# GAME

## GAME Reference Manual

September, 1997

Bay Networks

# Contents

# Figures

# *Tables*

# About This Manual

GAME (Gate Access Management Entity) is the master control software that manages the internal functions and resources of Bay Networks Routers.

| If you want to | Go to |
|---|---|
| Learn about the general properties and design characteristics of GAME | Chapter 1 |
| Learn about gates | Chapter 2 |
| Learn about mappings | Chapter 3 |
| Learn about buffers | Chapter 4 |
| Learn about inter-gate communication | Chapter 5 |
| Learn about memory management | Chapter 6 |
| Learn about the scheduler | Chapter 7 |
| Learn about the watchdog | Chapter 8 |
| Learn about semaphores | Chapter 9 |
| Learn about the timer and time-of-day services | Chapter 10 |
| Learn about miscellaneous function calls | Chapter 11 |
| Learn about fault management | Chapter 12 |
| Learn about the system event log | Chapter 13 |
| Learn about the system loader | Chapter 14 |
| Learn about Symmetric Multi-Processing | Chapter 15 |
| Learn about performance and scalability | Appendix A |

## Conventions

| | |
|---|---|
| `screen text` | Used for examples. |

## Supplementary Documentation

The following additional documentation is available to GAME users:

| Location | Description |
|---|---|
| /rte1/harpoon/doc/game/html/ game.html | General GAME documentation. The information in game.html is not as comprehensive as the information in this document. |
| /rte1/harpoon/doc/game/html/ game_debug.html | Information on debug features. |
| /rte1/harpoon/doc/game/game_hints | GAME hints and tips. |
| /rte1/harpoon/doc/sysman/ dyn_load_usr | Dynamic Loader User's guide |
| /rte1/harpoon/doc/hardware/freI.txt | Information on the FRE |
| /rte1/harpoon/doc/bf | Information on the ARE. The debug_tips.txt file explains differences between the ARE and the FRE. |
| /rte1/harpoon/doc/debug/ debug_spec.fm.ps | Debug specification. |
| ~josswald/ct/doc/users_guide.ps | Information on ALVIN, the crash dump analyzer. |

# Chapter 1
# Introduction to GAME

## Overview

GAME (Gate Access Management Entity) is the master control software that manages the internal functions and resources of Bay Networks Routers. GAME can be compared to an operating system.

The processing entity within GAME is called a *gate*. A gate is similar to a process or thread in other operating systems, but has considerable less state.

## Properties of GAME

Some properties of GAME are listed below:

- Game can perform multiprocessing, handling thousands of gates.
- GAME performs inter-slot communications via messages.
- Multiple CPUs are contained on the same slot (SMP).
- Gates run to completion or until they give up the CPU.
- GAME incorporates FIFO scheduling. There are no priorities (except for mappings and some signal deliveries).
- There is hardware and software fault management support and isolation support for dynamic reconfiguration.
- GAME is an embedded system.

## Comparisons to UNIX

GAME is similar to UNIX in that processes have an ancestral hierarchy.

GAME is different from UNIX in the following ways:

- There is no user mode in GAME. All code runs in supervisory mode with no protection from other processes.
- UNIX kernal code does not give up the CPU except for interrupts. In GAME, there is no context switching unless a gate gives up the CPU (except for CPU and hardware exceptions, obviously).
- GAME can handle a large number of threads/processes more efficiently.
- UNIX has process priorities; GAME schedules first-come, first-served.
- UNIX can time slice; GAME does not time slice.
- UNIX handles device interrupts asynchronously; GAME only enables device interrupts at specific times.

## Design Characteristics

GAME was implemented to meet the aggressive design goals of the Harpoon project.

The new Harpoon hardware (BCN/BLN) incorporated faster, more redundant architecture.

To support the hardware, Bay required an operating system especially geared to the needs of a packet forwarder. The software used previously (VRTX kernel) did not adequately support packet forwarding.

Listed below are the key points (Design Considerations) considered by the design team and the results (Design Results) after implementation.

## Encapsulation/Decapsulation

Design Considerations

Encapsulation and decapsulation of data needs to be efficient. These are tasks performed on every packet in layer 3 forwarding. The incoming layer-2 header is stripped and a new one is added for the outgoing interface.

Design Results

The buffer format allows the data to be "suspended" anywhere in the buffer (caveat: on the FRE hardware, the start of the data must be within the first 255 bytes of the buffer). When receiving or sending a packet, there is always sufficient "headroom" left at the start of the buffer to add a larger encapsulating header. It is easy to change the start or end offset and no data copying is necessary.

It important to note that you may need to add more header info than is removed (e.g. remove an FDDI SNAP MAC but then encapsulate for ENET transmission.)

## Performance via Caches

Design Considerations

The router needs to be high performance. One way to achieve this is to make good use of the processor's instruction and data caches.

On the instruction side, locality of reference is key. If you keep executing the same code, it's likely that the next instruction will already be in the instruction cache (i-cache). However, the code that you execute repeatedly has to fit in the i-cache footprint. (Remember, the time is 1991 you've been writing code for a 68020 with a 256 byte i-cache. Luckily, the new target architecture has a 4K cache.)

For the data cache (d-cache), there are two "hot" areas: the stack and whatever data the running functions are using. For a router, the hot data is often the routing table. Once we figure out where one packet is going, it is in our best interest to do the lookup for the next packet, because we're likely to have a good portion of the routing table already in the d-cache. There is also a good chance that consecutive packets are going to similar locations (e.g., traffic burstiness; or local workstations all going to the same server). Note that this trades off individual packet forwarding latency for overall throughput.

Design Results

Each gate performs a single "step" in packet processing, and that step fits in the cache footprint (on the 040, that is). A gate gets a list of packets because this allows us to stay in the i-cache.

A buffer's destination (within the router) is written into buffer so that the forwarding code does not have to waste time clipping packets out of the list to send to different destinations.

## Packet Accesses

Design Considerations

Caching accesses to actual data packets is problematic. The packet memory is shared between the CPU, link module and PPX. (PPX refers to the four data and control channels that comprise the backplane.) If the CPU caches the packet memory, any accesses by these other entities needs to be done "cache coherently".

The CPU could flush packets from its cache, but the CPU usually (for forwarded packets) doesn't care much about the packet contents other than the header. However, packets addressed to the router (such as Telnet packets or routing updates) will be examined more thoroughly. This means the CPU would be forced to go through the motions of flushing all data from a packet to handle every possible case. Flushing isn't free, so this would impact performance.

Another aspect about packet accesses is that the header fields are usually read just once. That is, you look at the MAC header, then the IP header, etc. You don't look at the MAC header multiple times. Therefore, caching the headers isn't very useful.

Yet another consideration is with data corruption. If you get the flush wrong or miss something you'll wind up with bugs that are really hard to track down.

Bottom line: Cache coherency without HW support is a scary proposition.

For these reasons, caching the packets wasn't a requirement. However, there are those areas of packets which are accessed a lot during forwarding (packet headers). If we could improve the non-cached access performance to just those areas, we would get some bang for our buck.

Design Results

The FRE-1 and -2 provide HW assist to make packet headers faster to access. This is done by mapping SRAM onto portions of the buffer space (where the headers live) and onto the buffer headers.

## Multi-Slot Forwarding Issues

Design Considerations

We were building a multi-slot box where packets may come in on one slot but need to go out another. Should every piece of code which routes a packet need to figure out if a buffer is being delivered to the local slot or off-slot? Obviously, it would be advantageous if each one didn't. If this decision was isolated in one place, it would make most of the code simpler.

This would also improve code performance because instead of having to check and branch in all the forwarding code, the code could just say "this packet goes here" where "here" may be either local or remote (or both) and is not something the forwarding code cares about.

Design Results

The destination stored in a packet is a 32-bit "gate handle" (see Figure 2-1) that describes both the destination slot(s) and process. The protocol can do a lookup and get a 32-bit result which it doesn't have to interpret. GAME does the interpretation about what this means.

## High Availability

Design Considerations

"No single point of failure" was a design goal. So, having tightly-coupled slots where one slot could corrupt the memory of another slot, was considered a bad idea. The slots should be as independent as possible.

However, communication between processes across slots was still important. A method was necessary to know about other slots and the processes on them - when they appear and when they go away.

Software failures (e.g., bus errors) should only affect the portion of the code where the failure occurred. Taking a whole slot or box down for anything but a catastrophic software failure is not an option.

Design Results

Gate handles and mappings are used to track the existance of other processes in the router.

GAME provides resource tracking in order to clean up resources when a process dies.

GAME maintains parent/child relationships between gates. Only the offending gate and its offspring are terminated upon software failures.

### Dynamic Reconfiguration

Design Considerations

Dynamic reconfiguration is somewhat like isolating software failures to particular pieces of code. The idea is that we can add, remove, or reconfigure a protocol/interface/slot/etc. and limit the effects of that reconfiguration.

Design Results

If gates restart, only that portion of the gate hierarchy is affected. (Basically the same as High Availability.)

### Internal Code Structure

Design Considerations

We wanted to get away from one big function call tree (dev_idle) for forwarding. This is bad for maintenance and future development. This also defeats the goal of isolating the software failures to the threads that caused the problem. Ditto for reconfiguration.

Design Results

Gates are lightweight and context switching is fast. This, along with the ancestral hierarchy and resource tracking, allows software isolation.

## Limitations

The following limitations must be considered.

### Zero-Packet Loss

High throughput numbers are useless if you drop a lot of packets along the way. The effects of the control path on the forwarding path (both use the same CPU) had to be taken into account. We now do a lot of painstaking work to reduce the run times of the control path. In some cases (ISP Mode for ANS), we locate the control processes on separate, non-forwarding slots.

## Application Portability

GAME is not an easy platform to port to. Several packages have been ported to GAME with varying levels of success.

## "Cheap" Gates

Gates are not as "cheap" as once thought. With the advent of high density link modules, such as the MCT1, protocols that used lots of processes per interface broke.

# Chapter 2 Gates

## Definition

A gate is a processing entity within GAME similar to a process or thread in other operating systems, but having significantly less state.

## Attributes

### Ancestry

A spawned gate becomes a "child" of the creator.

A spawning gate is the "parent." The parent gate lives at the top of the hierarchy. Upon termination of the parent gate, all offspring also terminate. The parent of a dying gate is not notified unless it has mapped the child gate (see the Mapping chapter).

### Identification

A gate ID "names" a gate. There can be multiple instances of a gate (one per slot).

A gate handle addresses one or more specific instances of a gate (more on this later).

A gate ID is like saying "Dunkin Donuts." A gate handle tells which Dunkin Donuts you are talking about (e.g., on Great Road in Bedford or the corner of Woburn Street and Lowell Street in Lexington, etc.).

GAME maintains a gate ID table (GID Table). There are 4 bytes per gate.

## Structure

The gate structure is 128 bytes.

The most accessed parts of the structure are in the first cache

```
      line (16 bytes on 040):
BUF  *head;          /* outstanding gate message queue  */
BUF  *tail;          /* outstanding gate message queue  */
void  (*act) ();     /* gate's action routine ptr       */
u_int32env;          /* gate's state data area ptr      */
```

Parent, sibling, child links are also included.

Not allocated for ensigns or davidians (same for everything that follows).

## Activation Routine

An activation routine is a function that executes in the context of the gate instance when buffers or signals are delivered.

## Environment Pointer

As far as GAME is concerned, an environment pointer is just a 32-bit number to pass to the activation routine. In practice, it's usually a pointer to a slab of memory allocated by the gate or an ancestor. This slab is referred to as the gate's environment. One of the reasons a gate environment is needed is because GAME does not allow global data/variables.

A gate does not necessarily own its own environment. If it doesn't, the env doesn't go away when the gate dies.

## State

Note that you will not find a "state" variable in the GATE structure. A gate's state is determined by a number of things, like if it is has buffers/ signals to be delivered, if it is using the CPU, etc. A gate can be in one of the following states: dormant, awake, active, pended, or zombie.

Table 2-1 Gate States

| State | Description |
|---|---|
| Dormant | The gate is not executing and is not scheduled to run. |
| Awake | The gate has been scheduled for execution due to an event but has not yet run. |
| Active | The gate is executing. Since the scheduler is non-preemptive, there is at most one such gate at any given time per CPU. |
| Pended | The gate has voluntarily given up ownership and is waiting for an un-pending event. |
| Zombie | The gate had been deactivated but not yet removed from the system. |

## Resources

A gate can allocate and free:

- memory
- buffers
- semaphores

One may think of child gates as resources, but they are not really "owned" in the same manner as the above resources.

"Mappings" may similarly be thought of as resources, as the gate does own them.

A gate can also act as a well-known signal handler; while not really a resource, it is a state associated with the gate

These resources are reclaimed by GAME when a gate instance terminates.

Each gate also has one periodic timer.

## Identification: Gate IDs and Handles

### Gate IDs

A gate ID provides a "name" for a gate that can exist anywhere on the box.

A gate ID is 17-bits long. Bits 12-0 identify the gate number. Bits 16-13 identify the class of the gate. (See "Classes of Gates" later in this chapter for a description of gate classes.) (Bits 16-13 are commonly called the "keeper" bits due to their use in identifying the keeper slot for dynamic gates.)

Table 2-2 Gate ID Bit 16-13 Values

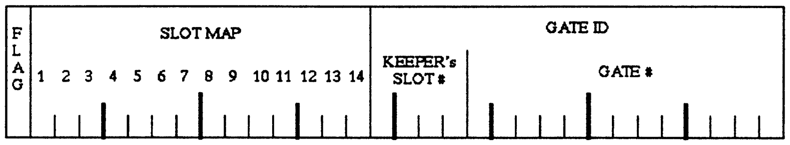
| Value | Meaning |
|---|---|
| 0 | Used to identify a well-known gate ID. |
| 1-14 | Used to identify dynamically allocated gate IDs. The value is the slot number where the gate ID was allocated (i.e., the keeper slot). This prevents multiple slots from allocating the same GIDs. |
| 15 | Used to identify gate aliases and davidians. For aliases, bits 12-9 contain the slot number of the allocating slot (i.e., the keeper bits shifted right 4 bits). For davidians, bits 12-9 are also set to 15. |

## Gate Handle

A gate handle contains both a gate ID (name) and its instantiation information.

A gate handle is 32-bits long. Bits 16-0 specify the gate ID. Bits 30-17 specify the slot MAP. A gate handle is illustrated in Figure 2-1.

Figure 2-1. Gate Handle



### Slot Map

The slot map is a bit mask of slots where this gate is instantiated, as defined below:

```
bit 30: slot 1
bit 29: slot 2
:
bit 17: slot 14
bit 31: FLAG bit
```

In mappings, the Flag bit indicates that the GID is allocated. In buffers, this indicates a reliable transport primitive. (See Figure 4-1.)

The format of a davidian gate handle is slightly different (later).

Note the difference between the keeper ID in a gate ID and slot bit in the slot map. The keeper ID tells you from which slot's space the GID came from. The bit in the slot map tells you which slots have an instantiation of the gate. In practice, multiple slot instances only occur for well-known gates, which have a keeper ID of zero, and aliases, which have a keeper ID of 15. For dynamic gates, the only map bit you will see set is the one corresponding to the keeper ID.

## Processing Macros

Some important macros for processing gate handles are defined in:

```
(include/kernel.h and include/game.h)
```

The macros for examining gate handles are described in Table 2-3.

Table 2-3 Macros For Processing Gate Handles

| Macro | Function |
|---|---|
| GH_IS_LOCAL (gh) | Is the gate instantiated on my slot? |
| GH_IS_PRESENT (gh) | Is the gate instantiated on any slot? |
| GH_IS_REMOTE (gh) | Is the gate instantiated on any non-local slot? |
| GH_IS_USED (gh) | Is the GID allocated (in the GID table)? |
| GH_GET_SLOT_MAP (gh) | Isolate the slot bits (30-17). |
| GH_GET_GID (gh) | Isolate the GID bits (16-0). |
| GH_IS_ALIAS (gh) | Is this an alias GID? |
| GH_IS_DAVIDIAN (gh) | Is this a davidian GH? |
| G_MY_SLOT_MASK | The local slot bit. |

A macro for setting gate handles is described in Table 2-4.

Table 2-4 Macro For Setting Gate Handles

| Macro | Function |
|---|---|
| GH_SET_LOCAL (gid) | Form a GH with the local slot bit and the gid. |

## Gate Instance Management

The g_req() system call implements almost all aspects of gate instance management.

**Call:**

```
GID g_req (GID gid, void (*action) (void *, BUF *, SIG),
             void *environment, u_int32 flags)
```

**Values:**

"gid" is the gate ID that the call applies to. The value of gid can be:

- G_REQ_NEW_GID (allocates a new gate ID from the slot's space)
- G_SELF_ID (the calling gate)
- any valid gate id (instantiated dynamic gate ID or well known).

"action" is the routine that is called when the gate instance is activated. The value of action can be:

- G_NOACT (if gid = G_REQ_NEW_GID, create an ensign gate)
- G_DAVIDIAN (if gid = G_REQ_NEW_GID, create a davidian gate)
- G_REQ_KILL (terminate the local instance of this gate)

- any valid function address identifying the new action routine for the gate instance.

"environment" is a 32-bit value. This is passed to the gate instance upon activation. In practice, this is a pointer to a slab of memory (the environment) associated with the gate.

"flags" are optional gate management functions. The value of flags can be:

- G_SIG_INI (send an initialization signal (SIG_INI) to a newly instantiated gate instance)
- G_REQ_INI (send an initialization signal (SIG_INI) to an existing gate instance - not a good idea to use this)
- G_REQ_SOLO (perform the soloist election procedure prior to creating the gate instance - more on soloists later)
- G_NO_SIG_INI (another name for "zero")

There are some additional flags associated with SMP which will be discussed in the section on the Scheduler.)

**Return Values:**

For all successful calls, the gate ID of the created/modified/terminated gate instance is returned.

If you tried to kill a gate that is already dead, a zero is returned. For severe errors, the calling gate is terminated.

**Examples:**

Example 1 shows how the IP routing table manager (RTM) creates a network interface (NWIF) gate:

```
gid = g_req (G_REQ_NEW_GID, ip_nwif_init_act, nwif_env,
G_SIG_INI);
```

Note that the RTM has already allocated the NWIF's environment. The NWIF gate never changes its action routine or environment (unfortunate name used for this routine...).

