becomes available. A task is swapped out under one of the following conditions:

- If it is waiting on an event that is of potentially long duration and swap pool memory is required by a competing task.
- If memory is required to roll in a higher priority task.
- If the task has been suspended by the operator.

Swapped In A task is swapped back in when the swap pool memory is available. The task may be swapped in immediately, it may be swapped in after tasks waiting on events of long duration are swapped out, or it may be swapped in after lower priority tasks are swapped out. A task is swapped back in when any event on which it was waiting has completed or when it is reactivated by an operator.

Task In Memory The entire context of a task in the task group must be in memory for the task to execute. Other tasks in the task group need not be in memory. However, thrashing may occur if too many users are assigned to too small a swap pool.

Protection A task in a swap pool can overwrite shared data designated for writing. A task cannot overwrite another task, another task's data, or shareable read-only data. Further, tasks in a swap pool can only read (not write) system structures.

Segment Assignments Tasks running in a swap pool have logical elements (for example, bound units) equated with segments. The Executive aligns the logical elements on segment boundaries. This configuration is represented in Figure 4-1.

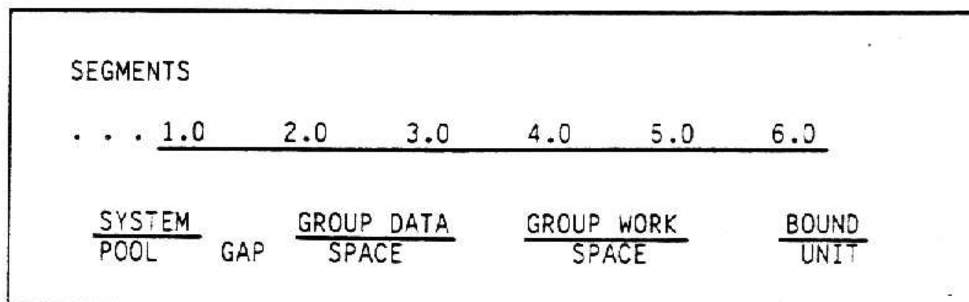

Figure 4-1. Sample Swap Pool Group Segment Assignments

Each task group in the swap pool has group global space that cannot be accessed by any other group. Each task in a swap pool group can have task private space that cannot be accessed by any other task. For further information refer to "Swap Pool Task Address Space" later in this section.

Independent Pools

An independent pool is protected and contained; it can be privileged or not and serial-usage or not. Tasks running in an independent pool have ring 2 access if the pool is unprivileged and ring 1 access if it is privileged.

Independent pools support both interactive and absentee processing.

Size

The size of an independent pool, plus the size of the Executive and its structures, cannot exceed 2 million bytes in a system having a BMMU and 16 million bytes in a system having an EMMU. On larger systems, multiple independent pools can be configured.

It is important that the memory requirements of the group(s) using an independent pool be estimated carefully because the entire context of a task group must be in memory for a task in the group to execute. It is possible that task group memory requirements may exceed the size of the memory pool because of memory fragmentation.

Protection

Note that even with protection and containment, a task in an independent memory pool can accidentally overwrite code or data belonging to its own or another group in the pool.

Virtual View

Each independent pool is a separate virtual view of the system. That is, each task in an independent pool can view that portion of the pool relating to itself and can view all of the global system space. Thus, a task in an independent pool can reference a memory location in that pool and in system global space.

Use

Independent pools are designed for applications that you may not want to execute in a swap pool.

Segment Assignments

The system aligns user memory pools on segment boundaries (multiples of 512 bytes). This configuration is shown in Figure 4-2.

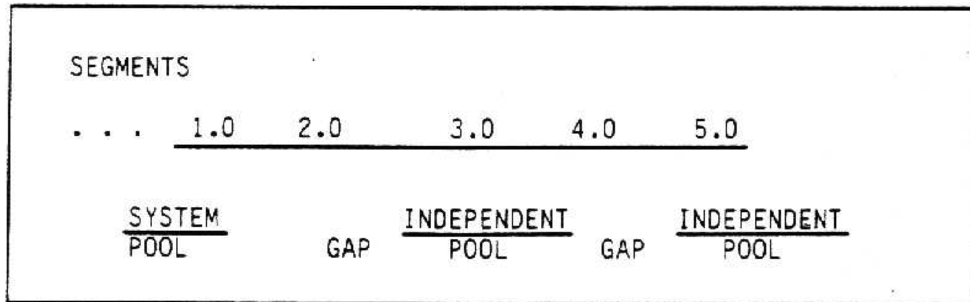

Figure 4-2. Sample Independent Pool Group Segment Assignments

Selecting Memory Pool Types

The different types of memory pools provide you with the means to respond to the unique demands of multiple application programs. Through the use of memory pools, you can exercise control over memory usage and, at the same time, provide individual task groups with specialized protection.

The degree to which the system can efficiently and effectively handle the concurrent execution of multiple task groups depends on the number and type of memory pools available for use. The following points should be kept in mind:

- All systems must have a system pool.
- In systems with a BMMU, all user memory should be devoted to multiple swap pools.
- In systems with an EMMU, all user memory should be one large swap pool.
- One or more independent pools can be selected for applications that you do not want to run in a swap pool.

Default Memory Pool

If you do not configure any memory pools, you will be provided with one swap pool whose size is all of memory, less the amount of memory occupied by the system pool.

Memory Pool Layout

To obtain efficient use of memory and of the memory management unit, the CLM sorts the memory pools specified in a configuration as follows:

1. The system pool is configured in the first available memory after the system data structures. The system pool cannot exceed 4 million bytes.
2. If swap pools are configured, they follow the system pool.
3. Independent pools are configured after all swap pools.

Fixed System Area

After the configuration process is complete, the following software components and data structures are located in the fixed system area of memory:

- Basic Executive plus resident overlays
- User-written or vendor-supplied extensions to the Executive
- Device drivers
- Intermediate request blocks needed for task groups
- Trap save areas
- Overlay area(s) for system software
- File control structures.

The fixed system area is static. Unlike the other memory areas whose contents can vary dynamically, its structure remains the same for the life of the system. Almost all code loaded into this area is reentrant so that a single copy of the code is available to multiple users, thus minimizing memory requirements.

BOUND UNIT CHARACTERISTICS

Task code is derived from programs written in a source language and compiled or assembled to form object units (also called compilation units). One or more object units are linked to form a bound unit that is placed on a file. The bound unit is an executable program that can be loaded into memory. A task represents the execution of a bound unit.

General Bound Unit Characteristics

Bound units have the following general characteristics:

- Each bound unit consists of a root segment and any related overlays.
- A load element is composed of one or more object units.
- The initial load element is called the root; it must be resident when the bound unit is being executed.
- A load element that replaces another load element when loaded into memory is called an overlay.

Linker Actions

You can direct the Linker to perform the following actions on bound units:

- Map the code and data into the same load element or into separate elements. If the bound unit is to be reentrant, the code and data must be in separate load elements.
- Specify ring access rights. If the bound unit is to be reentrant, the default access attributes are ring 3 read and execute access for both code and data, ring 0 write access for the code segment, and ring 3 write access for the data segment.
- Associate a specific segment number or numbers with a bound unit. Normally, the Linker assigns default segment numbers. If you assign segment numbers at link time, you must be careful to avoid segment conflicts in the configuration and application environment in which the bound unit is to run so as to avoid inefficient loading.

Address Space

Your physical address space is not necessarily contiguous. Memory requirements are satisfied on a segment basis rather than on a user basis.

Note that for systems with a BMMU, you have a maximum of 11 large segments available when constructing a task's address space. Frequently, fewer large segments will be available, depending on the system configuration.

Shareable Bound Units

The use of shareable bound units is a way of minimizing application task group memory requirements while making reentrant code available to multiple tasks. Unlike permanently resident bound units that are loaded during system configuration, shareable bound units are transient in memory and are loaded during processing. A usage counter is incremented each time a request is made for the bound unit, and decremented each time a request is completed. The unit remains in memory as long as a task is using the code. As soon as the usage counter is decremented to zero, the space occupied by the bound unit is returned to available status.

Shareable Bound Units In Swap Pools

If a bound unit is shareable only within a swap pool, its root segment descriptor is placed in a portion of memory where it is accessible to all tasks in that pool. The bound unit should have no fixed overlays; floatable overlays can be shared if an OAT is used. (See "Bound Unit Overlays" later in this section.) To be recognized as shareable by the Loader, and to be loaded into a user memory area, the bound unit must have been linked using the SHARE directive.

Additionally, task private segments are shared if the task forks. (A task forks if it issues a Create Task or Spawn Task command with an entry point address rather than a pathname definition.) Forked tasks share the same segments; they have the same access to and copy of the forked segments until one task modifies its address space. (Address space defines a task's boundaries in a swap pool. Refer to "Swap Pool Task Address Space" later in this section.)

Shareable Bound Units in Independent Pools

In an independent pool, bound units can be shareable by tasks within a task group or can be shareable by all task groups. Shareability is established when the bound unit is linked. To be shareable by tasks and task groups within a pool, the bound unit must be linked with the SHARE directive. Bound units linked with the SHARE directive are loaded into the requesting task group's memory pool.