Example 2 shows how the RTM starts BGP in both ISP (soloist) and non-ISP modes.

```
  if ( rtm_env->bgp_soloist )
  {
    if ( GH_IS_REMOTE(rtm_env->bgp_gh) )
    {
      /* We are in soloist mode, and a soloist is already
running on another * slot, don't start the soloist on the
local slot
       */
      return;
    }
    /* start the soloist BGP */
    g_req (GID_BGP, bgp_init, rtm_env, G_SIG_INI |
G_REQ_SOLO);
  }
  else
  {
     /* start the replicant BGP */
     g_req (GID_BGP, bgp_init, rtm_env, G_SIG_INI);
  }
```

When the BGP init strip runs, it changes both its action routine and environment:

```
g_req (G_SELF_ID, bgp_active, bgp_env, G_NO_SIG_INI);
```

The environment is always overwritten. So, if you're changing the activation, you'll need to pass in the old value of the environment if you don't want it to change.

RTM will kill the BGP gate if it learns that the BGP code base is being unloaded:

```
g_req (GID_BGP, G_REQ_KILL, 0, 0);
```

## Normal Activation

A gate can be activated via its action routine for two reasons:

- buffer delivery (multiple buffers can be delivered in a list)
- signal delivery (a single signal can be delivered in one activation)

Both cannot happen at the same time. Two separate gate activations will occur if this is the case.

A gate action routine must be of the following form:

**Call:**

```
void gate_act (void *environment, BUF *buffers, SIG signal)
```

**Values:**

"environment" is a pointer to a slab of memory, as set by the most recent g_req() call.

"buffers" is a pointer to a list of buffers that were sent to the gate, or NIL. If "buffers" is NIL, "signal" is the value of the signal delivered to this gate. If "buffers" is valid, "signal" is undefined.

**Considerations:**

The type of activation is determined by the value of the "buffers" parameter, which MUST be checked first. If buffers is NIL, a signal is being delivered via the "signal" parameter:

```
if (buffers)
{
    /* process list of buffers */
}
else
{
    /* process signal */
}
```

Once activated, the gate holds the CPU until it does one of the following things:

1. It exits the activation routine. This completes the current activation of the gate. It will not be activated again until buffers or a signal is sent to it. If buffers or a signal were sent during the current activation, the gate is immediately rescheduled, at the end of the scheduler queue. A gate will not see any new buffers or signal until it exits its activation routine.
2. It relinquishes the CPU either explicitly or implicitly through a system call. The gate will go into "pended" state until re-awakened by GAME.
3. It terminates itself either via g_req() or a system FAULT. In this case, GAME cleans up all resources owned by the gate and the instance no longer exists.

Some important things that will be covered more throughly later:

- A gate must "do something" with each buffer delivered in an activation.
- A gate must not hold the CPU for more than 4seconds (although this value is really performance dependent).

## Classes of Gates

There are several classes of gates in GAME:

- well-known
- dynamic
- ensigns
- davidians
- aliases

### Well-Known Gates

Well-Known gates are usually at or near the top of a subsystem hierarchy of gates. The ID has to be fixed so that communication can occur between slots (example: IP Routing Table Manager). Circuit gates are also well-known (circuit number + 1024). There are 8K IDs available, not including circuit numbers (about 360 used) The gate ID always has the "keeper" bits set to zero.

### Dynamic Gates

Dynamic gates comprise the bulk of running gate instances in a system. There are 8191 gate IDs per slot.

The gate ID always has the "keeper" bits set to the slot number.

## Ensign Gates

Originally created for the MIB service to represent states in the MIB (e.g., the current values of the read/write objects in a row of a table).

The name "ensign gate" is derived from the usage - it's a flag to indicate the presence of something.

An Ensign gate allocates a gate ID but does not create an instance.

An Ensign gate can be used to represent state (e.g. MIB service).

An Ensign gate is visible across slots via mappings.

An Ensign gate is not hierarchically attached to creator gate (no GATE structure). Therefore, GAME cannot reclaim ensigns if the creator dies (not a problem with MIB service - whole slot resets anyway).

An Ensign gate uses the same GID pool as dynamic gates (well-known IDs can't be ensigns).

The Ensign gate ID always has the "keeper" bits set to the slot number.

Use the following call to create an Ensign gate:

```
ensign_gid = g_req (G_REQ_NEW_GID, G_NOACT, 0, 0);
```

Use the following call to kill an Ensign gate:

```
g_req (ensign_gid, G_REQ_KILL, 0, 0);
```

## Davidian Gates

Davidian gates are a lot like ensigns except:

- a whole lot more are available (about 8 Million per slot)
- Not visible across slots via mappings
- Invented to replace ensigns for the MIB service:

Davidians allow representation of many more "states".

Davidians don't use/waste the slot's limited dynamic gate ID space.

In the Gate Handle the slot map space is used for extention of the Gate ID.

In the Gate ID the keeper bits are set to 15; bits 12-9 of gate number are also 15.

Use the following call to create a Davidian gate:

```
davidian_gid = g_req (G_REQ_NEW_GID, G_DAVIDIAN, 0, 0);
```

Use the following call to kill a Davidian gate:

```
g_req (davidian_gid, G_REQ_KILL, 0, 0);
```

## Alias Gates

Alias gates are used to group instantiated gates. When a message gets sent to an alias, a copy of that message is made for each member of the alias. This works only for unreliable messaging. Reliable messaging to aliases isn't supported. (covered throughly later).

In practice, only well-known gates and aliases will appear with instances on multiple slots.

## Soloists

Some well-known GAME gates require that only one instance runs in the box at a time.

Examples:

- circuit gates (multi-line scenario)
- Technician Interface (TI)
- OSPF
- BGP in ISP-mode

The Soloist mechanism is explained in detail below, although the mechanism really isn't that important to a user of GAME.

When a gate wishes to create a soloist, it must first ensure that the gate does not exist currently on any other slot. This is done via a mapping (covered exhaustively in the Mapping chapter). If the gate does not exist on another slot, the g_req call is made with the G_REQ_SOLO flag to start the soloist election. (See the BGP example above.)

GAME formats a message that contains the proposed gate handle of the soloist and the gate handle of the prospective parent of that soloist (the one that called g_req). This is sent to the Keeper Gate on every live slot (including the local slot).

When each Keeper gate receives this message, it first checks to see if the GID is locally instantiated. If so, it replies to the message with the gate handle for that gate PLUS the FLAG bit (bit 31 - this is key later!). Otherwise, it checks to see if there is a soloist election active for that GID. If so, it replies with the gate handle of the FIRST parent gate that it heard from. If there is no current election, a structure is created to represent the soloist election for that GID. The Keeper replies with the parent gate handle it received in the message.

The calling slot examines the replies, ORing-ing together the slot bits of the gate handles. If the local slot is the only bit set or the lowest numbered (leftmost) slot bit set, the soloist gate is created. After the election is lost or the soloist gate is created, another message is sent to all Keeper gates to clean up the soloist election state.

If one of the Keeper gates indicated that it had a local instantiation of the GID, the setting of bit 31 prevents any other slot from winning the election. This is because bit 31 looks like slot "0", which is a lower slot number than a real slot.

The gate handle slot bits are set as follows::

```
bit 31: FLAG bit (acts like "slot 0")

bit 30: slot 1

bit 29: slot 2

etc.
```

## Dueling Soloists

There is a chance, especially on a busy router, that the soloist election mechanism will fail and allow multiple soloists to be created. This was first discovered when multiple TI processes would arise and try to control the console, a situation known as "dueling TIs".

To guard against this, a gate that creates a soloist must maintain a mapping of the soloist GID and kill its local soloist if another soloist appears that has a lower slot number (details when we get into mappings). (Actually, you always need a mapping for a soloist because the g_req (solo) is a one-time event. If the current soloist dies, a new election needs to be held but the kernel doesn't do this automatically.

Soloist elections are independent per Gate ID.

In a booting router with one flash card, all soloists appear on the slot with the flash card (unless that slot is not "eligible" to run a particular soloist - controlled by configuration information). This is merely because that slot gets the code running first.

## Aliases

GAME can associate a single gate ID with multiple gate instances on the same slot and across slots. This allows unreliable buffer delivery to multiple gates using a single delivery primitive. A gate can become a member of an alias using the call g_alias(). The same call is used to remove a gate as a member. Members can be added or removed only on the slot where the "real" gate instance lives. Note that the member gate itself does not have to be the one to call g_alias().

In the Gate ID of an alias the keeper bits are set to 15. Bits 12-9 contain the slot number of the allocating slot (i.e., the keeper bits shifted right 4 bits).

When the first member of an alias is added on a slot, GAME will "turn on" that slot bit for any mappings of the alias gate id (GID_GAME is a legitimate first member). When all members are removed from the alias on a slot, GAME "turns off" the slot bit.

WARNING: When all members are removed from the slot on which the alias was created, GAME frees the alias gate ID. If you don't want the alias to be freed, add GID_GAME as a member. This will keep the alias allocated.

### Aliases of Aliases Method

The following call illustrates the aliases of aliases method:

With the advent of g_xmt_im(), aliases of aliases really shouldn't be needed any more. g_xmt_im() solves the same problem but in amore eficient manner. See Chapter 5 for a description of g_xmt_im().

**Call:**

GID g_alias (GID alias, GID gid, u_int32 mode)

**Values:**

The value for "alias" can be:

- G_ALIAS_NEW (allocate a new alias GID)
- G_ALIAS_ALL (requests removal from all aliases, in which case "mode" must be G_ALIAS_DEL)
- an existing alias ID.

The value of "gid" can be:

- the GID of "real" member gate instantiation on local slot
- GID_GAME. GID_GAME can be used as the first member when creating an alias in order to create a "permanent" alias that stays around even when all of the "real" members have gone away.

The value of "mode" can be:

- G_ALIAS_ADD: add member to alias set
- G_ALIAS_KILL: destroy entire alias set on the local slot
- G_ALIAS_DEL: delete member from a given alias set or all sets
- G_ALIAS_COUNT: count and return number of members of an alias
- G_ALIAS_NUM: returns the number of free aliases remaining on the local slot
- G_ALIAS_ALIAS: add an alias to an alias. The idea of adding an alias to an alias was created for the bridge code to simplify multi-slot alias maintenance. There are a couple of caveats about an alias that contains aliases as members:
    - It can only have one alias as a member on a given slot.
    - It cannot have any "real" gates as members on that slot.

## Bridge Use of Alias to Alias

Here's how the bridge uses an alias to alias:

All of the bridge encaps gates on a slot are added to a local alias. Suppose we have slots 2, 3, and 4, and they create the aliases A2, A3, and A4, respectively. All of slot 2's encaps gates are members of A2. Ditto for slot 3 / A3 and slot 4 / A4.

The alias IDs are broadcasted to the bridge gates on each slot, and each slot then adds its local alias to the other slots' aliases:

```
slot 2: adds A2 to aliases A3 and A4

slot 3: adds A3 to aliases A2 and A4

slot 4: adds A4 to aliases A2 and A3
```

When slot 2 wants to flood a packet, it sends it to alias A2. Since all of the local encaps gates belong to A2, they each get a copy of the packet. Since A3 and A4 belong to A2, a copy of the packet is sent to slots 3 and 4. When the packet arrives on the remote slot, it is replicated and sent to all of the members of the local alias.

The example below creates a local bridge flood alias. Note the use of GID_GAME as the first member. This ensures that the alias will not go away. Since GID_GAME is 0, no packets actually get delivered to this "member".

```
  dp_env->enet_flood_gh = g_alias (G_ALIAS_NEW, GID_GAME,
G_ALIAS_ADD);
```

This example adds an "encaps gate" to an existing flood alias:

```
  g_alias (GH_GET_GID (dp_env->enet_flood_gh),
      GH_GET_GID (ccb->lb_encaps[enet_index].isap_handle),
          G_ALIAS_ADD);
```

Members are never explicitly removed from this alias. GAME removes the encaps gates if they die.

Here's how the local alias is added to the flood alias of another slot:

```
          switch (flood_info->domain_id)
                    {
                    case ENET_FLOOD_DOMAIN:
                      our_flood_gh =
dp_env->enet_flood_gh; break;
          if (GH_IS_LOCAL(our_flood_gh))
     {
                    :
                    :
      g_alias(GH_GET_GID(flood_entry->flood_gh),
                                   GH_GET_GID(our_flood_gh),
                                   G_ALIAS_ALIAS);
                    }
```

There is a special version of the unreliable buffer delivery primitive (g_xmt_im) that sends buffers to all members of an alias except one (the exception is usually associated with the sender). This is covered in the Inter-Gate Communication chapter.

# Chapter 3 Mappings

## What's a mapping?

A mapping is a way that a gate can keep track of the state of any and all instances of a particular gate ID. Simply put, a mapping lets a gate know when an instance of some gate is created or terminated.

## Why?

Mappings are the primary way to deal with software and hardware reconfigurations and failures.

Examples of mappings are as follows:

- MIB service uses davidians to represent current database state.
- Instances of well-known gates map each other to learn "slot-up/down" events.
- Per-interface gates map the underlying circuit to learn about "circuit up/down" events.
- Parent gates map their children to learn of their demise and possibly do clean-up and restart.

## Function Call

A mapping exists independently in the gate of its creator (sometimes known as the "owner" of the mapping). It is created via the g_map() call:

**Call:**

```
void g_map (GID gid, GH *gh, void (*map_activation) (GH *,
GH) )
```

**Values**:

"gid" is the gate ID to map.

"gh" is a pointer to the local copy of a gate handle.

"map_activation" is the mapping activation routine to call when a change occurs. The value of map_activation can be:

- G_NOACT: no activation routine. (Uses the system default routine which just updates the *gh.)
- G_UNMAP: terminate an existing mapping
- G_CURRENT_GH: just return the current gate handle without creating a mapping
- A valid function address for the activation routine for the mapping

## Mapping to Retrieve the Current GH

Sometimes a gate just needs to know where the current instances of a gate exist. This is normally used if you want to send a message to a gate whose state you don't track continuously.

```
g_map (some_gid, &(env->some_gh), G_CURRENT_GH);
```

GAME writes the gate handle for some_gid at the time of the call into env->some_gh.

GAME maintains no further state.

The next example is from the RSVP Interface (RIF) gate, where buffers have to be delivered to a control gate (GID_RSVP_CONTROL) on one slot (the soloist) only. A second gate (GID_RSVP_SOLO) exists to mark this slot.

```
/* find out where is the SOLOIST */ g_map(GID_RSVP_SOLO,
&solo_control_gh, G_CURRENT_GH);


/* send these buffers to the CONTROL gate on the soloist
slot */      solo_control_gh = (solo_control_gh &
~GID_RSVP_SOLO) | GID_RSVP_CONTROL; rif_env->fwdlist_id =
g_fwd_list (solo_control_gh, fwd_blist, fwd_blist_tail, 0);
```

## Mapping With No Activation Routine

A gate can instruct GAME to maintain a gate handle for a particular gate id, updating that gate handle whenever there is a state change. This is normally used when a gate sends messages to instances of a well-known gate but does not need to do any processing based on the up/down state transitions of those gate instances.

```
g_map (some_gid, &(env->some_gh), G_NOACT);
```

GAME creates state regarding the mapping, including:

```
the gate handle pointer (*)

GID of the mapped gate (*)

GID of the owner gate

the mapping activation routine (G_NOACT, in this case)
```

The asterisked items are the items that index the mapping state. Therefore you cannot have multiple owner gates mapping the same target gate using the same physical gate handle.

Upon initial mapping, GAME fills in gate handle:

Gate ID equals the gate ID requested.

Each bit in the slot map is set if the corresponding slot contains an instance of the gate.

FLAG bit is set if any slot has allocated the GID.

### Caveats

- Ensign gate slot bits appear for remote instances even though there really is no "instance" of the gate on that slot. This was the only way to get multi-slot mappings of ensign gates to work.
- A davidian gate slot map field is not applicable (part of gate ID).

- Game updates the FLAG and slot bits whenever there is a state change.
- If an instance terminates the slot bit is cleared; if an instance is created, the slot bit is set.
- If all instances terminate/GID not allocated, the FLAG bit is reset.

The memory used to hold the gate handle MUST be in a block where the allocation is via g_malloc(). The only case where you can use stack space for a mapped gate handle is for a G_CURRENT_GH call.

- When the mapping owner no longer cares about the gate handle of the mapped gate, it MUST call g_map() to remove the mapping.
  g_map (some_gid, &(env->some_gh), G_UNMAP)

  Otherwise, GAME will continue to update the memory where the gate handle was located. This is particularly dangerous if that memory was freed and then allocated by another gate!
- GAME will clean up after mappings if the owner dies.

### Example

This example is from the IP Policy gate, which doesn't really care if BGP or OSPF are up, other than to be able to send messages about changes in routing policies:

```
g_map (GID_IP_OSPF, &(ip_policy_env->ospf_gh), G_NOACT);

g_map (GID_BGP, &(ip_policy_env->bgp_gh), G_NOACT);
```

## Mapping With An Activation Routine

Usually, a mapping is done because a gate wants to perform specific actions when another gate instance goes up or down. This can be done by specifying an action routine to execute upon a state change.

```
g_map (some_gid, &(env->some_gh), mapping_activation);
```

GAME again creates mapping state, including the action routine.

During the initial g_map() call, GAME suspends the current gate context, creates a temporary gate to run the initial mapping activation, and executes that gate IMMEDIATELY. This means that "mapping_activation" runs BEFORE the g_map() call returns!

**Form:**

A mapping activation routine must be of the following form:

```
mapping_activation (GH *gh, GH new_gh)
```

**Values:**

"gh" is a pointer to the gate handle, as passed in the second parameter to the g_map() call.

"new_gh" is the new value of the gh.

**Considerations:**

Unlike a mapping without an activation routine, GAME does NOT set *gh to the new gate handle value. It is up to the activation routine to do this after comparing the new value to the previous value (to see what changed). GAME does set *gh equal to the GID (no slot bits set) before calling the activation routine for the first time. new_gh is set to the current state of the GID. This will include the FLAG bit (31) if the GID is allocated on any slot.