Globally Shareable Bound Units

To be shareable by task groups in other pools (globally shareable), the bound unit must be linked with the GSHARE directive. Bound units linked with the GSHARE directive are loaded into the system pool. Since system pool memory is a critical resource, the use of globally shareable bound units requires careful planning and control. If all of user memory is one large swap pool, the SHARE directive has the same effect as the GSHARE directive, and does not clutter up the system pool.

Operator commands can be used to load and unload globally shareable bound units.

Shareable Bound Units And Executive Extensions

Shareable bound units and the Executive extensions that are loaded through LDBU directives when the system is configured differ in one major respect. Executive extensions can be accessed symbolically by any task, but a shareable bound unit must be accessed as a bound unit.

When an Executive extension is loaded during system configuration and is made permanently resident by an LDBU directive, its symbols are included in the system symbol table. Since a shareable bound unit is transient and is loaded after the system has been configured, no entry is made for it in the system symbol table. For this reason, it must be accessed as a bound unit. Table 4-2 compares permanently resident Executive extensions and transient shareable bound units.

Table 4-2. Comparison of Executive Extensions and Shareable Bound Units

Characteristics	Executive Extension	Shareable Bound Units
Multiple users	Yes	Yes
Permanently resident (fixed system area)	Yes	No
Temporarily resident (system or user pool)	No	Yes
Symbols in system table	Yes	No
Accessed symbolically	Yes	No
Have overlays	No	Yes (See Note 2)
Called by bound unit name	No	Yes

NOTES:

1. If the extension is an Assembly language bound unit, it may have within it sections of code or control structures controlled by semaphores that would be accessible to other Assembly language tasks (refer to "Semaphores" in Section 5 for further information).
2. Overlays are not shareable unless Overlay Area Tables (OATs) are used (refer to "Bound Unit Overlays" below).
3. The Executive does not "remember" extensions by their names. A request for an extension by name results in another copy being brought into memory.

Bound Unit Search Rules

The Loader uses search rules to locate a bound unit to be loaded. The Loader starts the search in response to a command containing an argument naming the bound unit to be loaded.

The rules that regulate the search process define three directory pathnames and the sequence in which they are used during a search. The pathname sequence is as follows:

1. User task group working directory.
2. System directory -LIB1 argument of the Change System Directory command.
3. System directory -LIB2 argument of the Change System Directory command.

The Change System Directory command can be used to change pathnames associated with system directory arguments -LIB1 and -LIB2. The pathname of a user's working directory is established through a Change Working Directory command or through the -WD argument of the Enter Group Request or Spawn Group command. For login users, the -WD argument of the Spawn Group command issued by the Listener is taken from the -HD argument in the Login command line.

Bound Unit Overlays

In smaller systems, you may need to minimize the amount of memory required to execute a bound unit containing application code. You can accomplish this by directing the Linker to create the bound unit as a series of overlays (separately loadable pieces) so that the entire bound unit does not have to be resident at one time. Each bound unit consists of a root and, optionally, one or more related overlays. The system loads the bound unit root automatically when the bound unit is invoked. Overlay loading is controlled by the application itself. The maximum number of overlays is 65,536. The use of overlays requires careful planning so that required code is not lost or repetitively loaded.

Two types of overlays are available for your use: nonfloatable and floatable. In addition, you can use overlay areas to control the placement of floatable overlays.

Nonfloatable and Floatable Overlays

Overlays can be loaded at a fixed displacement from the base of the root (nonfloatable overlay) or into a block of memory allocated explicitly by you or implicitly by the system (floatable overlay).

Nonfloatable A nonfloatable overlay is loaded into the same memory location relative to the root each time it is requested. Object units whose code is to be loaded as nonfloatable overlays must be defined as fixed overlays by the Linker OVLY directive. When the root of a bound unit having fixed overlays is loaded, the Loader allocates a container (segment or memory block) large enough to hold the root and all of its fixed overlays.

Assembly language programs can use system service macrocalls to load and execute nonfloatable overlays. COBOL programs can use CALL/CANCEL statements to control nonfloatable overlays. FORTRANA and Pascal provide overlay handlers as part of the run-time libraries. BASIC programs must link a user-written Assembly language overlay manager with the application program, since the BASIC language does not supply this functionality.

Floatable A floatable overlay is linked without having a fixed relative location to the base of the root. It can be loaded into any available memory location. Floatable overlays must meet the following criteria:

1. The overlay must not contain external definitions referenced by the root or another overlay.
2. The overlay must not make displacement references to the root or any other overlay.
3. The overlay must not contain external displacement references that are not resolved by the Linker.