Suppose a gate maps GID_DP_INI (16), and that gate currently exists on slot 2, 3, and 4. The initial activation parameters would be:

```
*gh     = 0x00000010     (GID only; no slot bits set)
new_gh  = 0xb8000010     (FLAG bit, slots 2-3-4, GID)
```

Suppose an ensign gate with gate id 16400 (0x4010) is mapped on slot 2, and that ensign gate is currently allocated (but, obviously, not instantiated) on slot 2. The initial activation parameters would be:

```
*gh    = 0x00004010      (GID only)

new_gh = 0x80004010      (FLAG bit turned on)
```

If the mapping is done on another slot:

```
*gh    = 0x00004010      (GID only)

new_gh = 0xa0004010      (FLAG bit turned on + slot 2)
```

The presence of the slot bit is an unfortunate side-effect of being able to map ensigns across slots. The mapping routine should only check for the presence of the flag bit (see the GH_IS_USED macro later) and ignore the slot bits.

From this point on, GAME will call the activation routine every time an instance of that gate is created or destroyed. For ensigns/davidian, the routine is called when the GID is allocated or deallocated. Mapping activation routines get scheduled ahead of any other gates scheduled for buffers or signals (more on this in the Scheduler section).

A single mapping activation for a well-known gate or an alias can contain MULTIPLE slot bit transitions. For example, a later activation of the mapping for GID_DP_INI might receive the parameters:

```
*gh    = 0xb8000010  (FLAG bit, slots 2-3-4, GID)

new_gh = 0xb2000010  (slot 4 instance went away,
slot 6 instance came alive)
```

The memory used to hold the gate handle MUST be in a block allocated via g_malloc(). The only case where you can use stack space for a mapped gate handle is for a G_CURRENT_GH call.

When the mapping owner no longer cares about the state of the mapped gate, it MUST call g_map() to remove the mapping:

```
g_map (some_gid, &(env->some_gh), G_UNMAP);
```

Otherwise, GAME will continue to call the mapping activation routine. If the unmap is done within the mapping routine itself (not uncommon), the activation terminates before the return from g_map(). That's right, you don't return.

It's also important to unmap when the mapping fails because a dynamic gate has died. This is because that gate ID will be reused eventually for a different purpose and your mapping of it is no longer appropriate.

Always unmap before freeing a GH.

Within a mapping activation routine, there are a collection of macros that are used to examine and compare the old *gh and the new_gh:

Table 3-1 Mapping Macros

| Macro | Purpose |
|---|---|
| GH_IS_USED (new_gh) | Is this ensign/davidian allocated? |
| GH_BECAME_LOCAL (*gh, new_gh) | Was an instance created on this slot? |
| GH_BECAME_REMOTE (*gh, new_gh) | Was an instance created on another slot? |
| GH_BECAME_PRESENT (*gh, new_gh) | Was an instance created on any slot? |
| GH_CEASED_LOCAL (*gh, new_gh) | Did an instance die on this slot? |
| GH_CEASED_REMOTE (*gh, new_gh) | Did an instance die on another slot? |
| GH_CEASED_PRESENT (*gh, new_gh) | Did an instance die on any slot? |

The return values of these macros are the slot bits of the applicable gate instances.

When a mapping activation routine runs, it is a separate "thread" from the base context of the gate. This is done by creating a new, temporary gate. This gate is setup to be a child of the mapping creator who called g_map). This gate only exists while the mapping is active. In some cases, resources allocated by the mapping activation belong to the owner gate. In other cases, the temporary mapping gate owns them. Resources are treated as follows:

- Memory: All allocated memory becomes property of the owner gate.
- Buffers: Transient buffers are part of the mapping gate. However, any use of the private pools (e.g., g_bsave) are in the context of the owner gate. A mapping gate cannot exit with buffers on its transient pool. The owner gate will be terminated if this happens. (Note that this is a more drastic punishment than if a normal gate activation orphans buffers. In that case, only a message is logged.).
- Semaphores: A created semaphore is the owner's property. A token acquired by a mapping gate belongs to that gate.
- g_req() calls: Any gates created in a mapping routine are children of the owner gate.
- g_map() calls: Any mappings created in a mapping routine are owned by the owner gate.
- g_isr() calls: Any signal handling requested in a mapping routine is registered in the context of the owner gate.

**Additional Considerations:**

If you call g_myid(), you get the owner's gate ID.

If you call g_env(), you get a pointer to the owner's environment. A mapping gate cannot have its own environment.

A single gate can have its base context and several mapping contexts in the active/pending states at once. Watch out for races if accessing data structures shared between these contexts! Semaphores or other locking mechanisms are necessary in these case (and the know-how to use them!).

Finally, before a mapping activation routine exits, it must update the allocated gate handle:

```
*gh = new_gh;
```

## Some Mapping and Unmapping Gotchas

One problem is when a routine that did a mapping with G_NOACT, but [] did not do a G_UNMAP. Is an "unmapping" (i.e. g_map (gid, gh, G_UNMAP)) necessary in this case?

Yes, an unmap is necessary in all g_map cases before you free a gate handle. Here's the scenario: Gate 1 allocates memory location 0xabcdef for a *GH. It then maps to the gate and passes g_map G_NOACT for no action routine (usually done to check for a gates existence). GAME saves that Gate 1 maps some gate at location 0xabcdef. Gate 1 frees memory location 0xabcdef with out doing an unmap. Gate 2 allocates memory for a *GH and happens to get the same location 0xabcdef. Gate 2 then places a GH into that location. Game thinks that another gate (gate 2) is mapping the same gate that gate 1 mapped. It triggers a mapping change for both Gate 1 and Gate 2's mapping action routines (Gate 1's action routine was G_NOACT so nothing happened. If it wasn't, bad things could have happened here.) Game allows for gates to modify GH's through mapping routines even though they are not the owner of the mapping. It does, however, log a message that says, in so many words, that this is happening.

Gate 2 thinks that it owns the mapping of the GH at location 0xabcdef. Game thinks Gate 1 owns it because gate 1 never unmapped that location. When Gate 2 does the unmap, Game panics in map_rem because gate 2 is not the owner of the mapping at that location. When Game unmaps, it compares the info it saved on the mapping creation with the info from the gate calling unmap. Remember, the info saved is the GH of the gate that created the mapping and the memory location containing the GH of the mapping. In this case, Game first checks to see if location 0xabcdef has a mapping associated with it. It then compares the GH of the creator of the mapping with the GH of the caller of unmap. If they don't match then GAME panics. The situation is potentially even more dangerous if Gate 2 has used the memory for something other than storage for a mapped gate handle. It would be possible for Gate 1's mapping to then corrupt Gate 2's memory, resulting in unpredictable system behavior,

Remember, the info saved is the GH of the gate that created the mapping and the memory location containing the GH of the mapping. In this case Game first checks to see if location 0xabcdef has a mapping associated with it. It then compares the GH of the creator of the mapping with the GH of the caller of unmap. If they don't match then game panics.

Just to be a bit more precise with the terminology (the difference between a gate handle and gate ID can be very important). For a given mapping, GAME keeps track of:

- the gate ID of the creator of the mapping (it knows that it is on the local slot, so the slot bits are not necessary)
- the location of the gate handle in memory the gate ID of the mapped gate.

A gate has a chain of mapping blocks. When an unmap is done, GAME searches this list for a match of the GH memory location and the GID of the mapped gate (the mapping owner GID is implied, since you are looking in the gate's list). If a matching entry is not found, you get the "don't own such mapping" log message.

To make a long story short - always unmap BEFORE you free a GH that is mapped, no matter what the action routine is.

There are some cases where you cannot do this. If you do the G_UNMAP in the actual mapping routine for the gate, you do not return. If you had allocated the GH memory dynamically, you need to free it before the unmapping. This should be done immediately before the unmapping in order to reduce the possibility that someone will introduce a window later. The GATE ID (gid) argument passed to the g_map() function should not be extracted/derived from the memory that was being used to hold the value of the current gate handle after it has been freed. Any such access to already freed memory is risky and should be eliminated. The gid value should be extracted from the memory and stored in a local variable before the memory is freed. Alternatively in a mapping routine itself, the gid can always safely be extracted from the 'new_gh" argument passed in by GAME when it triggered the mapping, provided it has not been modified.

When doing an unmap within a mapping routine, the g_mfree must be done first because mapping routines do not return from an unmap call. This can be done because g_mfree does not pend. Be careful as to the owner of the g_malloc()'d memory in this case.

## Mapping An Alias

Since alias gate instances don't really exist, GAME handles the mapping of alias gate IDs somewhat different from "real" gates. However, to the application using the gate handle supplied by a mapping, it doesn't make any difference.

When the first gate on a slot joins an alias, GAME considers this an "up" event for the alias instance on that slot and turns on the slot bit in the gate handles for any mappings of the alias gid. Further member additions on that slot do not cause any state change. When the last member removes itself from the alias on a slot, game considers this a "down" event and it removes the slot bit from the gate handles for any mappings of the alias gid.

Suppose, on slot 2, g_alias() is called to add gate A to alias 0x0001e401, the first member on the slot and on the box. Any gate mapping gid 0x0001e401 would see the following state change:

```
*gh = 0x0001e401, new_gh = 0x2001e401
```

Gates B, C, and D also join on slot 2. No state change occurs.

Now suppose gate E on slot 5 was added:

```
*gh = 0x2001e401, new_gh = 0x2401e401
```

Finally, gate E on slot 5 is removed:

```
*gh = 0x2401e401, new_gh = 0x2001e401
```

## Some Frequently Asked Questions

```
*** Can I map "myself"?
```

Yes. Self-mappings are not only allowed but are even subject of a special treatment. They execute ahead of all the other mappings that may be triggering at the same time and ahead of gate termination clean-up. Due to the fact that a it is owned by a gate that has been marked "dead" and will soon be removed from the system, a self-mapping is restricted in what it can do. For instance, it may not pend, which it would do if it tried to create a gate or a mapping, attempt to allocate a buffer with a pending option, call g_delay() or g_idle(), or send a reliable message. It may send an unreliable message, however. Usually, all a self-mapping does is log a message and/or update some MIB statistics.

`*** Can I map across slots?`

Yes. Mappings were invented specifically to track gate state changes occurring on all slots and to update gate handles.

`*** Can I map "ensign" and "davidian" gates?`

Yes. Ensign and davidian gates were invented specifically to be mapped. There is little else they can do for you.

`*** If I have two mappings, which one runs first?`

The order of mapping triggering and execution is inherently unpredictable. GAME does not specify which mappings run first except for the self-mapping's special treatment described above.

`*** Will a mapping trigger while the owner gate pends?`

Yes. Mappings with activation routines execute as temporary gates which are children of the mapping owner gate. After the initial activation, the rest of triggered mapping life is just a life of a gate. This may also lead to certain race conditions when the triggered mapping affects an environment shared with its owner gate (or other gates, for that matter).

`*** Can my mapping activation combine several state changes?`

Yes. In larger configurations it is possible, even likely, that one trigger activation will cover almost-simultaneous gate instantiation on several slots. Note that one given mapping activation may SIMULTANEOUSLY cover gate instance creation and disappearance on different slots (both GH_BECAME_PRESENT and GH_CEASED_PRESENT may be non-zero!).

`*** Will my mapping run twice with the same "new_gh"?`

No. This used to be possible, but it has been fixed. However, your mapping routines should work correctly if this should happen (i.e., bulletproofing is a good thing).

`*** Can I miss a trigger?`

No. The events for a map trigger are queued up and the mapping is guaranteed to see every transition. However, if triggerings are queued up, the state given in new_gh is not necesarily the current state.

`*** What happens if I do a suicide in a mapping routine?`

This terminates the current mapping, the owner gate, and all other mappings. Effectively, all context related to the owner gate is terminated. This is true for any gate termination encountered by a mapping gate (other than a normal, clean exit).

## Activation Routines for Well-Known Gates

A mapping of a well-known gate is usually one that hangs around for the life of the owner. The typical use is within a subsystem. It allows the local subsystem component to learn when the subsystem comes up or goes down on other slots. The "up" processing usually involves synchronizing the data between slots. "Down" processing cleans up information learned from that slot.

A side-effect of the mapping is that it provides a gate handle to use when the subsystem wants to send something to all other slots. The local slot bit is first removed via an AND with ~G_MY_SLOT_MASK.

There are two approaches that are used for well-known gate mappings (actually, this applies to all mappings) to deal with synchronization issues with the base gate context and other mapping contexts. Each of these has its pros and cons:

### Data Signal Approach

One approach is that the mapping does nothing other than send a data signal to the owner gate (most common approach).

The advantages are:

- You don't have to worry about multiple gate threads accessing and modifying the same data.
- The gate handle maintenance is separated from the up/down processing, resulting in timely updates of the gate handle (although, in some cases, having the GH out of sync with the up/down processing can cause problems).
- This fits in well with how GAME expects mappings to act (short lived, no pending).

The disadvantages are:

- A temporary resource (memory) has to be allocated.
- The event does not get processed until the base gate context is scheduled for the signal. If the gate is currently active or has another activation on the scheduler queue, it may not get the signal in a timely fashion.
- The base gate context becomes a bottleneck, having to do all of the normal buffer and signal processing along with the mapping events.

The magnitude of these downsides all depends on the workload of the gate. It may not matter if there is little work to do. In any case, if the developer has no experience in developing multi-threaded code (the test: do you know what a "critical section" is?), they must use this approach.

The well known gate GID_IP_RTM uses this approach. It maps a second well-known gate called GID_IP_RTM_UP to determine connectivity to its peers on other slots. The mapping routine:

void

```
ip_rtm_up_self_map(gh, new_gh)
     GH *gh;
     GH new_gh;
{
  char cbuf [80];
  RTM_ENV *rtm_env = (RTM_ENV *) g_env();
  if ( GH_CEASED_LOCAL(*gh, new_gh) )
    {
    /* If local GID_IP_RTM_UP panic'ed, local RTM must go down.    *
       *
     * In the future this may change in the cases, when problems    *
    * in RECEIVING new information were detected on the local slot,*
    * s.t. by bouncing local IP_RTM_UP gate (requesting info from  *
     * remote slots) can cure the problem.                         */
     g_req(GID_IP_RTM, G_REQ_KILL, 0, 0);
    }
  else
    {
     /* correct for rtm_env->gh                                    */
     new_gh &= GH_GET_SLOT_MAP(rtm_env->gh) | GH_GID_MASK;

    sprintf (cbuf, "RTM up self map old %lx, new %lx", *gh, new_gh);
     g_log (IP_DBG_INFO_MSG, cbuf);

    /* Process remote slots going down, as if in ip_rtm_self_map() */
     /* Process any slot that comes up: local or remote            */

      if ( GH_CEASED_REMOTE(*gh, new_gh) ||
       GH_BECAME_PRESENT(*gh, new_gh) )
     {

       ip_send_map_local_msg (rtm_env, gh,
                                                *gh, new_gh,
RTM_SELF_MAP_MSG);
     }
    }

  rtm_env->up_gh = GH_GET_SLOT_MAP(new_gh) | GID_IP_RTM;
  *gh = new_gh;
}
```

BAD CODING PRACTICE ALERT!!! The log message should be defined in an EDL file and not dynamically produced within the code. This wastes CPU and log space, and the string is not a candidate for compression in the image file.

## Perform All Processing in the Thread Approach

A second approach is to perform all necessary processing in the mapping thread.

The advantages are:

- Allows concurrent processing of the base gate context and up/down processing, avoiding a bottleneck in the base gate.
- Does not require any messaging to the main gate context.
- Up/down processing occurs in a timely manner.

The disadvantages are:

- If the mapping pends, this mapping's execution can overlap with activations of the base gate and other mappings of the gate.
- This usually requires access/modification of data structures by multiple threads. This can be dangerous if not done by experience hands and/or if the code is not documented well.
- Long lived up/down processing that has to give up the CPU delays the reception of further mapping activations. The gate handle can get stale.

BGP uses this approach. The (somewhat abbreviated) activation routine:

void

```
bgp_map_bgp_quick(gh, new_gh)
     GH*gh;
     GHnew_gh;
{
BGP_ENV*bgp_env;        /* BGP gate's environment */
BGP_CONN_GATE_MAP *bgp_conn_gate_map;
BGP_CONN_ENV      *conn_env;
bgp_env = (BGP_ENV *) g_env ();
/* see if we're dying */
if (GH_CEASED_LOCAL (*gh, new_gh) )
{
/* set our mib state to not-present and log our goodbye */
bgp_env->wfBgp_inst->wfBgpState = BGP_NOTPRESENT;
if (BGP_LEVEL_LOG(bgp_env, INFO_MSG))
g_log (BGP_TERM_MSG);
/* some counter zeroing edited-out here... */
} /* end if dying locally */
else
{
/* some soloist stuff edited out here... */
/* check for remote instances dying - we have to clean up its routes */
if (GH_CEASED_REMOTE(*gh, new_gh) && BGP_LEVEL_LOG(bgp_env,
WARNING_MSG))
{
bgp_slot_down (GH_CEASED_REMOTE(*gh, new_gh));
} /* end if died remotely */
/* check for new instances - we have to send them update messages */
if (GH_BECAME_REMOTE(*gh, new_gh))
{
bgp_slots_up (bgp_env, GH_BECAME_REMOTE(*gh, new_gh) );
} /* end if remote instance came alive */
} /* end else didn't die locally */
/* save the gate handle */
*gh = new_gh;
} /* end bgp_map_bgp_quick */
```

BAD CODING PRACTICE ALERT: The values for GH_CEASED_REMOTE and GH_BECAME_REMOTE should be cached in stack variables. (For various reasons, mostly due to access of hardware registers, the compiler doesn't optimize this).

## Activation Routines for Dynamic Gates.

Mappings of dynamic gates are different from well-known gates in two important respects:

- The mapped gate is only instantiated on one slot. In most cases, this is the local slot.
- When the mapped gate dies, the mapping owner un-maps the gate. THIS IS CRITICAL, as the GID will be recycled.

Always unmap a ceased dynamic gate in the mapping routine itself, even if a signal or a message is sent to the owner gate. The reasons are:

- Not unmapping causes an annoying "mapping survived" message to appear in the log. This wastes precious log space.
- Not unmapping means that the base gate context must do the unmap. This is not as intuitive as doing it in the mapping routine, and experience has shown that people often forget about the unmapping or some conditional code gets added later that skips the unmapping. A lingering mapped dynamic gate is not a fun thing to debug.