The application program can use one or more areas of available memory for the placement of floatable overlays. The program can deal with memory management in one of the following ways:

1. Allow the system to place the overlay in an available memory block allocated from the user's independent memory pool or, if loaded in a swap pool, from group work space. (Group work space is a segment common to all tasks in a group. Refer to "Swap Pool Task Address Space" later in this section.)
2. Create a set of overlay areas using system service macrocalls, and allow the system to manage the areas and locate the requested overlays. In an independent memory pool, the overlay areas are created from a memory block in the pool. In a swap pool, the segment(s) allocated for the root are expanded to contain the created overlay area.

3. Perform its own memory management by linking a user-written Assembly language overlay manager with the root of the bound unit. In an independent memory pool, you may choose to have the overlay occupy part or all of a memory block. In a swap pool, you may choose to have the overlay occupy part or all of a segment.

Linking

Floatable and nonfloatable overlays are defined through the Linker. When using the Linker, forward references can be made to symbols defined in object units to be linked later (the rules for floatable overlays must be observed). Backward references can be made to symbols previously defined, provided the defined symbols were not purged from the Linker symbol table by a Linker BASE or PURGE directive. Since the specification of the BASE directive removes from the Linker symbol table all previously defined and unprotected symbols that are at locations equal to or greater than the location designated in the BASE directive, you must take one of the following actions:

- Define all symbols that are to be preserved in a part of the root that is not overlaid.

- Protect the symbols to be preserved by using the Linker PROTECT directive.

A floatable overlay can refer to fixed addresses in the root, in a nonfloatable overlay, or in itself, but cannot refer to addresses in another floatable overlay.

When a root or overlay of a bound unit is loaded, the Loader examines the attribute tables associated with the bound unit if an alternate entry point is specified. The Loader tries to resolve any references to symbols that remain unresolved by searching the system symbol table (that is, the resident bound unit attribute table). The Loader cannot resolve any references to symbols that do not exist in the symbol table. (Linker symbol tables do not exist at load time.)

Figure 4-3 shows the relative location in memory of memory pool AA. Figure 4-4 is the layout of overlays in memory pool AA. When the root is loaded, the largest contiguous amount of memory necessary to accommodate the root and all nonfloatable overlays is allocated. Except for space for any floatable overlays, no other memory requests need be made. In Figure 4-4, this memory area begins at the base of the root and continues to the end of object unit OBJD. The root consists of object units OBJ1 and OBJ2. When loaded, OBJ5 of overlay ABLE will replace the previously loaded OBJ2 code of the root. Similarly, the overlay locations were specified so that OBJC of overlay ZEBRA will replace part of OBJB.

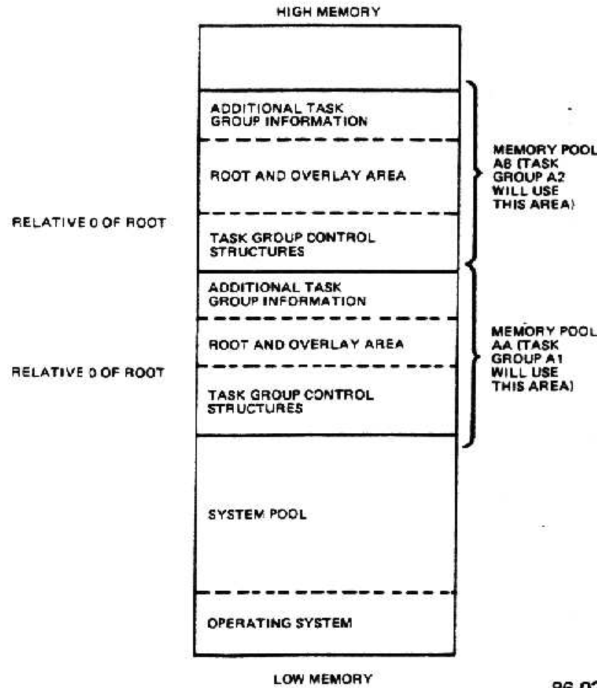

Figure 4-3. Relative Location in Memory of Memory Pool AA

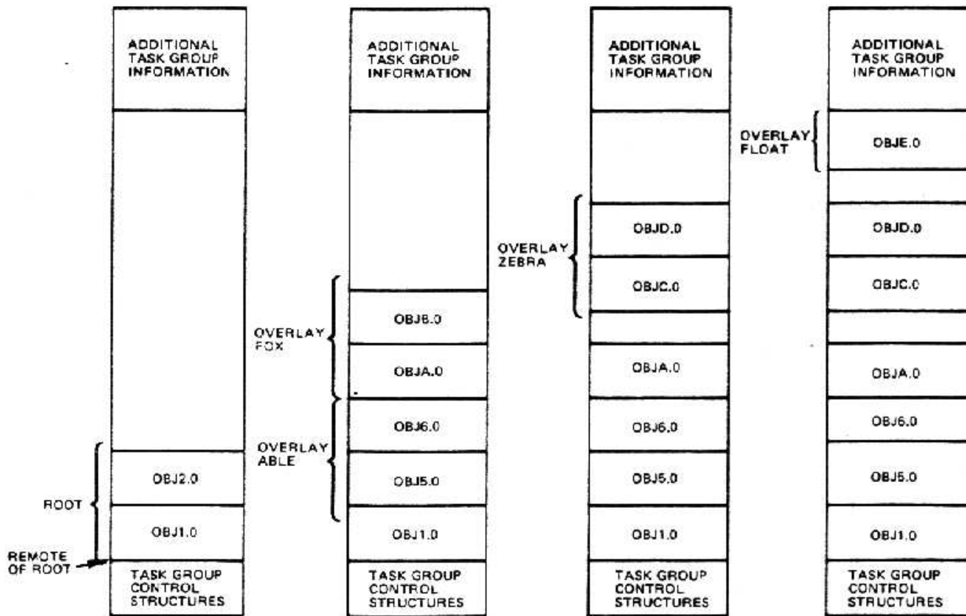

Figure 4-4. Overlays in Memory Pool AA

Overlay Areas

Only floatable overlays can be associated with overlay areas. Overlay areas are a mechanism that allows you to control the placement of floatable overlays without being required to write your own overlay manager.

Overlay areas are fixed size areas of memory whose use is controlled through an Overlay Area Table (OAT). If the bound unit is shareable, the overlays can be shared with other tasks in the task group or with tasks in other task groups. Overlays can also be shared if the bound unit is replicated through the -SHARE argument of the Create Task command.

You create an OAT through the Create Overlay Area Table system service macrocall. You reserve an overlay area and execute the overlay through an Overlay Reserve and Execute macrocall. You exit from the overlay through an Overlay Release macrocall.

As an example of overlay area use, assume that you desire to share both the root and the overlays of a shareable bound unit whose structure is shown in Figure 4-5.

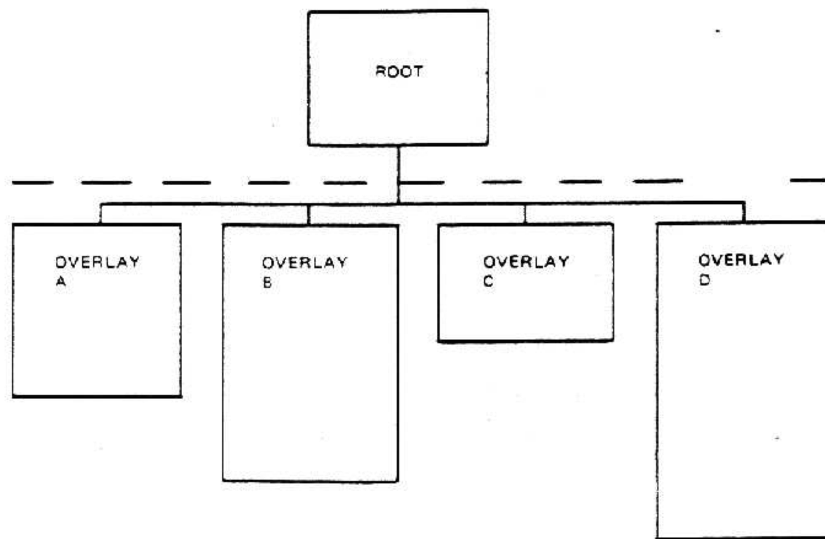

Figure 4-5. Sample Bound Unit Structure for Overlay Area Use

Assume further that tasks 1, 2, and 3 (of the same or another task group) are executing the shareable bound unit and that task 1 has encountered a create OAT function while executing the root.

When the create OAT function is encountered, an overlay area (controlled by the OAT) is created for the task group. In this example, the overlay area has three entries, each entry being 512 bytes long. There is no direct relationship between the number of overlays to be shared and the number of entries in the overlay area. The entries in an overlay area are of equal size. You must create overlay areas large enough to contain the largest overlay (overlay D in this example). The overlay area reserved is depicted below.

ENTRY 1	ENTRY 2	ENTRY 3
512 BYTES	512 BYTES	512 BYTES

When task 2 (or task 3) executes the same create OAT request (that is, when it executes the root), the task is given the address of the OAT already existing in memory.