When unmapping a ceased dynamic gate in the mapping routine, always use the "new_gh" argument to derive the gate id and the "gh" argument for the memory pointer:

```
g_map(GH_GET_GID(new_gh), gh, G_UNMAP);
```

and not the a gid or gh derived from the gate's environment or a casted "*gh" pointer. The reason is that GAME guarantees that "new_gh" and "gh" are valid. "*gh" and the environment could have been changed by another gate or thread or could even be referencing memory not owned.

Often, the only reason for mapping a dynamic gate is to restart that gate if it dies. This is the case when an IP NWIF gate maps its forwarding cache gate, as shown here. Note that there will be no return from the G_UNMAP call!

void

```
ip_nwif_cache_map(gh, new_gh)
GH *gh;
GH new_gh;
{
GID gid;
GID old_gid;
NWIF_ENV *nwif_env;
if (GH_BECAME_LOCAL (*gh, new_gh) )
{
*gh = new_gh;
}
else if (GH_CEASED_LOCAL (*gh, new_gh) )
{
*gh = new_gh;
nwif_env = (NWIF_ENV *) g_env ();
nwif_env->fft = NIL (TBL);  /* No more FFT */
nwif_env->cfg_rec->wfIpInterfaceCacheNetworks = 0;
old_gid = GH_GET_GID (new_gh);
gid = g_req (G_REQ_NEW_GID, ip_nwif_rt_cache_init, nwif_env,
G_SIG_INI);
g_map (gid, &nwif_env->cache_gh, ip_nwif_cache_map);
/* un-map the mapping that got us here ! */
g_map (old_gid, &nwif_env->cache_gh, G_UNMAP);
}
} /* end ip_nwif_cache_map */
```

A non-obvious characteristic of this mapping routine: When the g_map() call is made to map the new gid, the current mapping gate context pends and another is immediately scheduled to process the first activation for the new gid.

BAD CODING PRACTICE ALERT: As mentioned previously, the G_UNMAP call should use the "gh" parameter instead of the address of the gate handle in the environment. Using "gh" would run faster, too!

BAD CODING PRACTICE ALERT: This one isn't so bad! There's no need for the GH_BECAME_LOCAL check. *gh can be set unconditionally at the end of the routine.

Other dynamic gate mappings are more complicated, and the developer often has to make the choice of sending a data signal to the base context or deal with the multi-thread situation. The following snippet is BGP's mapping of one of its peer gates (highly edited again). Note the use of a non-so-ensign ensign gate (i.e., it has a real, but unused, activation routine) to indicate that cleanup is happening for this peer. This prevents the base context from re-starting this peer until the cleanup is finished.

void

```
bgp_map_peer(gh, new_gh)
GH    *gh;
GH    new_gh;
{
BGP_ENV*bgp_env;          /* BGP gate's environment */
BGPN_PEER*bgpn_peer;      /* BGP peer structure */
BGP_CLEANUP*bgp_cleanup;/* cleanup block */
BGP_CLEANUP*prev_cleanup;/* previous cleanup block */
BGP_CONN_WAIT*bgp_conn_wait;/* connection wait structure */
```

```
GID  conn_gid;          /* gate id of connection gate */
BGP_NET_INFO*net_info;
char                    message [128];
/* see if the connection gate died */
if (GH_CEASED_PRESENT (*gh, new_gh) )
{
bgp_env= (BGP_ENV *) g_env ();
if(! (GH_IS_LOCAL(bgp_env->self_gh) ))
{
/* bgp main gate has just died, unmap this mapping */
g_map (GH_GET_GID (new_gh), gh, G_UNMAP);  /* does not return */
}
bgpn_peer= RECV_GH_2_BGPN_PEER(gh);
/*
*  create an "ensign" gate and structure that indicates
*  we're busy cleaning this up.  attach it to the bgp env
*/
bgp_cleanup = g_malloc (sizeof (BGP_CLEANUP) );
bgp_cleanup->peer_key = *BGP_PEER_KEY(bgpn_peer);
bgp_cleanup->ensign_gid = g_req (G_REQ_NEW_GID, bgp_dummy_act,
0, G_NO_SIG_INI);
bgp_cleanup->next = bgp_env->bgpn_cleanup;
gp_env->bgpn_cleanup = bgp_cleanup;

bgpn_peer->flags |= PEER_IS_DOWN;
if (bgpn_peer->flags  &  IBGP_PEER)
{
--bgp_env->ibgp_peers;
}
else
{
--bgp_env->ebgp_peers;
}
if (BGP_DBG_EVENT_LOG(bgp_env, BGP_REMOVE_PEER_MAP))
g_log(BGP_REMOVE_PEER_MAP, "Down",
IP_PRINT_ADDRESS (BGP_PEER_LOCAL_IP(bgpn_peer)),
P_PRINT_ADDRESS (BGP_PEER_REMOTE_IP(bgpn_peer)),*gh );
/* prevent RIB processing by other mappings   */
BGP_GET_SEMA (bgp_env->bgp_semaphore);
/* LOTS of cleanup code edited out here */
conn_gid = GH_GET_GID (bgpn_peer->peer_recv_gh);
/* clean up our "cleanup" block and kill the assocated gate */
for (prev_cleanup = (BGP_CLEANUP *) (&bgp_env->bgpn_cleanup);
prev_cleanup->next  !=  bgp_cleanup;
prev_cleanup = prev_cleanup->next)
; /* do nothing */
prev_cleanup->next = bgp_cleanup->next;
g_req (bgp_cleanup->ensign_gid, G_REQ_KILL, 0,G_NO_SIG_INI);
g_mfree (bgp_cleanup);
if((bgpn_peer->flags & PEER_IS_DOWN) &&
queue_isempty(&bgpn_peer->path_queue))
{
/* no use for it - remove from the table */
if (u_delete (bgp_env->bgp_peers, (OCTET *)BGP_PEER_KEY(bgpn_peer),
(OCTET *)BGP_PEER_KEY(bgpn_peer)) )
{
g_log (BGP_DEBUG_MSG, "u_delete failed");
CRASH_BGP;
}
}
/* free the semaphore (after the KILL for CR16894) */
BGP_FREE_SEMA (bgp_env->bgp_semaphore);
/* unmap this mapping */
g_map (conn_gid, gh, G_UNMAP);/* does not return */
} /* end if not present */
/* save the gate handle */
*gh = new_gh;
} /* end bgp_map_peer */
.
```

BAD CODING PRACTICE ALERT: The gate id for the G_UNMAP should be derived from "new_gh"

BAD CODING PRACTICE ALERT: It's a good idea to zero out pointers after their referenced memory had been freed. It catches the bugs a lot faster. In this case, bgp_cleanup should be set zero after the g_mfree().

Also: there is really no reason for the "self_gh" check. GAME will not schedule the owner gate of a mapping if that owner gate has been killed.

## Activation Routines for Ensign/Davidian Gates

An ensign/davidian mapping is somewhat like a dynamic gate mapping for the two reasons listed in the previous section. It differs in that the mapping activation routine can only examine the FLAG bit (31) of the gate handle to determine if the gate is "up" (allocated) or not. You usually see this type of mapping in association with a MIB resource.

This example is the BGP gate's mapping of the davidian gate representing the wfBgpPeerEntry object (when this mapping triggers down, it means a new row of this table exists). When this code was written, the MIB still used ensigns (davidians hadn't been invented yet), so the comments are wrong.

Note that when the existing davidian dies, the MIB is queried for a the davidian representing the new state. The new mapping is set up before this one exits (via G_UNMAP). Again, you don't want to confuse the old and new GIDs!

void

```
bgp_map_wfBgpPeerEntry_obj(gh, new_gh)
GH   *gh;
GH   new_gh;
{
BGP_ENV*bgp_env;          /* ptr to BGP's environment */
OBJ_IDwfBgpPeerEntry_obj_id;/* object id of wfBgpPeerEntry */
INST_IDwfBgpPeerEntry_inst_id;/* instance id of  wfBgpPeerEntry */
/* see if the ensign gate died */
if ( ! (GH_IS_USED (new_gh) ) )
{
bgp_env                   = (BGP_ENV *) g_env ();
if(! (GH_IS_LOCAL(bgp_env->self_gh)))
{
```

```
/* bgp main gate has just died, unmap this mapping */
g_map (GH_GET_GID (new_gh), gh, G_UNMAP);  /* does not return */
}
/* re-bind to the object and re-map */
mib_ascii2obj (BGP_PEER_ENTRY_ASCII_ID,wfBgpPeerEntry_obj_id);
bgp_env->wfBgpPeerEntry_obj_gh =
mib_bind_obj (wfBgpPeerEntry_obj_id, PRIMARY);
g_map (bgp_env->wfBgpPeerEntry_obj_gh,
&(bgp_env->wfBgpPeerEntry_obj_gh), bgp_map_wfBgpPeerEntry_obj);
/* process all of the existing instances */
while (mib_get_new_inst (wfBgpPeerEntry_obj_id,
wfBgpPeerEntry_inst_id) )
{
bgp_new_wfBgpPeerEntry_inst (bgp_env, NIL(BGP_CONN_GATE_MAP),
wfBgpPeerEntry_obj_id,
wfBgpPeerEntry_inst_id);
}
/* un-map this mapping */
g_map (GH_GET_GID (new_gh), gh, G_UNMAP);
} /* end if ensign gate died */
else
{
*gh = new_gh;
} /* end else ensign gate didn't die */
} /* end bgp_map_wfBgpPeerEntry_obj */
```

## Changing a Mapping Activation Routine.

It's not very common, but you can change the activation routine of an existing mapping. This is done by calling g_map() with the new routine:

```
g_map (some_gid, &(env->some_gh), new_map_act);
```

BGP connection gates do this. They first map the well known TCP gate with no action routine. If the configuration information is valid, the gate changes the mapping to use an activation routine (whether the connection is going active or not).

```
g_map (GID_TCP, &(bgp_conn_env->tcp_gh), G_NOACT);
:
:
:
if (bgp_wfBgpPeerEntry_validate (bgp_conn_env)  ==  FALSE)
{
/* remain disabled.... */
} /* end if validation failed */
/* if the entry is disabled or TCP is not active, we just hang out */
else if ( (bgp_conn_env->wfBgpPeerEntry_inst->wfBgpPeerDisable ==
BGP_PEER_DISABLED)  ||
( !(GH_IS_LOCAL(bgp_conn_env->tcp_gh)) &&
!bgp_conn_env->bgp_env->bgp_soloist) ||
(bgp_conn_env->bgp_env->bgp_soloist &&
!(GH_IS_PRESENT(bgp_conn_env->tcp_gh))) )
{
:
:
/* use a real mapping routine for the TCP gate */
g_map (GID_TCP, &(bgp_conn_env->tcp_gh), bgp_conn_map_tcp);
:
:
}
else
{
:
:
/* use a real mapping routine for the TCP gate */
g_map (GID_TCP, &(bgp_conn_env->tcp_gh), bgp_conn_map_tcp);
:
:
}
```

## Soloist Mapping By the Parent

The gate that creates a soloist gate must map the soloist and perform specific tasks if the soloist appears on multiple slots or if all instances of the gate die.

Specifically, before making the g_req() call to create a soloist, the mapping should be done to see if the soloist exists on another slot. If so, no attempt should be made to create the soloist. (Actually, even if you do make an attempt, the gate won't get created because it already exists.)

If the mapping routine ever detects more than one slot bit set in the soloist's gate handle, and if one of those bits is the local slot, the soloist should be terminated if the local slot bit is not the lowest slot bit (leftmost) in the gate handle.

If a mapping activation occurs where the new gate handle has no slot bits set, it is time to start a new soloist election. The mapping routine needs to make the g_req() call to start the new soloist. All mappings may be doing this at the same time. The soloist election procedure will pick only one slot to create the gate.

Here's an example (again from BGP) of how to do a soloist mapping from the parent gate. Note that the "real" mapping activation routine sends a message to the base RTM context. This is the routine that runs in the base context. Also note that BGP can run in both soloist and replicant mode (as indicated by rtm_env->bgp_soloist).

void

```
ip_rtm_map_chg_bgp ( gh, old_gh, new_gh )
GH   *gh;
GH   old_gh;
GH   new_gh;
{
RTM_ENV*rtm_env;          /* RTM's environment */
GH                        temp;
char                      d_str[160];
rtm_env = (RTM_ENV *) g_env ();
if ( ( GH_IS_LOCAL ( new_gh ) )  && ( rtm_env->bgp_soloist ) )
{
/* BGP soloist gate exists on local slot */
temp = new_gh & ~(G_MY_SLOT_MASK);/* Strip off my slot bits */
if ( G_MY_SLOT_MASK < ( temp & GH_SLOT_MAP_MASK ) )
{
```

```
/* soloist exists on another slot with a lower slot number,
* kill local soloist
*/
sprintf ( d_str, "killing local BGP soloist 0x%08", new_gh );
g_log ( IP_DBG_INFO_MSG, d_str );
g_req ( GID_BGP, G_REQ_KILL , 0 , 0 );
return;
}
}
if ( GH_CEASED_LOCAL ( old_gh, new_gh ) )
{
/* BGP gate died locally, remove the locally authored routes */
ip_rtm_remove_bgp_routes(rtm_env);
}
if ( !( GH_IS_LOCAL ( new_gh ) ) )
{
/* BGP gate is not present on this slot */
if ( GH_IS_LOCAL ( rtm_env->bgp_load_gh )  )
{
ip_rtm_start_bgp ( rtm_env );
}
else if GH_IS_LOCAL ( rtm_env->bgprs_load_gh )
{
/* BGP route server code is loaded on local slot */
g_req (GID_BGP, bgprs_init, rtm_env, G_SIG_INI);
}
}
} /* end ip_rtm_map_chg_bgp */
```

Just one comment here: the "temp" variable isn't really necessary. There is no reason to remove the local slot bit to do the comparison.

In case you are curious: The routine ip_rtm_start_bgp() checks to see if BGP is running in soloist mode. If so, and if it's running on another slot, it is not started.

## A General Warning About Mappings

Change the base gate's env before calling g_map(), if you use g_env() in the mapping. Otherwise, a mapping that fires before the g_req() that changes the environment will get the wrong environment pointer.

# Chapter 4 Buffers

## What Are Buffers Used For?

Buffers can be used for communication between gates on the same slot or different slots. For cross-slot communication, this is the only choice (besides the limited information that a mapping conveys).

## Fast Facts About Buffers

Memory on a slot is divided between what is called "local memory" and "global memory". Local memory is used for code, stacks, allocated memory, etc. Global memory is used exclusively for buffers.

On most platforms (including FRE1, FRE2, ASN), local and global memory is carved out from a common DRAM pool. The amounts to use are based on configuration parameters. Once carved, however, only the CPU can access local memory. The memory decoding scheme employed by the backbone and link module interfaces only allows access to global memory.

Excluding ANs and ASNs without SRAM installed, portions of each buffer (one cache line for the buffer header and four cache lines where the link and network headers would usually reside) are mapped to fast SRAM memory. Access to this memory is faster than a non-(processor-)cached DRAM access, but slower than a (processor-)cached DRAM access. This accounts for a major portion of the box's forwarding performance.To avoid cache coherency problems, none of the buffer memory is cachable by the processor.

On an ARE and FRE3, local and global memory are physically separate. The DRAM is used exclusively for local memory. Global memory is managed by a Virtual Buffer Memory (VBM) system. With VBM, physical memory is not statically assigned to buffers as on the FRE. VBM uses a separate physical memory (between 1 and 7 MB) which is mapped as needed to manage up to 32MB of virtual buffer space. The physical memory is organized into 256 byte pages and is assigned upon a "write" operation into a buffer.

## Global Facts

- Each buffer on a slot has the same maximum size (5K on a FRE, up to 10K on an ARE).
- Buffer memory is separate from the memory free pool.
- GAME maintains a single free buffer pool.
- Service is FCFS.
- A single gate can "own" an unlimited amount of buffers, to the point where it can exhaust the buffer pool.

## Facts That Apply Only to the FRE1/2, ASN, AN

- Each slot has a fixed number of buffers.
- Each buffer on a slot is exactly the same size (usually 5K).
- The last cache line of a buffer is set to "no access" if tags are supported.

## Facts That Apply Only to the ARE, FRE3

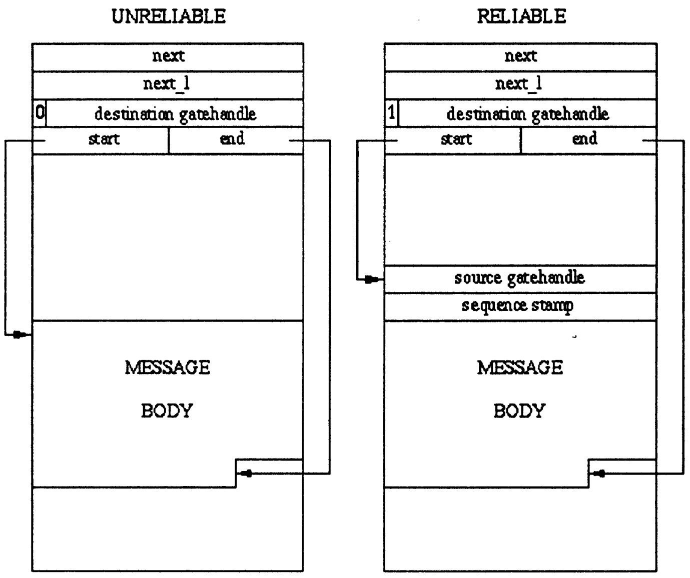
- The free buffer pool is maintained by hardware, but the GAME buffer primitives still work.
- There are a finite number of virtual buffers. However, availability may also be constrained by lack of physical pages.

- Reading unmapped VBM virtual space causes a fatal error. Note that this isn't a problem on the FRE, since the physical memory is always there. It's usually still a bug, though, since uninitialized data is being read.
- The difference between access times of a cached DRAM access and a buffer access on an ARE is much greater than on a FRE1/2

## Buffer Format

Buffer format is illustrated in Figure 4-1.

Figure 4-1. Buffer Format



**Format:**

typedef struct BUF

```
{
 struct BUF *next;   /* next buffer on the list          */
 struct BUF *next_l; /* next list's head buffer          */
 u_int32  dest_gh;   /* destination gate handle          */
 u_int16    start;   /* start offset                     */
 u_int16      end;   /* end offset                       */
} BUF;
```

**Values:**

"next" is a pointer to the next buffer on the list. NIL pointer indicates the end of list.

"next_l" is a pointer to the head of the next list in a transient buffer pool (later).

"dest_gh" is the gate handle of the destination gate. The FLAG bit (31) is set if the buffer transmission is reliable and cleared otherwise.

"start" is the byte offset, relative to the start of the buffer header, to the beginning of the floating message body. Buffers sent over the backplane must have a start offset less than 256. If reliable transmission is used, a 4-byte source gate handle and a 4-byte sequence stamp precede the message body while the buffer is in transit. Even though the start offset is adjusted before the buffer is delivered to the gate, the source gate handle can still be retrieved via the g_src() call.

"end" is the byte offset of the first byte after the message body, relative to the start of the buffer header. Another way to think about it is "end" is start plus length.

**Considerations:**

There are macros that are used to access these fields in a buffer (include/ buffer.h):

```
#define G_BUF_NEXT( buf )         (((BUF *) (buf))->next)
#define G_BUF_NEXT_L( buf )       (((BUF *) (buf))->next_l)
#define G_BUF_DEST_GH( buf )      (((BUF *) (buf))->dest_gh)
#define G_BUF_START( buf )        (((BUF *) (buf))->start)
#define G_BUF_END( buf )          (((BUF *) (buf))->end)
```

Normally, the message body should begin after nominal headroom space (G_BUF_START_PKT or G_BUF_START_MSG) in order to maximize use of special hardware accelerators that may be available on some versions of hardware. Specifically, on the FRE, this allows the link and network layer headers to reside in SRAM.

## Buffer Pools / Lists

Buffers change ownership very often and it is paramount to minimize the system overhead required to track them. The scheme is based on three buffer pools:

- A free buffer pool.
- A transient buffer pool containing buffers owned by a gate only temporarily.
- A set of private buffer pools containing buffers owned by a gate for a greater length of time (over multiple activations).

## The Free Buffer Pool

Buffers that are not owned by any gate are maintained in the free buffer pool. On non-VBM systems, this is a simple linked list of buffers, connected by the "next" pointers. On VBM systems, the free buffer pool is maintained by VBM hardware. The same GAME calls, such a g_balloc() and g_breplen(), work on both systems.

## The Transient Buffer Pool

A transient buffer pool only exists for gates that are in the active or pended states. That is, a gate only has a transient pool if it has been activated for buffer delivery, signal delivery, or a mapping (each mapping context has its own transient pool). When a gate exits an activation, it must have an empty transient pool. Otherwise, it is said to "orphan" buffers. The punishment in this case is mild (a message is logged and the buffers freed), but this is an indication of an error in the application - the buffer was meant to go "somewhere". As discussed in the Mapping section, the punishment for leaving buffers on the transient queue after exiting a mapping activation is more severe (the gate is terminated).

Buffers can be placed into the transient buffer pool for three reasons:

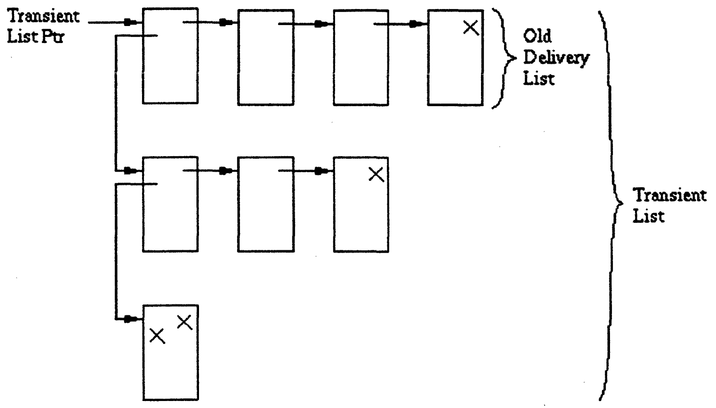
- Buffers can be sent to a gate using unreliable (g_xmt, g_xmt_im, g_fedex, g_fedex_clean) or reliable (g_fwd, g_fwd_list, g_rpc) GAME buffer transport functions. GAME puts these buffers onto a gate's "delivery" list (managed by "head" and "tail" in the GATE structure). This is a linked list, using the "next" field in the buffer header. When a gate is activated for buffer delivery, the delivery list is transferred to the transient pool. This simple linked list comprises the entire transient pool and its head is passed in the "buffers" parameter of the gate's activation routine.

- Any GAME buffer allocation primitive (g_balloc, g_breplen, g_copy) will create a separate list of buffers in the transient pool. Note that a single buffer can constitute an entire list if the primitive only returns one buffer (g_balloc). The relationship between this list and the activation list is covered just ahead.
- The g_rpc() call can also return a list of buffers which are placed in the transient pool.

Buffers can be removed from the transient pool via any of the following methods:

- A bounded list of buffers can be explicitly freed via g_bfree().
- An entire list of buffers can be delivered to other gates via g_xmt() or g_xmt_im(). If all of the buffers are going to the same gate, g_fedex() or g_fedex_clean() can be used.
- A single buffer can be reliably delivered to one or more instances of a gate via g_fwd(). If a reply is needed, g_rpc() can be used.
- g_fwd_list() reliably delivers a list of buffers to the same destination.
- GAME will return all of the transient pool buffers to the free pool should a gate die in the active or pended states.

Figure 4-2. Transient Pool



The transient pool is managed as a linked list of linked lists. The "next" pointer is used to form the independent linked lists. In the head buffer of each list, the "next_l" pointer is used to link the lists together.

The order of the list of lists is "most recently acquired". For example, suppose a gate is activated with 10 buffers on its activation list. The transient pool pointer points to the head buffer on that list. That buffer's "next_l" is NIL and "next" points to the 2nd buffer in the list. The gate then does a g_balloc(). Now, the transient pool pointer points to the newly allocated buffer. That buffer's "next" pointer is NIL, and its "next_l" points to the head of the activation list.

GAME functions that remove buffers from the transient pool maintain the integrity of the transient pool. To continue the above example, suppose the gate now frees the first buffer on the activation list. Here's what happens: The "next" pointer of the buffer being freed is used to find the next (2nd) buffer in that list, which becomes the new head of the list. The "next_l" pointer from the freed buffer is written into the new head's "next_l" pointer (which, in this case, is NIL). Finally, the other list head buffer (the one acquired via g_balloc) is modified so that its "next_l" pointer points to the the new head of the initial list.

IMPORTANT: A gate must never directly modify the "next" or "next_l" pointers in a buffer. Only GAME functions can do this. Otherwise, the transient pool may become corrupted. g_bmove() can be used to re-arrange the order of buffers within the transient pool.

During an activation of a gate, lists in the transient pool are usually traversed via one of the following methods:

- Each buffer is processed and modified but remains linked in its place in the transient pool. This is a normal case for the data path forwarding code and results in the best performance, assuming that the gate does not pend. The list of buffers, either delivered or allocated, is batch-processed first and then wholesale-shipped to other gates using g_xmt(), g_xmt_im(), or g_fedex[_clean]().
- Each buffer is reliably transmitted elsewhere (an involved process during which the ownership of the buffer may change several times) via g_fwd() or g_rpc(). This process begins with the buffer's removal from the transient pool. At this point, the gate CANNOT reference that buffer any more (this is true for all of the buffer transport functions). A gate must obtain the next buffer pointer (G_BUF_NEXT) before submitting the prior buffer for transmission.

- g_repeat() removes the current head buffer from the the list, puts it in its own list, and spoon-feeds it into an application-supplied routine. THIS METHOD IS HIGHLY DISCOURAGED!! It is much more effiencient for a gate to walk the buffer list itself (method 2).

Finally, the transient buffer pool structure is internal to GAME and must not be manipulated by an application. It is explained in some detail here so that application writers understand the underlying structure and also as an aid to debugging.

Application writers must ignore the "next_l" chaining aspects of the transient pool in their code and simply deal with the independent buffer lists. The GAME system calls will maintain the appropriate chaining on behalf on an application. This implies that applications should only walk buffers lists via the "next" pointer. The following are the GOLDEN RULES of buffer usage:

- Never write a "next" pointer.
- Never read or write a "next_l" pointer.

## Private Buffer Pools

As mentioned previously, a gate cannot have any buffers in its transient pool when it exits an activation. However, there are some cases where a gate must take ownership of buffers over multiple activations. For example, a device driver must hold on to buffers that are assigned to the driver rings (either waiting for transmission or available for receiving incoming frames). For this reason, GAME provides private buffer pools to each gate.

Each gate has, by default, two private pools (designed for the driver gates). Additional private pools can be allocated. Buffers can be transferred from the transient pool to either private pool and vice versa. No other buffer manipulation can occur when a buffer is on a private pool! For example, you cannot g_bfree() a buffer unless you first move it onto the transient pool. You cannot move buffers directly between private pools.

## Single Linked Private Pool

The private pools are usually organized as simple linked lists. (Private pool #1 can be organized into a doubly-linked list. See "Doubly-Linked Private Pool later in this chapter.) Buffers moved from the transient pool to a private pool are put at the end of the list. Buffers can be retrieved back into the transient pool from anywhere in the private pool list (the head and tail of the desired private pool buffers are specified). Each retrieval creates a new list in the transient pool.

The functions that manipulate the default private pools are:

- g_bsave (head, tail)save a list of buffers from the transient pool to the end of private pool 1
- g_bsave2 (head, tail)save a list of buffers from the transient pool to the end of private pool 2
- g_brestore (head, tail) restore a list of buffers from private pool 1 to the transient pool
- g_brestore2 (head, tail)restore a list of buffers from private pool 2 to the transient pool
- g_bhead () returns the head of private pool 1
- g_btail () returns the tail of private pool 1
- g_bhead2 ()returns the head of private pool 2
- g_btail2 ()returns the tail of private pool 2

The g_bheadX() functions do not remove the head buffer from the private pool. Ditto for g_btailX(). They just return the address of the application buffer.

The following functions allocate and manipulate additional private pools. Up to 32 pools (an arbitrary maximum) can be allocated.

- g_npools (num) - allocate "num" private pools. "num" indicates the _total_ number of private pools needed, not the increment beyond the first two. "num" must be greater than 2.
- g_bsaven (n, head, tail) - save a list of buffers from the transient pool to the end of private pool "n"
- g_brestoren (n, head, tail) - restore a list of buffers from private pool "n" to the transient pool
- g_bheadn (n) - returns the head of private pool "n"
- g_btailn (n) - returns the tail of private pool "n"

A gate can only call g_npools() once, so it must determine the maximum number of pools it needs for its entire life before making the call. A gate can call g_npools() even after it has saved buffers on pools 1 and 2 (this was a error condition once upon a time), but this is not recommended.

After calling g_npools(), the pools numbered 1 and 2 are the first two private pools, usually accessed by g_bsave() and g_bsave2(). The following function calls are equivalent (but only after calling g_npools!):

```
g_bsave(...)        ==  g_bsaven (1, ...)

g_bsave2(...)       ==  g_bsaven (2, ...)

g_brestore(...)     ==  g_brestoren (1, ...)

g_brestore2(...)    ==  g_brestoren (2, ...)

g_bhead()           ==  g_bheadn (1)

g_bhead2()          ==  g_bheadn (2)

g_btail()           ==  g_btailn (1)

g_btail2()          ==  g_btailn (2)
```

### Doubly Linked Private Pool

Private pool #1 can be organized into a list that is doubly-linked instead of a single-linked list. This was done to support the Tsunami ATM driver. This driver uses private pool 1 to save buffers on the driver receive ring. Unlike other drivers, data reception can complete out-of-order in respect to the buffer list. Because of this, a method was needed to remove buffers from the free pool without requiring a walk of the list (performance!). So, the following two calls were invented:

- g_bsave_dbl (head, tail) - save a list of buffers from the transient pool to the end of private pool 1, doubly-linked
- g_brestore_dbl (head, tail) - restore a list of doubly-linked buffers from private pool 1 to the transient pool

The back-link of buffers on the pool is done using the "next_l" pointer. This is hidden within the function call code, however. The caller MUST NOT reference the "next_l" pointer of the buffers for any reason.

NOTE: Manipulations of private pool 1 must be exclusively single-linked or double-linked. If g_bsave_dbl() is used, g_brestore_dbl() is the only other call that can be used to manipulate private pool 1 (g_bsave, g_brestore, g_bsaven, and g_brestoren CANNOT be used)

The Tsunami driver is the only user of this feature.

## Buffer Allocation

A gate can allocate buffers via the g_balloc(), g_breplen(), and g_copy() function calls.

### g_balloc() Function Call

**Call:**

```
BUF *g_balloc (u_int32 tmo)
```

**Values:**

"tmo" is the amount of time to wait for a buffer, if none are available. The units are roughly milliseconds (1/1024). The actual time used for timer expiration is not necessarily what was entered and usually is longer. The FRE1, FRE2, ASN, ACE25, ACE32, AFN, and ARE round this time up to multiples of 16 ms. The AN and the ARN round this time up to multiples of 64ms. These macros are available for use in setting "tmo":

- G_TMO_SECONDS (sec) yields a value representing "sec" seconds
- G_TMO_DEFAULT yields 1/2 second
- G_NO_WAIT yields a zero

If "tmo" is set to G_NO_WAIT, g_balloc() will not wait for a buffer when none are available.

**Return Value:**

The return value is a pointer to a single buffer on its own list in the transient pool. NIL(BUF) is returned if no buffer could be allocated. YOU MUST CHECK FOR THIS CONDITION AFTER ALL CALLS TO g_balloc()!

The caller cannot assume anything about the contents of the returned buffer or its start and end offsets.

## g_breplen Function Call

**Call:**

```
u_int32 g_breplen (u_int32 num, BUF **head, BUF **tail)
```

**Values:**

"num" is the number of buffers desired.

"head" is a a pointer to a location where the head pointer of the returned buffer list can be written.

"tail" is a pointer to a location where the tail pointer of the returned buffer list can be written.

**Return Value:**

The return value is the number of buffers actually allocated. Unlike g_balloc(), there is no way to wait for additional buffers if less than "num" are available. If the return value is not zero, the list of buffers resides on it's own list in the transient pool.

## g_copy() Function Call

**Call:**

```
BUF *g_copy (BUF *buf)
```

**Values:**

"buf" is a pointer to a buffer to be copied. This buffer must reside in the caller's transient pool or in a private pool.

**Return Values:**

The return value is a pointer to a single buffer on its own list in the transient pool. The start and end offsets match those in "buf", and the contents of the message body (between the offsets) matches "buf". Nothing else from the buffer is copied (specifically, "next", "next_l", and the gate handle are NOT copied).

NIL(BUF) is returned if no buffer could be allocated. YOU MUST CHCK FOR THIS CONDITION AFTER ALL CALLS TO g_copy()!

g_copy() provides no provision for waiting for a buffer.

Always check for a NIL return value from g_balloc() and g_copy()! This is a common mistake. Code that does not check for NIL has been released - and it crashes due to an invalid memory reference when the buffer supply is low!

## Buffer Manipulation

Besides those discussed previously for the buffer headers, there are some other useful macros and functions for buffer manipulation (include/ buffer.h).

### Macros

```
(type *) G_BUF_INI        (BUF *buf, type)
(type *) G_BUF_PDU        (BUF *buf, type)
(char *) G_BUF_PDU_START  (BUF *buf)
(char *) G_BUF_PDU_END    (BUF *buf)
(int)    G_BUF_PDU_SIZE   (BUF *buf)
```

"buf" is the pointer to the buffer.

"type" is the C data type that will be held in the buffer.

G_BUF_INI is used by a gate that creates a message in a buffer. The macro sets the start offset of "buf" to G_BUF_START_MSG, sets the end according to the structure size, and returns a casted pointer to the structure within the buffer.

G_BUF_PDU is used by a gate that is reading a buffer that contains a message. The macro can only act on a previously initialized buffer. It returns a casted pointer to the structure within the buffer.

G_BUF_PDU_START also acts on a previously initialized buffer. It returns a simple char pointer to the data within the buffer. This is used when the buffer contains a data stream rather than a structure.

G_BUF_PDU_END returns a simple char pointer to the space following the data within the buffer.

G_BUF_PDU_SIZE returns the number of bytes of data in the buffer, as indicated by the start and end offsets.

### Primitives

The G_BUF_MAX_END and g_blen() primitives can be used to determine the amount of data that can be written into a buffer.

G_BUF_MAX_END returns the maximum G_BUF_END value that can be used for buffers that is guaranteed to work for delivery to any other slot in the machine. Currently this is 2000, except for the AN and ARN (1776).

g_blen() returns the maximum G_BUF_END value that can be used on the local slot. Note that if a buffer is filled to this size, sent to another slot, and that slot has a smaller buffer size, the buffer will not be delivered. g_blen() returns slightly less than 5K on a FRE.

g_blen() returns 5K as the buffer size on an ARE. This is a hack which alows AREs to talk with FREs. ARE buffers are created at 10K ecause of ATM, but most applications can ignore that possibility.

## Freeing Buffers

Buffers can be freed in three different ways:

- A bounded list of buffers in the transient pool can be explicitly freed via g_bfree().
- Buffers given to a reliable or unreliable transmission function will be freed if the buffer's gate handle contains zero.
- GAME will return all of a gate's buffers to the free pool should a gate die in the active or pended states.

### Using g_bfree()

**Call:**

```
void g_bfree (BUF *head, BUF *tail)
```

**Values:**

"head" is a pointer to the first buffer a the list in the transient pool to be freed.

"tail" is a pointer to the last buffer in the list to be freed. This must be on the same list in the transient pool as "head" and must follow "head"in that list.

Note that head can equal tail, freeing exactly one buffer.

Any pointers referencing the free buffers should be modified. If no more buffers exist on the original list, the pointer(s) should be set to zero. If buffers exist after tail, the pointer to the buffer after tail must be saved before g_bfree() is called.

## Gate Handle Contains Zero

**Call:**

```
G_BUF_DEST_GH (buf) = 0;
```

**Considerations:**

The only case where this makes sense is when a gate processes an entire list of buffers without pending. In this case, the gate sets the gate handles to real values or zero and uses g_xmt() or g_xmt_im() to deliver the list.

## Dying Gate

GAME will return all of a gate's buffers to the free pool should a gate die in the active or pended states. Killing a gate is the most drastic way to free its buffers.