Assume that task 1 issues an Overlay Reserve macrocall to reserve an overlay area defined by the OAT and to load overlay A in that area. The code and/or data composing overlay A will be loaded in the first free overlay area, and task 1 will be given access to this area. At this instant the status of the overlay area is as follows:

ENTRY 1	ENTRY 2	ENTRY 3
OVERLAY A USAGE = 1	USAGE = 0	USAGE = 0

TASK 1

When tasks 2 and 3 now perform the request for overlay A, they will be given access to the existing copy of the overlay. At this instant, the status of the overlay area is as follows:

ENTRY 1	ENTRY 2	ENTRY 3
OVERLAY A USAGE = 3	USAGE = 0	USAGE = 0

TASKS 1,2,3

Task 2 now requests overlay D. Since a task cannot have more than one overlay in an overlay area at any time, task 2 must explicitly release overlay A before requesting the loading of overlay D. The result of releasing overlay A and requesting overlay D is as follows:

ENTRY 1	ENTRY 2	ENTRY 3
OVERLAY A USAGE = 2	OVERLAY D USAGE = 1	USAGE = 0
TASKS 1,3	TASK 2	

A request by task 3 for overlay C will result in the following situation:

ENTRY 1	ENTRY 2	ENTRY 3
OVERLAY A USAGE = 1	OVERLAY D USAGE = 1	OVERLAY C USAGE = 1
TASK 1	TASK 2	TASK 3

If there were another task in the group (for example, task 4), and the task were to request overlay B, it would have to wait until one of the overlay areas was freed (by an Overlay Release macrocall). If task 4 requested overlay A, C, or D, the task would be given access to the loaded copy of the overlay.

Note that at any given instant several OATs, controlling several different overlay areas, may exist. Even if a task is sharing overlays in different overlay areas, it cannot reference more than one overlay area at any given time. The task must release an overlay in an OAT prior to requesting an area for another.

You use an Overlay Area Release macrocall to exit from an overlay. When this call is executed, the count of the number of users of the overlay is decremented in the defining OAT. When the count drops to zero, the overlay area is marked as available and can be reused by an Overlay Reserve and Execute function.

Bound Unit Allocation

Initial Bound Unit

Each task is associated with at least one bound unit. The initial bound unit with which a task is associated is specified at the time the task is created or spawned. At this time, the segment is created/allocated in memory, and the root is loaded in this segment.

If the bound unit was designated as shareable at link time and is currently residing in memory, no loading takes place. The requesting task shares the bound unit already in memory, and the bound unit user count is increased by one. If the bound unit is not in memory, it is loaded.

Attach Bound Unit

Execution of a task begins with the specified bound unit. During the execution of this bound unit, the Assembly language user can employ system service macrocalls to load or attach another bound unit. Loading or attaching a bound unit causes the allocation and loading of the segment containing the root of the requested bound unit. (The difference between loading and attaching is that loading returns the entry point of the root segment to the issuing task, while attaching starts the execution of the bound unit root segment at the entry point.) Up to eight bound unit units can be attached. In BMMU systems, the availability of segment descriptors may limit you to fewer than eight attached bound units.

Create Segment

During its execution, a task can issue a system service macrocall to request the creation of a segment to be associated with the task's initial bound unit or any other of its attached/loaded bound units. The macrocall can either specify a segment number or allow the system to select the number in accordance with the specified size.

Memory Allocation

The allocation of memory for a bound unit depends on whether the bound unit is nonshareable or shareable.

For a nonshareable bound unit, each logical segment is uniquely mapped to a physical segment in memory. Unless the task is forked or the segment is in an OAT, two or more tasks wishing to concurrently use a nonshareable bound unit each receive a copy of the bound unit.

If more than one task is executing a pool-shareable bound unit, only one copy of the segment containing the root is allocated in the pool. All tasks use this single copy. Overlays of the bound unit can be shared if an OAT is used. If the bound unit was separated into a code element and a data element, only the code element is shared. Except for forked tasks, each user has a separate copy of the data element.

Segment Numbers

The Memory Manager assigns a segment number (or numbers) based on the segment descriptors available to the task that initially loads the bound unit. Concurrent users must access the bound unit under the same segment numbers. If the segment utilization of the second and subsequent tasks that attempt to load the shareable bound unit conflicts with its segment number or assignment, an error is returned when the tasks attempt to load the bound unit. In this case, the tasks are not given addressability to the shareable bound unit.

Memory Deallocation

Assembly language users can explicitly deallocate a user-created segment by issuing a Delete Segment macrocall. Users can deallocate bound units by issuing a Detach Bound Unit macrocall for any but the initially assigned bound unit. A segment can be implicitly deallocated from physical memory as the result of the task being deleted or swapped out. It is reallocated when the task is swapped back in.

Overlay areas and defining OATs are deallocated when the last usage of a shareable bound unit has terminated.