## Moving Buffers Around (g_bmove)

To move one or more buffers into a specific location within a transient pool list, use g_bmove().

**Call:**

```
void g_bmove (BUF *ins, BUF *head, BUF *tail)
```

**Values:**

"ins" is a pointer to a buffer in a transient pool list that serves as the insertion point. The buffers are inserted after this buffer.

"head" is a pointer to the first of a list of buffers to be moved.

"tail" is a pointer to the last of a list of buffers to be moved.

"head" and "tail" obviously must belong to the same list within the transient pool. "ins" cannot be "head", "tail" or any buffer in between.

**Considerations:**

GAME will take the list of buffers, remove it from its current place in the transient pool, and splice it into the list that "ins" belongs to, directly after "ins". If "ins" is NIL, head and tail form a new list in the transient pool.

## Removing/Adding Buffers From GAME

This feature is quite dangerous and not something that you will commonly use, unless you do platform development.

It is possible to remove buffers from a gate's transient pool, effectively disconnecting them from GAME completely. Similarly, you can pull into the transient pool buffers that do not belong to GAME. This feature exists because some hardware, such as the ARE ATMizer, needs to take complete control over the buffers it is using.

### Removing Buffers From GAME

**Call:**

```
void g_export_bufs (BUF *buf_list)
```

**Values:**

"buf_list" is a pointer to a list of buffers on the transient pool.

**Considerations:**

GAME will remove the indicated list from the transient pool and leave the buffers in an "unowned" condition. The caller usually delivers the buffers to another piece of hardware.

### Adding Buffers To GAME

**Call:**

```
void g_import_bufs (BUF *buf_list)
```

**Values:**

"buf_list" is a pointer to an unowned list of buffers.

**Considerations:**

GAME will put the owned buffers into the transient pool, creating its own list. The caller usually gets these buffers from another piece of hardware.

## Performance Tips

Across all platforms, accesses to buffer memory is more expensive than accesses to DRAM locations that are cached by the local processor. Therefore, one should always follow the rule "read once, write once" when it comes to data in a buffer.

This includes the buffer header structure BUF, and the macros that reference it. A common bad practice is to continually reference the start and end offsets via the G_BUF_PDU_SIZE macro. Instead, the value should be cached in a stack variable.

A code strip may need to add data to a buffer a little bit at a time. An example would be a protocol like RSVP, which builds a message out of multiple "objects". A bad way to code this would be to set the end offset after adding each object and then reading it again when adding the next object:

```
 /* add object 1 */
 object1 = (OBJECT1 *) G_BUF_PDU_END (buf);
 :
 :
 G_BUF_END (buf) = G_BUF_END (buf) + sizeof (OBJECT1);
 /* add object 2 */
 object2 = (OBJECT2 *) G_BUF_PDU_END (buf);
 :
 :
G_BUF_END (buf) = ( (u_int32) object2) + sizeof (OBJECT2);
 /* etc... */
```

The problem here is that we are constantly reading and writing into memory that is slower than cached DRAM. A better way to code this would be:

```
char *local_buf_end;
:
local_buf_end = (char *) /* end of header */
:
/* add object 1 */
object1 = (OBJECT1 *) local_buf_end;
:
:
local_buf_end = ( (char *) object1) + sizeof (OBJECT1);
/* add object 2 */
```

```
object2 = (OBJECT2 *) local_buf_end;
:
:
local_buf_end = ( (char *) object2) + sizeof (OBJECT2);
/* etc... after all objects are added: */
G_BUF_END (buf) = local_buf_end - ( (char *) buf);
```

This way, we meet the "read once, write once" criteria.

## Debug Tips

### debug krnl Command

The 'debug krnl' command provides a few settings of use for debug buffer problems (note that the "debug" module must be loaded). The use of "debug krnl" is discussed in file:/rte1/harpoon/doc/game/html/game_debug.html.

#### buf_chk

buf_chk verifies buffers are valid, on the same list and owned by the caller. Applicable to g_bfree(), g_bmove(), g_bsave(), g_bsave2(), g_bsaven().

#### buf_pool

buf_pool verifies private pool is valid and the head and tail arguments to the restore syscalls are for buffers actually in the private pool. Applicable to g_brestore(), g_brestore2(), g_brestoren().

#### all_buf

all_buf is the combination of buf_chk(), buf_pool(), buf_size(), buf_xmt(). The latter two settings are discussed in the Inter-Gate Communication section.

## Dumping Buffer Contents

The contents of a buffer can be dumped to the event log via the buf_dump() call:

**Call:**

```
void buf_dump (BUF *buf)
```

**Values:**

"buf" is a pointer to the buffer to be dumped to the event log.

**Considerations:**

The buffer headers and the first 64 bytes of data (beginning at the start offset) are dumped, in hex, to the log. A checksum of the buffer is also done and displayed.

This call is useful for debugging cases where a gate receives a buffer that it doesn't expect.

## Debugging Orphaned Buffers

An orphaned buffer occurs because a GATE completes execution and there are still buffers on its Transient Buffer List. Now that I have given you the pure technical reason, let me explain what it means. When a GATE is scheduled to run because something has sent a buffer list to it, the GATE must do one or more of the following before it completes execution.

- Free the buffer(s)
- Put the buffer(s) on one or more of its private buffer lists
- Send the buffer to some other gate

If it doesn't do one of the above to EVERY buffer it has received before it completes execution, then an orphan buffer error occurs.

Now WHY would this happen? There are several possible reasons.

- The gate received buffers and it was not designed to process them
- The gate received a buffer that it did not recognize
- The gate received a buffer that it recognized, but there is a bug in the code so it lost it.

The first two reasons indicate that there is a bug in the sending gate. The third reason indicates that there is a bug in the receiving gate.

How do you determine which gate is the problem? This involves experience with the operation of the router and knowledge of the code involved. I'll use your buffer as an example.

The first thing to do is look at the buffer header. This is always the first piece of information displayed in the log. The format of the buffer header is as follows

```
u_int32 *next      - Pointer to next buffer on list

u_int32 *next_l    - Pointer to next buffer list

u_int32 dest_gh    - Gate handle that buffer is destined for

u_int16 start    ' - Byte offset from beginning of packet
where data starts

u_int16 end          - Byte offset from beginning of packet
where new data would go
```

The buffer dump in the log begins with the address of the buffer and then dumps the buffer header.

```
buf=0x8022e800 - 0x00000000 0x00000000 0x000982af 0x009000cc
0x60000052
```

In this example, the buffer is located at address 0x8022e800. The buffer header values are:

```
next    0x00000000
next_l  0x00000000
dest_gh 0x000982af
start   0x0090
end     0x00cc
```

The destination gate id (not handle) is 0x182af. The leading bit that makes the one a 9 is actually part of the "slot mask" field and indicates that the destination slot is 12. Thats good because that is the slot that is reporting the error. The log messages also indicate that the gate that last had the buffer was 0x182af. This is also good because that is what the dest_gh field told us.

What we have right now is a buffer that was orphaned by the gate that it was purposely sent to. The question is "who is at fault?". Tho determine that, we need to look at what this gate is supposed to do with the buffers. In order to find that out, we need to look up the address of the code that the gate executes when it comes to life.

Where do we find the address of the code? It is in the log. The log message

```
Scheduler exited due to orphaned buffer
Last gate gid = 0x182af @ 0x316a59ae buf=0x8022e800
```

tells us that the buffer at address 0x8022e800 was orphaned by gate 0x182af while running the code at location 0x316a59ae.

So where do we find out what code this is connected with? You have to look at the loadmap or sometimes the code logs a message if it receives a buffer it does not like. This message is

```
ip_ft_proxy_act received an unexpected buffer
```

Knowing this, we can look at the function ip_ft_proxy_act and see how it handles buffers. The code for this function looks like the following:

```
void
ip_ft_proxy_act(env, buf, signal)
   void    *env;
   BUF     *buf;
   SIG     signal;
{

   if (buf) {
      g_log(IP_UNEXPECTED_BUFFER_MSG,"ip_ft_proxy_act");
      buf_dump (buf);
   }
   else {  /* Unexpected signal */

      g_log(IP_UNEXPECTED_SIGNAL_MSG,"ip_ft_proxy_act");
     CRASH(IP_CRASH);
   }

} /*....End of function ip_ft_proxy_act....*/
```

As you can see, this code was never designed to receive buffers at all so it falls under reason 1. This means that the sender of the buffer was at fault. The likely problem is that the sender has a stale gate handle saved away. The gate handle used to be something that the sender sent buffers to but that gate died. The sender however did not update the info about where to send buffers. Eventually, the dead gate handle was reused for something else. That new gate started receiving buffers and orphaned them.

So how do you find out who the sender was? That's the tough part. There are two kinds of buffer movements in the router "Reliable" and "Unreliable". If a buffer is tranmitted as reliable, there will be a header just before the data in the buffer that indicates who sent the buffer. The left most bit of the dest_gh will also be a "1". In this case, the left most bit of dest_gh is a "0" so it is not reliable. At this point, the only thing you can do is draw on experience and see if the data in the buffer means anything to you. To find where the data starts, you take the address of the buffer and add the start field to it in this example, the result is

```
0x8022e890
```

If you go back to the log, you will see that the buffer dump contains this data:

```
0x8022e880 - 0x02608c2d 0x3b90ab00 0x8c2d4166 0x08004500
0x8022e890 - 0x4500003c 0xf4db0000 0xfb013a4d 0x8b33bf1d
0x8022e8a0 - 0x8b33bb13 0x0800ebe4 0xb433011b 0x2f5f0ac7
0x8022e8b0 - 0x00061ca0 0x00000000 0x00000000 0x00000000
0x8022e8c0 - 0x00000000 0x00000000 0x00000000 0xd4f71500
0x8022e8d0 - 0x76be7f00 0x7a3db41d 0xb81a0260 0x8c2d6f0d
0x8022e8e0 - 0xf9516f0d 0xf9516f0d 0xf9516f0d 0xf9516f0d
0x8022e8f0 - 0x00000000 0x00000017 0x00020000 0x0003000a
0x8022e900 - 0x0001400b 0x2f536a1f 0x0000000a 0x00000002
0x8022e910 - 0x00000001 0x00000001 0x90000000 0x10000000
0x8022e920 - 0x1000495d 0xa85c0310 0x0000e2c1 0x61ca0376
0x8022e930 - 0x804fa8b0 0x0004e00b 0x00000000 0x00000000
0x8022e940 - 0x00000000 0x804fa8c0 0x00000000 0x00000000
0x8022e950 - 0x00000000 0xc5efc68e 0x804fa8d0 0xc391449f
0x8022e960 - 0xa2a55b1b 0x9d5c823e 0xc16f10c8 0x804fa8e0
0x8022e970 - 0x0e500000 0x012c0000 0x00010000 0x012c0000
```

The reason the buffer dump starts 16 bytes in front of the data is so that you can see the reliable message header if there is one. If you look at this data, it looks like it could be an IP header since it starts with 0x4500. If you go on that premise and try to decode the rest of the data, you see that it looks like the following.

```
Version - 4
IHL - 5
Type of Service - 00
Total Length - 3c
Id - f4db
Flag/Fragment - 0000
Time to live - fb
Protocol - 01 (ICMP)
Header Checksum - 3ad4
Source Address - 8b33bf1d (139.51.191.29)
Destination Address - 8b33bb13 (139.51.187.19)
```

So far it looks like it is IP. Since it indicates it is an ICMP message, we can decode the rest of the data based on the ICMP header and see what happens.

```
Type - 08 (ICMP Echo Request)
Code - 00
Checksum - ebe4
Identifier - b433
Sequence Number - 011b
Data - The rest of the data until you reach the end offset
```

So this looks like a ping request going from 139.51.191.29 to 139.51.187.19. Since the packet is being sent to the wrong place, there is stale gate handle information on the slot handling the source address (139.51.191.29). If you bounce the interface handling the destination address (139.51.187.19), this may cause IP to invalidate the info on this station on the other slots. If IP does this (and it should), then an ARP should be sent out to the destination and then connectivity will be restored and the orphan errors should stop.

If you take this info back to the router that is having the problem, you can easily verify whether the decode of the packet is correct by seeing if the IP addresses are valid.

Basically, decoding orphan buffers can be difficult. Getting this error ALWAYS means there is a bug. Finding the bug is the problem. It always helps if a reproducible case is available.

One final note: In the recent past, IP has had several changes made to fix stale gate handle problems. You should check with Sustaining Management about the availability of those fixes.

# Chapter 5
# Inter-Gate Communication

## Types of Inter-Gate Communication

There are three types of inter-gate communication in GAME:

1. Buffer delivery: Works locally and across slots.
2. Signals: Works locally only. Can be accompanied by memory transfer (G_SIG_DATA).
3. Mappings: Works locally and across slots. A gate can be killed to indicate an event.

## Buffer Delivery

GAME provides seven functions that deliver buffers to other gates. Four of these are unreliable and three provide reliability through acknowledgment and retry mechanisms.

A few common rules regarding all buffer delivery mechanisms:

- The buffer must have a valid gate handle set via G_BUF_DEST_GH(). This can be a zero if the buffer is to be freed.
- The start and end offsets must be set properly to point to the first byte of data and the byte following the last byte of data, respectively.
- On VBM systems, data must have been written to all memory indicated by the start and end offsets.
- After calling the GAME function, the calling gate no longer owns the buffer and it must not reference it. It's a good idea to zero out buffer pointers once a buffer is delivered in order to surface such bugs early in the testing process.

## Unreliable Buffer Delivery

Unreliable buffer delivery is analogous to a network datagram service. The data will be delivered in a best-effort manner. It will most likely get to where it has to go, but there is no guarantee. This provides a very efficient, low overhead transfer of data. Not surprisingly, it is used to provide datagram forwarding.

In the current BN implementation, unreliable delivery to a gate on the same slot is actually 100% reliable, assuming the destination gate exists. However, there are two reasons to not rely on this:

1. There has long been discussion of implementing a "buffer clipping" mechanism that would remove unreliable buffers from gates' delivery lists when the free buffer pool empties. However, the chance of clipping ever getting implemented is almost nil.
2. On a VBM system, there is the possibility of dropping a unreliable buffer between gates on the same slot. If the sending gate allocates and writes to a buffer, there is a possibility of running out of physical buffer space and acquiring a wastebasket page (more on this later - see g_fedex_clean). When this happens, the buffer is dropped when delivered.

The format of a message in a buffer must be agreed upon by the sender and the receiver (i.e., located in a *.h file). GAME knows nothing about the data contents, other than its size.

### g_xmt() - Unreliable Delivery of a List of Buffers

**Call:**

```
void g_xmt (BUF *buf_list)
```

**Values:**

"buf_list" is the pointer to the head of a list in the transient pool (i.e.,linked by the "next" pointers).

**Considerations:**

Every buffer in the list will be processed by GAME (i.e., until it gets to a NIL "next" pointer). GAME will deliver each buffer according to the gate handle in the buffer. An individual gate handle can indicate zero, one, or multiple gates. If the GH is zero, GAME merely frees the buffer. If one slot bit is set, the buffer goes to exactly one gate instance. If multiple slot bits are set, the buffer goes to the instances on the indicated slots. GAME will silently disregard any request to send a buffer to a gate instance that does not exist.

g_xmt() is intended to be used when the sending gate is transmitting buffers to many different destinations (e.g., L2 or L3 forwarding code). If all of the buffers on the list will always have the same gate handle, g_xmt() can be used, but g_fedex() is much more efficient.

Here is a very edited version of the ip_xmit() function. This gate receives packets from IP applications on the router and transmits them out the appropriate interface. Shown here is the loop and the various places that the gate handle can be set in the packet. Finally, the entire list is delivered via g_xmt()

```
FOR_EACH_BUF (rx_pkt, buflist)
{
     :
          rx_pkt->dest_gh = 0;
     :
          rx_pkt->dest_gh = ((GH_SLOT_MAP_MASK &
              dest_nwif->nwif_map.gh) | GID_IP_XMIT);
     :
          rx_pkt->dest_gh = 0;
     :
          rx_pkt->dest_gh = rtm_env->mp_gh;
     :
          rx_pkt->dest_gh = rtm_env->cache_gh;
     :
ip_xmit_final_considerations(dest_nwif, fwd_entry, dest_gh,
rx_pkt,
    rtm_env);
     :
} /* end for each buf in buflist */

  /* Send it on it's way */

  g_xmt (buflist);
```

The macro G_BUF_DEST_GH (rx_pkt) should be used instead of referencing rx_pkt->dest_gh directly.

## g_fedex() - Efficient Unreliable Delivery to One Destination

**Call:**

```
void g_fedex (GH dest_gh, BUF *head, BUF *tail)
```

**Values:**

"dest_gh" is the destination gate handle for all buffers in the list represented by "head" and "tail".

"head" is the head of a list of buffers in the transient pool.

"tail" is the tail of a list of buffers in the transient pool.

**Considerations:**

Fedex should be used when all of the buffers in a list are going to the same destination gate. The dest_gh parameter must exactly match the GH in every buffer in the list (caveat below). This allows GAME to avoid a list walk and deliver the buffers in the most efficient manner.

Don't call g_fedex() with the FLAG bit set in the GH. It will call g_xmt() to remove the FLAG bit in the buffers, which requires a buffer walk.

Don't call g_fedex() with multiple slot bits set in the GH. Fedex can't deliver local and remote copies, so it punts the multicast scenario entirely.

Don't call g_fedex() with a zero gate handle. It's much more efficient to just call g_bfree().

This example is from the sync (mk50) driver's receive interrupt processing. Buffers have been assigned to the driver receive ring, and now some of them contain received packets. Most of these will be delivered to the DP decaps gate for the circuit (some get processed locally - details left out for brevity).

```
for (pkt=head,tail=NIL(BUF); pkt;) {
    /* if current desc is owned by the MK50, stop! */
    if ((data = rd->addr) & MK5025_OWN) {
        /* AND this is the 1st packet, spurious int, bag
out! */
        if (!tail) {
            head = NIL(BUF);
        }
        break;
    }
    :
        G_BUF_DEST_GH(pkt) = env->decaps_gh & ~ GH_MSG_FLAG;
        /* make tail previous pkt, get next pkt */
        tail=pkt;
        pkt=G_BUF_NEXT(pkt);
    :
}   /* end of for RINT loop */

/* if we have a valid list to forward */
if (head) {

    /* restore buffers to transient pool for delivery */
    g_brestore(head, tail);
    :
        g_fedex(env->decaps_gh & ~ GH_MSG_FLAG, head, tail);
```

**A g_fedex() trick:**

If you have a list of buffers with various gate handles, none of which contain the local slot bit, you can trick g_fedex() into delivering them for you. If the "dest_gh" parameter is set with a single slot bit for another slot, GAME simply tacks the list onto the backbone transmit queue (g_xmt() would do a list walk). The receiving slots parcel out the buffers to the proper gates, and have no clue whether the original sender used g_xmt() or g_fedex() (nor do they care). This can be a big win for forwarding code in a multi-slot box where a slot contains a single interface or a small percentage of the interfaces on the entire router.