SWAP POOL TASK ADDRESS SPACE

Task address space defines a task's boundaries in the swap pool; that is, its visibility within the collection of tasks executing in the pool. The following elements constitute a task's address space:

- Bound unit
- User stack area
- Dynamically created segments
- Group work space
- Group system space
- System global space.

Bound Unit

During its execution life, a task executes one or more bound units. The initial bound unit to be executed is the one specified when the task is created or spawned. In Assembly language programs, other bound units (if any) can be attached or loaded through the Bound Unit Attach or Bound Unit Load macrocalls.

User Stack Area

The user stack area is available to users as a work area through the hardware stack instructions.

Dynamically Created Segments

During execution, a task can extend its address space by creating segments. Assembly language programs use the Create Segment macrocall for this purpose. These dynamically created segments become part of the issuing task's address space.

Group Work Space

The group work space is common to all tasks in a given task group. Assembly language programs can obtain blocks of memory from the group work space when they issue Get Memory macrocalls. All tasks in the task group have read, write, and execute access to the group work space.

The group work space can occupy up to two 128K-byte segments. The group work space grows dynamically, as requests for memory are issued. In both BMMU and EMMU systems, the maximum size is 256K bytes. However, this maximum is reduced to 128K bytes if the adjacent segment descriptor has been allocated.

Group System Space

One group system space is provided for each task group. The system control structures used to support a task group and its member tasks (for example, file control blocks, bound unit descriptors for nonshareable bound units, logical file tables, and logical resource tables) are allocated from the group system space.

The group system space can occupy up to two 128K-byte segments. The group system space grows dynamically, as requests for memory are issued. In both BMMU and EMMU systems, the maximum size is 256K bytes. However, this maximum is reduced to 128K bytes if the adjacent segment descriptor has been allocated.

System Global Space

System global space consists of the fixed system area (permanently configured memory) and the system memory pool. A task's address space includes the segments required for system global space. System code and data are distributed in the task address space.

System Representation of Task Address Space

Figures 4-6 and 4-7 are examples of the mechanism used by the system to represent a task's address space. Figure 4-6 is an example of a system having a BMMU; Figure 4-7 is an example of a system having an EMMU.

Task Address Space In System With Basic Memory Management Unit

The following points should be noted when using Figure 4-6.

1. The layout of memory is logical, not physical.

2. The layout applies only to this example; it is possible to generate systems whose layout is different from that shown in Figure 4-6.

3. The segments available to you for your bound units are 6 through F. If the group system space requirements are less than or equal to 128K-bytes, segment 3 can be used. If the group work space requirement is less than or equal to 128K bytes, segment 5 may be used.

4. One copy of segments 0.0 through 1 exists in the system in this example. These segments contain the system global space. All tasks in the system can access these segments.

5. Segments 6 through F are unique to the task unless they are being shared. If one of these segments is being shared, each task sharing the segment accesses the same copy of the segment. When a segment number is assigned by the Memory Manager, the lowest available segment (or segments for objects of size greater than 128K bytes) beginning with the Group Work Space (GWS) segment plus 2 (segment 6 in this example) will be used. If all segments from GWS+2 through large segment F have been used, the segments GWS+1 (segment 5 in this example) and GSS+1 (segment 3 in this example) are allocated in that order, if available.

6. Only one copy of the group work space segment (segment 4 in this example) exists per task group. All tasks in the task group have unlimited access to this segment. Only one copy of the segment that contains group system space (segment 2 in this example) exists per task group. All tasks in the task group have read and execute access to this segment. Both the group work space and the group system space segments are dynamically expanded as demands are made on them. Each space can grow to a maximum of 256K bytes if the adjacent ascending segment descriptor (segment 3 for the group system space and segment 5 for the group work space in this example) has not previously been allocated to contain a task private segment.

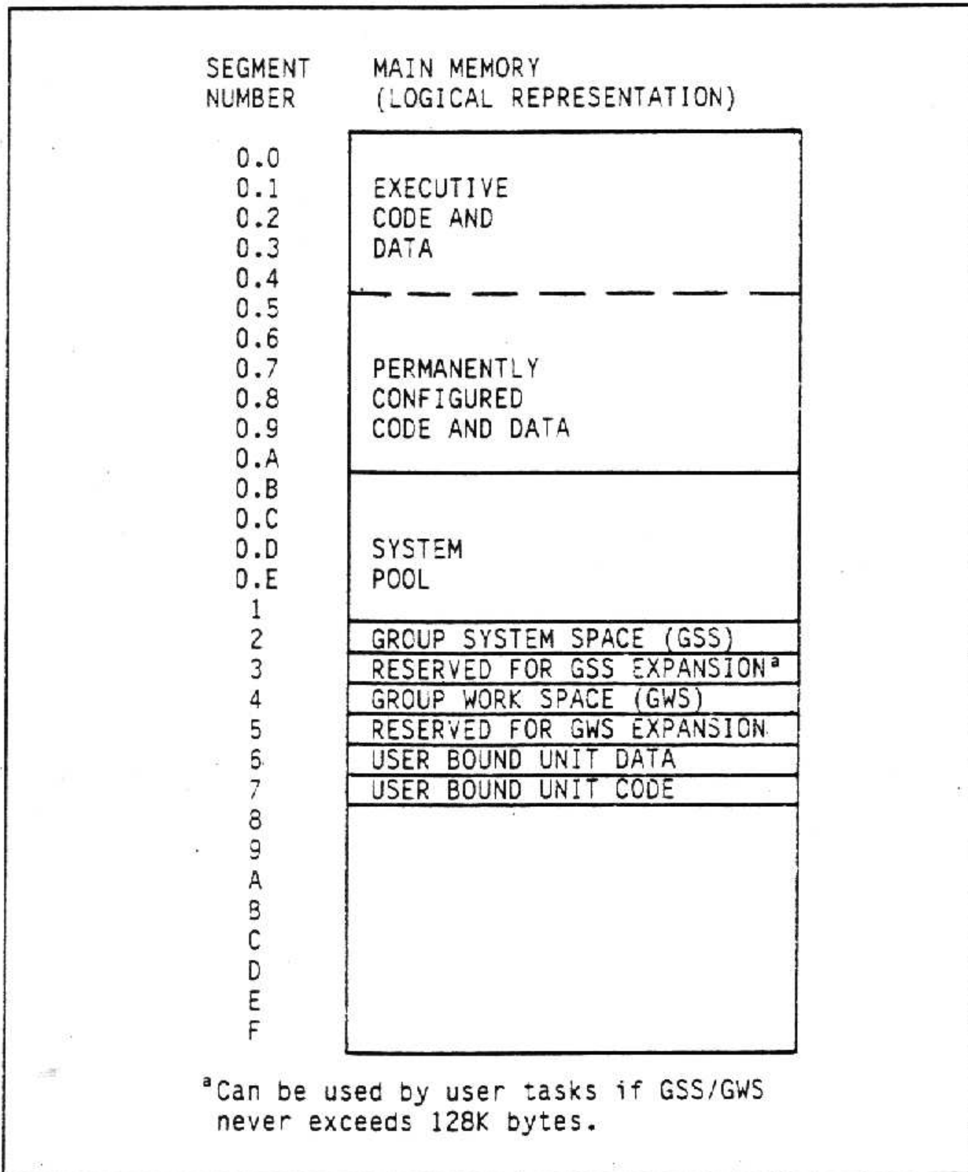

Figure 4-6. Task Address Space in BMMU System

Task Address Space In System With Extended Memory Management Unit

The following points should be noted when using Figure 4-7.

1. The layout of memory is logical, not physical.

2. The layout applies only to this example; it is possible to generate systems whose layout is different from that shown in Figure 4-7.

3. The segments available to you for your bound units are 80 through FF.

4. One copy of segments 00 through 7F exists in the system in this example. These segments contain the system global space. All tasks in the system can access these segments.

5. Segments 80 through FF are unique to the task unless they are being shared. If one of these segments is being shared, each task sharing the segment accesses the same copy of the segment. When a segment number is assigned by the Memory Manager, the lowest available segment (or segments for objects of size greater than 128K bytes) beginning with segment 80 will be used.

6. Only one copy of the group work space segment (segment 46 in this example) exists per task group. All tasks in the task group have unlimited access to this segment. Only one copy of the segment that contains group system space (segment 44 in this example) exists per task group. All tasks in the task group have read and execute access to this segment. Both the group work space and the group system space segments are dynamically expanded as demands are made on them. Each space can grow to a maximum of 256K bytes if the adjacent ascending segment descriptor has not been previously allocated to contain a task private segment.

SEGMENT NUMBER	MAIN MEMORY (LOGICAL REPRESENTATION)
00 01	EXECUTIVE CODE AND DATA
02 03 04	PERMANENTLY CONFIGURED CODE AND DATA
05 06 07	SYSTEM POOL
44	GROUP SYSTEM SPACE (GSS)
45	RESERVED FOR GSS EXPANSION[a]
46	GROUP WORK SPACE (GWS)
47	RESERVED FOR GWS EXPANSION[a]
80 81 . . FF	USER-DEFINED SEGMENTS

[a]Can be used by user tasks if GSS/GWS never exceeds 128K bytes.

Figure 4-7. Task Address Space in EMMU System