An example of this is contained in the IP forwarding code. The destinations GHs of all packets are 'or'ed together. If all of the packets are going to a remote slot, g_fedex() (actually, g_fedex_clean(); see next section) is used.

```
dest_gh = fedex_remote = 0;
:
for (next_buf = start_buf, buf_start = start_buf->start;
    next_buf;
    buf_start = next_start, fedex_remote |= dest_gh) {
:
:   (dest_gh gets set to current buffer's gh)
:
} /*end for...*/

/*
    * If all buffers in the list are destined for a remote
slot or
    * are being freed g_fedex can be called for performance
gain
    */

if (GH_IS_LOCAL (fedex_remote))
    g_xmt (buflist);
else
{
    dest_gh = GH_SET_LOCAL(0);
    dest_gh <<= 1;
    if ( (int) dest_gh < 0)
        dest_gh >>=  2;
    g_fedex_clean (dest_gh, buflist, cur_buf);
}
```

## g_fedex_clean() - Special g_fedex() for VBM Systems

This function is a special version of g_fedex() that provides faster delivery on systems with Virtual Buffer Memory (VBM), such as the ARE. The parameters are exactly the same as g_fedex(). On non-VBM systems, g_fedex() and g_fedex_clean() are equivalent.

Since VBM allocates physical pages in 256-byte clumps, it is possible for the owner of a buffer to write over a page boundary, requiring a new physical page. The VBM hardware handles this, and you normally get another page. However, if the free page pool is depleted, a "wastebasket" (WB) page is assigned to the buffer. The owner can continue writing to the buffer, but the writes go to the equivalent of /dev/null. Reading a WB page is a fatal error. If you send a buffer with a WB page to another gate, it will be dropped.

I/O devices and the backbone check for WB pages in received buffers so that only "good" buffers actually get delivered to gates. If a packet is simply forwarded through the system without adding any data (adding new link level headers is OK, as that page is real), that buffer can never acquire a WB page.

When using g_fedex() on a VBM system, GAME has to check each buffer to ensure that no WB pages are present. g_fedex_clean() skips this check and avoids walking the buffer list.

g_fedex_clean() cannot be used if any buffers in the list have been acquired via g_balloc().

## g_xmt_im() - g_xmt() With Alias Member Ignore

**Call:**

```
void g_xmt_im (BUF *buf_list, GID im)
```

**Values:**

"buf_list" is the pointer to the head of a list in the transient pool (i.e., linked by the "next" pointers).

"im" is the gate id of a local member of one or more aliases.

**Considerations:**

This function was created to efficiently handle the case where multiple interfaces on the box belong to the same "broadcast domain". All of the DP encaps gates used in this domain can join a single alias. When one interface receives a packet, and it has to broadcast it out all of the other interfaces, it uses g_xmt_im(), setting "im" to its own encaps gate. This way, the packet goes out all interfaces except the one it came it.

This behavior only applies to buffers in "buf_list" that contain alias gate handles, and only if "im" is a local member of a particular alias.

Setting "im" to zero is equivalent to calling g_xmt(). In fact, that's what g_xmt() does.

Here's another forwarding loop. This one is from the DP decapsulation routine dp_decaps_lan_act(). Since some of the packets may be bridged, and bridged packets can be flooded, g_xmt_im() is used so that the encaps gate for the local circuit is not included in the flooding.

```
/* loop for each packet */
for (rx_pkt = start;  rx_pkt;  rx_pkt = G_BUF_NEXT (rx_pkt) )
{
:
/* DP table lookup */

t_find (&tbl_lookup_result,lookup_tbl, tbl_ref);
:
    G_BUF_DEST_GH (rx_pkt) = isap_temp_ptr->isap_handle;
:
        G_BUF_DEST_GH(rx_pkt)
            cc_env->sr_env->sr_fwd_isap.isap_handle;
:
        G_BUF_DEST_GH(rx_pkt) = 0;
:
    G_BUF_DEST_GH(rx_pkt) =
        cc_env->sr_env->sr_fwd_isap.isap_handle;
:
    G_BUF_DEST_GH (rx_pkt) = 0;
:
    G_BUF_DEST_GH (rx_pkt) = 0;  /* SBAR800 */
:
    G_BUF_DEST_GH (rx_pkt) = 0;
:
            G_BUF_DEST_GH (rx_pkt) =
                isap_info[lb_index].isap_handle;
:
        G_BUF_DEST_GH (rx_pkt) = *cc_env->flood_gh;
:
}  /* end for all packets */

/* Call the g_xmt ignore member function in case we're
flooding.
    10/13/94 lp */

g_xmt_im (head, GH_GET_GID(*cc_env->im_gh) ); /* send list
to isaps
    */
```

## Reliable Buffer Delivery

The operation of reliable buffer delivery is quite different on the new Strangelove platform. While the function calls described here operate the same, the underlying details are different. Color everything here with the phrase "on the BN".

GAME's reliable buffer delivery really means "acknowledged delivery with retry and timeout". That is, after sending a reliable buffer "unreliably", if no acknowledgment is received within a certain time period, GAME will retry the transmission. After so many retransmissions, GAME gives up and returns a failure indication to the caller.

### g_fwd() - Reliably Transmit a Buffer

**Call:**

```
GH g_fwd (GH dest_gh, BUF *buf)
```

**Values:**

"dest_gh" is the is the destination gate handle for the buffer.

"buf" is the buffer requiring reliable delivery.

**Consderations:**

This function reliably delivers a single buffer to one or more instances of a gate (depending on how many bits are set in the gate handle). The gate is put into the pended state while waiting for acknowledgments. Note that the function pends even if delivery is to a local gate. This is so that the callers can be ensured that they give up the CPU when they call g_fwd() (some applications rely on this).

The function returns when a copy of the buffer has been placed on the delivery list of every requested destination gate instance or upon failure to reliably deliver the buffer to all instances.

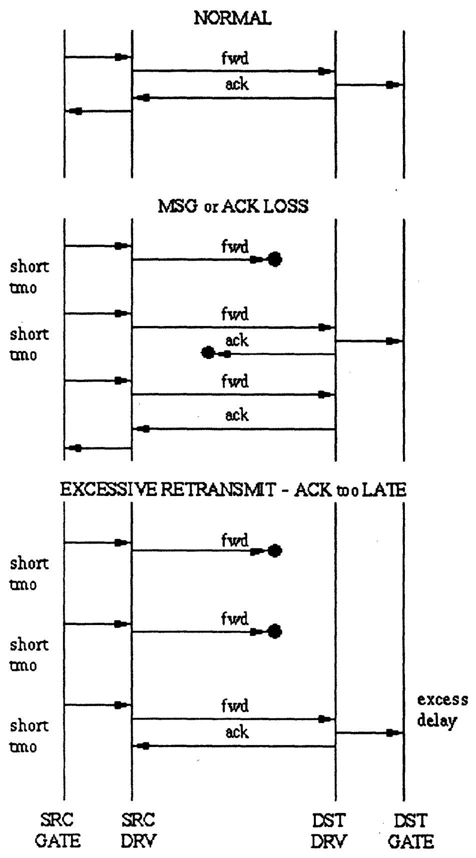
The return value is zero if all intended recipients received the buffer. Otherwise, it contains the slot bits of the gate instances that did not acknowledge receipt of the buffer.

For local delivery, GAME puts the buffer on the destination gate's delivery list, calls g_idle (G_IDLE_TAIL), and returns (successfully).

For remote delivery, after sending the buffer, GAME waits roughly 1/16 of a second for an acknowledgment from a destination slot. If an ACK is received, it then repeatedly waits to collect up any other outstanding ACKs, if the dest_gh indicated delivery to multiple slots. It then waits about 16 seconds (!) for the backbone to return the original buffer. If the buffer does not come back, a PANIC occurs (this indicates a GAME bug). In practice, this takes much less time. In fact, on the BN, the original buffer is usually returned before the ACKs.

If all ACKs and the original buffer were received, a zero is returned to the caller. If some ACKs were not received, the buffer is transmitted again (only to the applicable slots) and the whole deal is repeated. After 128 failures, the error will be returned to the caller.

Figure 5-1. Buffer Transmission



The return value of g_fwd() MUST be examined!! Failure means that the one or more indended receipients did not get the message. The return value should be checked against the current mapped GH for the receipient gate to detect a slot-down event.

**Some other details:**

If game knows a slot is down or that an instance of a gate does not exist on that slot, or if it discovers one of these situations after some amount of retrying, it will immediately mark that slot as failed and will not do any further transmissions/retries.

In a situation where a remote gate instance goes down at the same time a g_fwd() is being attempted, there is a race condition between the following events:

- The return of the g_fwd() indicating a failure.
- The local gate's mapping activation for that gate.

In other words, if the calling gate maintains a mapping for the gate it is sending to (which it should_), it may receive a g_fwd() failure before it learns of the destination gate's untimely death.

This example is from the IP code that sends routing information changes to remote slots. Having this information synchronized across slots is very important. If the operation fails, IP terminates itself. Notice that it takes the current RTM_UPDATE gate handle (up_gh) into account.

```
failed_slots = g_fwd ( dest_slots  | GID_IP_RTM_UPDATE,
    rtm_env->rtm_buf );

/* If somebody didn't get the message, CRASH! */
if (failed_slots & dest_slots & rtm_env->up_gh)
{
    g_log (IP_RTM_G_FWD_FAILURE, (dest_slots &
rtm_env->up_gh),
        failed_slots);
    CRASH (IP_CRASH);
)
```

The "and" with dest_slots in the "if" isn't really necessary, as no bits can show up in "failed_slots" that were not set in "dest_slots".

## g_rpc() - Remote Procedure Call

**Call:**

```
BUF *g_rpc (GH dest_gh, BUF *buf)
```

**Values:**

"dest_gh" is the destination gate handle for the buffer.

"buf" is the buffer requiring reliable delivery with replies.

**Considerations:**

As the name suggests, this call is used for implementing remote procedure calls. For purposes of discussion here, we'll assume a client-server relationship between gates.

When the client gate calls g_rpc(), "buf" is reliably delivered to each instance of the server gate (dest_gh), exactly like g_fwd(). After a server gate processes the buffer (possibly modifying it), it must return the buffer to the client gate via the g_reply() call:

```
void g_reply (BUF *buf)
```

Do not modify the start offset of the buffer before the start offset. This is where GAME keeps the information which maps the g_reply() buffer to a given g_rpc().

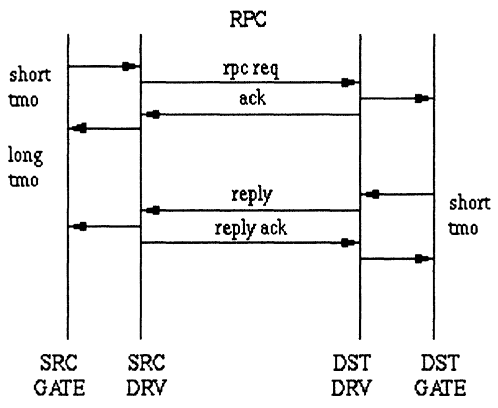
The reply buffer is also delivered exactly like a g_fwd().

GAME will wait until all replies are received or a time-out occurs. The return value from g_rpc() is a pointer to the head of the list of returned buffers. A successful call results in a returned buffer from each requested instance of the server gate. The client gate can identify which server sent a particular reply by using the g_src() call (the application can include this information in the message within the buffer, making the g_src() unnecessary). If a buffer is not received from a particular slot, no reply buffer will be included on the list. If no replies are received, the return value is NIL(BUF).

The client will wait up to 16 seconds for a single reply.

g_rpc() is implemented using the same mechanisms as g_fwd(). In this case, there are two or more reliable buffer transfers: one for the request buffer, and one or more for the replies.

Figure 5-2. Remote Procedure Call



Whereas a g_fwd() can only fail due to the inability to deliver a message to a destination gate, a g_rpc() can time out even if the receiving gate is up and healthy. If the receiving gate has a large backlog of messages to process and that processing is very CPU intensive, it may not process the sender's buffer in time to avoid the 16 second timeout (Really, we've seen it). Worse, the receiver will eventually process the buffer and send a reply. GAME will then throw away the reply because the transaction has timed out.

A gate that can get backlogged in this manner must not be used as a server gate for a g_rpc(). The API calls for the MIB service use g_rpc(). This is the mib_bind_obj() call:

```
u_int32
mib_bind_obj(obj_id, type)
OBJ_ID obj_id;
u_int32  type;
{
    BUF          *buf;/* message buffer pointer */
    MIB_ENT_MSG *msg;/* pointer to message contents */
    u_int32      ensign_gate;/* returned */

    /* first get a message buffer */
    if ((buf = g_balloc(BALLOC_TMO)) == NIL(BUF)) {
        g_log(MIB_BALLOC_ERR, g_myid());
        CRASH(MIB_CRASH);
    }

    /* get pointer to message contents */
    msg = G_BUF_INI(buf, MIB_ENT_MSG);

    /* fill message with arguments */
    msg->op_code = MIB_ENT_BIND_OBJ;
    msg->source_gid = g_myid();
    msg->bind_type = type;
    mib_copy_id(obj_id, msg->obj_id);
    /* if PRIMARY binding, get the binding entity's gate
id      --> mapping */
    if (type == PRIMARY) {
        msg->bind_gate = g_myid();

    /* for SECONDARY and OMNI bindings, set bind_gate to
zero --> no mapping */
    } else {
        msg->bind_gate = 0;
    }

    /* send message to MIB manager */
    if (!(buf = g_rpc(GH_SET_LOCAL(GID_MIB), buf))) {
        g_log(MIB_RPC_ERR, g_myid());
        CRASH(MIB_CRASH);
    }

    /* get pointer to reply message contents */
    msg = G_BUF_PDU(buf, MIB_ENT_MSG);

    /* successful? */
    if (msg->ret_code != MIB_OK) {

        /* error - kill calling entity's gate */
        g_log(MIB_BIND_OBJ, g_myid(), msg->ret_code);
        CRASH(MIB_CRASH);
    }

    /* get ensign gate */
    ensign_gate = msg->ensign_gate;

    /* free message buffer */
    g_bfree(buf, buf);

    /* done - return ensign gate */
    return(ensign_gate);
}
```

This is the code in the MIB gate that serves the mib_bind_obj() request in ent_dispatch():

```
/* get pointer to message contents */
mib_msg = G_BUF_PDU(buf, MIB_MSG);

/* dispatch on op_code */
switch (mib_msg->op_code) {
:
case MIB_ENT_BIND_OBJ:
    ent_process_msg(mib_env, &buf);
    break;
:
}

/* send reply message g_rpc initiator */
if (buf) {
    g_reply(buf);
}
```

### g_fwd_list() - Forward a List of Buffers Reliably

**Call:**

```
u_int32 g_fwd_list (GH dest_gh, BUF *head, BUF
*tail, u_int32 pipe_id)
```

**Values:**

"dest_gh" is the destination gate handle for the buffers. ONLY 1 slot bit can be set.

"head" is the head of a list of buffers in the transient pool.

"tail" is the tail of a list of buffers in the transient pool.

"pipe_id" is the return value from a previous g_fwd_list() call to the same gate, or zero.

**Considerations:**

This function provides a mechanism to reliably deliver multiple buffers to a single instance of a gate without putting the sending gate into the pended state. This is done by creating a new gate that forwards the buffers using g_fwd(). Since this can facilitate asynchronous delivery of lists of buffers, a "piping" feature allows the caller to assure that buffers from different g_fwd_list() calls to the same gate are delivered in order.

When called with a zero "pipe_id", or if the operation corresponding to a non-zero "pipe_id" is finished, GAME creates a child gate for the calling gate and puts the buffers into its first private pool. The gate is then scheduled with a SIG_INI signal. When that gate runs, it pulls the buffers off the private pool and does a g_fwd() for each one. After finishing, or if one of the g_fwd() calls fails, a data signal (data signals are explained in detail later) is sent to the calling gate to report the status. The format of the data delivered is found in include/kernel.h:

```
typedef struct FWD_LIST_STATUS
{
    u_int32 id;      /* the Id from the g_fwd_list() call
*/
    u_int32 status;  /* 0=successful,
non-zero=failure            */
    GH dest_gh;     /* the destination gatehandle           */
    u_int32 cnt;       /* how many msgs were successfully
sent      */
} FWD_LIST_STATUS;
```

The gate then terminates itself, unless another g_fwd_list() was done for the same pipe (see next few paragraphs).

The return value from g_fwd_list() is a pipe identifier that can be used to synchronize deliveries to the same gate via separate g_fwd_list() calls. By using the returned pipe ID from the previous call, the caller is assured that the next list of buffers will not be delivered before the previous list. This is accomplished by using the same child gate for delivery, if it still exists (if it doesn't exist, the previous buffers have obviously been delivered).

When the function is called with a non-zero "pipe_id" and the gate performing that pipe's g_fwd() calls still exists, GAME adds a structure to that gate's environment for the additional buffers and puts the buffers at the end of the first private pool. When the gate finishes the previous list of buffers, it checks for more lists before terminating. If found, it repeats the process of sending the buffers and delivering a signal to the calling gate.

Once there are no more buffers to deliver, the gate terminates itself.

Note that a separate status signal is sent for each g_fwd_list() call.

Example

This highly edited example is from the DLS transmit code:

```
for (buf = buf_head; buf; buf = buf_next)
{

    /* we have to keep the next buffer in case we delete or
send */
    buf_next = G_BUF_NEXT(buf);
    :
    /* set the destination GH */
    G_BUF_DEST_GH(buf) = rem_gh;
    :
        buf_end = buf; /*DF CR20602*/
    :
} /* for each buffer */

/* send the list onward */
if (buf_head)
{

    /* do the reliable forward */
    sock->loc_pipe_id = g_fwd_list(rem_gh, buf_head,
buf_end,
        sock->loc_pipe_id);
}

/* we don't need to set the timer here because we will get a */
/* signal back from g_fwd_list which will wake us up */
```

## Debug Tips

The "debug krnl" command provides a few settings of use for buffer delivery debug (note that the "debug" module must be loaded). (The use of debug_krnl is discussed in file /rte1/harpoon/doc/game/html/ game_debug.html.)

Table 5-1 describes the buffer delivery debug settings.

Table 5-1 Buffer Delivery Debug Settings

| Setting | Function |
|---|---|
| msg_xmt | Logs messages if xmt buffers are being sent to gates which we don't think exist on remote slots. Applicable to g_xmt(). |
| msg_deliver | Logs message and dumps buffer if message is received for a gate which is not present on the receiving slot. |

| Setting | Function |
|---|---|
| buf_chk | Verifies buffers are valid, on the same list and owned by the caller. Applicable to g_fwd(), g_reply(), g_rpc(), g_fwd_list(). |
| buf_size | Verifies the buffer end is less than the max buffer size and that start is less than end. Applicable to g_fwd(), g_reply(), g_rpc(), g_fwd_list(). |
| buf_xmt | Verifies xmt buffers are valid and the start and end offsets are correct. Applicable to g_xmt(). |
| all_msg | Is the combination of msg_xmt and msg_deliver. |
| all_buf | Is the combination of buf_chk(), buf_pool(), buf_size(), buf_xmt(). The former two settings were discussed in the Buffers section. |

## Signals

Buffer delivery, as described in the previous section, is a general purpose mechanism that has one down side: it requires dedicated resources (buffers) to function.

Buffer delivery is the only option when data has to be sent across slots. When communicating between gates on the same slot, there are cases where buffers are overkill for several reasons:

1. Only a small amount of information needs to be conveyed.
2. The communication has to happen frequently.
3. There is a short latency requirement.
4. Buffers are a more precious resource than memory.

## Uses of Signalling

Signalling is used primarily for four purposes: interrupts, timer expirations, software signals, and memory passing.

### Interrupts

Hardware interrupt delivery is strictly controlled by GAME (see the Scheduler section). Interrupts are intercepted by the kernel and translated to software signals, allowing GAME to schedule interrupt handlers just like all the other gates.

### Timer Expirations

Timer expirations (see the Timer section) are conveyed via delivery of the SIG_TMO signal (see include/vectors.h).

### Software Signals

A limited number of simple software signals are supported (see include/ vector.h). The receiving gate gets activated with the signal number and no additional information.

### Memory Passing

The SIG_DATA signal is used to pass a memory segment (obtained via g_malloc) from one gate to another. This allows arbitrary amounts of data to be transferred without using buffers.

## Signal Handling Urgency

From the GAME scheduler point of view, there are two classes of signal handling gates:

1. Normal, "base level" handlers that are scheduled for signal delivery in a FIFO fashion among all other base level gates.
2. Low latency, "interrupt level" handlers that are scheduled to run at the nearest opportunity (when the currently active gate completes or pends).

There is a strict prioritization between signal handing gates. Interrupt level handlers are scheduled ahead of base level handlers and all other gates. However, once activated, the gate handling a signal is never interrupted or preempted (unless it voluntarily gives up the CPU).

## Using Signals

A single gate instance can register to receive one signal (as defined in include/vectors.h) and a single signal can, in most cases, be handled by only one gate. In some cases, the gate does not need to explicitly register for the signal, but it can still receive only one. The caveat is that a gate can always receive two additional special signals, which are always delivered in a "base level" fashion:

1. SIG_INI. This signal is delivered upon the creation of a gate instance if the creator set the G_SIG_INI flag in the g_req() call.
2. SIG_TMO. This signal is delivered every time the gate's periodic timer expires.

A gate that is activated due to a signal delivery, whether at the base or interrupt level, is passed a signal vector number instead of a buffer pointer (the "buffer list" parameter is NIL). Besides this difference in passed arguments, the two gate activation modes are identical (run to completion uninterrupted, full access to system resources, etc...).

A single gate instance cannot have both a buffer and a signal activation (active/pended) at the same time. As discussed later in the Scheduling section, if a signal activation is scheduled while the gate is pended during a buffer activation (or vice versa), GAME remembers this and schedules the gate for the new activation when the old one exits.

## Registering for a Signal

**Call:**

```
void g_isr (GID gid, SIG sig, u_int32 flag)
```

**Values:**

"gid" is the gate id of the signal handler gate.

"sig" is the signal number to be handled (from include/vector.h).

"flag" is the signal handling option flag; choose one of the following:

- G_ISR_SIG - interrupt (high priority) signal handler
- G_BASE_SIG - base level (low priority) signal handler
- G_CANCEL - cancel signal handling

**Considerations:**

This function call tells GAME which gate is handling the particular signal on the local slot. The calling gate will be terminated for any of the following offenses:

1. The gate "gid" is not instantiated on the local slot.
2. The gate "gid" is registered to handle a different signal.
3. Some other gate is registered to handle "sig".
4. "flag" was set to G_CANCEL and the gate is not handling "sig".

Otherwise, whenever a gate calls g_sig("sig"), the signal will be delivered to "gid". The scheduling of the signal delivery depends upon the "flag" parameter.

If "flag" is set to G_CANCEL, the gate will no longer be scheduled for "sig".

A gate never registers for SIG_INI, SIG_TMO, or SIG_DATA signals. It also does not register for signals sent via g_sig_gid() (later...).

## Sending Signals

There are three functions calls in GAME that result in a signal delivery:

- g_sig
- g_sig_gid
- g_sig_data()

Gates cannot send SIG_INI or SIG_TMO signals; only GAME can do this (SIG_INI delivery is initiated by some gate's g_req() call, however).

**g_sig Call:**

```
void g_sig (SIG sig)
```

**Values:**

"sig" is the signal to be delivered.

**Considerations:**

This function is only used for signals where the receiver does an explicit g_isr() call. The registered gate is scheduled to receive the signal, based upon the "flag" parameter used in the g_isr() call.

If no gate is registered for the signal, or if the signal has been previously delivered with the g_sig_gid() call, no signal is delivered.

The driver for the ILACC chip, which is used on the quad ethernet boards, registers for an interrupt associated with the connector the driver services:

```
env->line_sig = (SIG)(SIG_CSMACD + env->line);
:
/* Register for interrupts. */
g_isr(env->gid, env->line_sig, G_BASE_SIG);
```

The QENET hardware interrupt handler (real interrupts) sends a signal when an interrupt is received from the chip:

```
/* dispatch off MISR signaling Line Drivers */
misr = *(u_int8 *)(hwrec->wfModMisr);
if (!(misr & ILACC1))
    g_sig(SIG_CSMACD + CSMACD_CONN_ONE);

if (!(misr & ILACC2))
    g_sig(SIG_CSMACD + CSMACD_CONN_TWO);

if (!(misr & ILACC3))
    g_sig(SIG_CSMACD + CSMACD_CONN_THREE);

if (!(misr & ILACC4))
    g_sig(SIG_CSMACD + CSMACD_CONN_FOUR);
```

The ILACC driver gate is eventually scheduled for the signal:

```
void
ilacc_up_state(env, buf, signal)
REG ILACC_ENV *env;/* ptr to parents environment */
REG BUF *buf; /* buffer pointer list, packets to xmt */
REG SIG signal; /* ILACC int signals or watchdog SIG_TMO`s */
{
    /* if there are buffers to be transmitted */
    if (buf != NIL(BUF)) {
        ilacc_xmt(env, buf);
    }
    /* OR, if this is an interrupt signal from device */
    else if (signal == env->line_sig)
        ilacc_intr(env);
```

**g_sig_gid Call:**

Given that the number space for signals is small (256), it is desirable to be able to use the same signal for multiple gates that are running essentially the same code. As shown in the previous section's example, multiple copies of a device driver may run on the same slot due to the number of physical interfaces on the link module, potentially using a signal per interface. Therefore, g_sig_gid() was born.

```
u_int32 g_sig_gid (GID target_gid, SIG sig, u_int32 option)
```

**Values:**

"target_gid" is the gate id of the destination gate on the local slot.

"sig" is the signal to be delivered.

"option" is G_BASE_SIG (base level delivery) or G_ISR_SIG (interrupt)

**Considerations:**

The return value is non-zero if "target_gid" doesn't exist and zero otherwise.

The receiving gate must NOT register for the "sig" or any other signal. It also cannot receive data via g_sig_data().

The calling gate will be terminated for any of the following offenses:

1. The receiving gate has registered for any signal.
2. The receiving gate has received data via g_sig_data().

The munich driver, used for the multi-channel T1 and E1 boards, uses g_sig_gid() in order to save on signals. Otherwise, with 96 logical lines possible per slot, that many signal numbers would be necessary.

There are many calls in the driver similar to this:

```
g_sig_gid (env->ldg_gid[i], SIG_LOG_LINE, G_BASE_SIG);
```

"ldg_gid" is an array of the gate ids for the "line driver gates".

**g_sig_data Call:**

The g_sig_data() call really serves two purposes. Its a way to send multiple signals to a single gate as well as a way to move memory segments between gates.

The first step is for the sending gate to call g_sig_data() and pass in the address of a memory segment. This must be the same address as was returned by a g_malloc(). It's not possible to send partial segments. If the g_sig_data() call is successful, the sender no longer owns the memory. Any pointers to that memory should be nulled out.

```
u_int32 g_sig_data (GID dest_gid, u_int32 type, void *data)
```

**Values:**

"dest_gid" is the gate id of the destination gate on the local slot.

"type" is the data signal type, as defined in include/data_sig.h

"data" is the pointer to the memory segment to transfer. If NIL, "type" is the only information delivered to "dest_gid". This is how g_sig_data() is used to deliver multiple signals.

**Considerations:**

The return value is zero if the memory was delivered and non-zero otherwise. The only reason for the non-zero return is if the destination gate does not exist locally. The calling gate still owns the memory segment in this case.

The calling gate will be terminated for any of the following offenses:

1. "dest_gid" is zero.
2. "dest_gid" is scheduled to receive a different signal (other than SIG_INI, SIG_TMO).
3. "dest_gid" is registered to handle a signal (DEBUG only).
4. The calling gate does not own the memory segment (DEBUG only).

GAME gives the memory segment to the receiving gate and schedules it for a SIG_DATA signal (unless already scheduled).

When the receiving gate is invoked with the SIG_DATA signal, it must g_get_sig_data() to retrieve the memory. It is possible to have multiple data deliveries before the reciever gets scheduled. The receiver will only see one SIG_DATA activation. Therefore, it needs to do g_get_sig_data() in a loop until told there is no more data available (zero return). The order of signal delivery is preserved.

```
u_int32 g_get_sig_data (GID *send_gid, u_int32 *type, u_int32
**data,
    u_int32 *size)
```

"send_gid" is a pointer to a (GID) where the gid of the gate that sent the signal can be written.

"type" is a pointer to a (u_int32) where the data signal type, as defined in include/data_sig.h, can be written.

"data" is a pointer to a (u_int32 *) where the data pointer can be written.

"size" is a pointer to a (u_int32) where the data size (in bytes) can bewritten. The size returned is the total size of the memory segment. This is rounded up to a cache line size and may not be the size of the actual data contained within the segment. This is to aid in the reuse of memory segments for different purposes.

The return value is non-zero if data was delivered and zero if not.

As with buffer delivery, the format of the messages delivered must be agreed upon by the sender and the receiver (i.e., located in a *.h file).

If g_sig_data() is being used for message delivery (as opposed to transfer of "permanent" memory ownership), the receiving gate must free the memory segment when it is finished examining it.

DP implements a generic service that will allocate a "resource" and deliver it to destination gates, using memory if the destination is local and a buffer otherwise.

This is the routine that allocates either a memory chunk or a buffer:

```
u_int32 get_sigbuf(GH gh, u_int32 struct_size, u_int32 **ptr)
{
    :
    if (GH_IS_REMOTE(gh))
    {
        /* Allocate a buffer, adjust to point to data and
return */

        BUF *buf;
        if ( !(buf = g_balloc(G_TMO_DEFAULT)) )
        {
            g_log(DP_NO_BUF);
            CRASH(DP_CRASH);
        } /* if ! buf = g_balloc */
        G_BUF_START(buf) = G_BUF_START_MSG;
        G_BUF_END(buf)   = G_BUF_START_MSG + struct_size;
        *ptr = (void *)((char *)buf + G_BUF_START_MSG);
        :
        return(IS_BUFFER);
    }
    else
    {
        /* Allocate a piece of memory and return it */
        /* This will eventually be optimized to use free
pools */
        *ptr = g_malloc(struct_size);
        zero((u_int8 *)(*ptr), struct_size);
        return(IS_SIGNAL);
    }
} /* get_sigbuf */
```

This is the routine that sends the memory or buffer:

```
GH send_sigbuf(GH gh,u_int32 m_type, u_int32 m_size, u_int32
**ptr)
{
    char log_buf[120];
    GH g_fwd_rtn_mask = 0;
    :
    if (m_type == IS_BUFFER)
    {
        /* Is remote */
        /* Get back to your buf pointer */
        *ptr = (u_int32 *)((char *)*ptr - G_BUF_START_MSG);
        g_fwd_rtn_mask = g_fwd(gh,(BUF *)(*ptr));
    }
    else
        if ((GH_IS_REMOTE(gh)) && (m_type == IS_SIGNAL))
        {
            u_int32 *b_ptr;

            /* GH became remote on us.. Aaaaaagh! Copy it
into a
                buffer fast...  */
            m_type = get_sigbuf(gh,m_size,&b_ptr);
            bcopy((u_int8 *)(*ptr),(u_int8 *)b_ptr,m_size);
            g_mfree(*ptr);
            g_fwd_rtn_mask =
                send_sigbuf(gh,m_type,m_size,&b_ptr);
        }
        else
            if (m_type == IS_SIGNAL)
            {
                if (g_sig_data(GH_GET_GID(gh),
                    (u_int32)(**ptr),(void *)*ptr))
                {
                    /* Set our local slot bit to indicate that
                        signalling failed to our slot */
                    GH_SET_LOCAL(g_fwd_rtn_mask);
            }

        /* Remember , your ptr will not be valid anymore */

        return(g_fwd_rtn_mask);

} /*send_sigbuf */
```

The return value from g_sig_data() should be examined. If non-zero, g_fwd_rtn_mask should be set to the local slot bit.

Also, copy() should be used rather than bcopy(). This function attempts to optimize the data copy if the data alignment allows.

## Mappings

Mappings can be used as a messaging mechanism to indicate events to gates on the same or on remote slots.

The basic idea is that a gate is created to indicate a particular state. Gates interested in that state map the "state" gate. When the state is no longer valid, the "state" gate is killed, causing the mappings to trigger.

Note that the creating and killing of gates involves a non-trivial amount of CPU. This procedure should be avoided for frequent events.

The prime example of this is the MIB service. It uses davidian gates (which limits the "messaging" to local slots) to represent:

1. The contents of a row instance of a table. Whenever a "restart" variable in the row is changed, the davidian gate is killed.
2. The creation of a table row instance. Whenever a new row is added to a table, the davidian is killed. It is not killed when a row is removed (the application has to do the removal, so there's no need to tell it).

The use of ensigns or davidians for this purpose is problematic, since GAME does not clean up these gates when the creator dies (there's no GATE structure, and hence, no ancestry information). The MIB can get away with this because if the MIB dies, the whole slot goes down. Therefore, use real gates with dummy activation routines instead until told otherwise.

# Chapter 6 Memory Management

## Overview

GAME has a single pool from which all memory is allocated, except for buffers. In addition to the typical malloc/free memory, this also includes the memory used for stacks and the memory in which an application executes.

The memory pool is composed of slabs and segments. A "slab" is a single large chunk of memory which gets divided up into smaller pieces called "segments". Normally, there is only one slab of memory in the system. It runs from the end of the kernel image to the end of normal memory, or to the beginning of buffer memory on systems, such as the AN, where buffer memory isn't implemented in separate hardware.

Each memory segment contains a header, called a MSEG, at its beginning. This contains previous and next pointers to link the MSEG onto a doubly linked list. It also indicates the size of the MSEG. The MSEG occupies the first cache line of a segment. All MSEGs are cache line aligned. (A cache line is 16 bytes for the 68k, 32 bytes for the PPC.) This alignment is required by tags (see below). This alignment also results in the rounding up of the size of a MSEG to whole cache line increments.

The MSEG is shown below. The flag bit is a way for the kernel to mark certain MSEGs as special so that they can be found on a gate's memory list. For example, if a gate owns a semaphore token, that token is actually represented by an MSEG linked into the gate's memory list. That MSEG will have flag set and the first word of the MSEG body will tell the kernel that it's a semaphore token. This was implemented in this manner to avoid having to double the size of the gate control block (64 bytes at the time). That has since become unavoidable, so we now have room to track these on their own list, if we so desired.

The PowerPC has a larger cache line size than the 68k does. That explains the extra padding when PPC is set.

```
 typedef struct MSEG
 {
     struct MSEG *next;      /* next seg on singly/doubly
linked list   */
       struct MSEG *prev;      /* previous seg on doubly
linked list      */
     unsigned     flag:1    /* See note below             */
     unsigned     size:31;  /* segment length              */          *,
     struct MSEG *resv;      /* reserved                    */

#ifdef PPC
       u_int32      pad[4];  /* Pad to 32 byte cache line    */
#endif /* PPC */

} MSEG;
```

An MSEG can be linked in one of two places: the free memory pool or onto a gate's memory list.

The free memory pool is a list of all the MSEGs which are available for allocation. This list is always arranged in order of increasing memory address. When a gate asks to allocate some memory, the free memory pool is linearly searched until a big enough MSEG is found (first fit). If this MSEG is larger than what was asked for, it is split into two pieces. One piece goes to the gate and the other remains in the free pool.

When memory is freed, it is inserted back into its place in the free memory pool. This insertion is aided by a binary tree whose pointers occupy the 2nd cache line of MSEGs in the free memory pool. Once the appropriate place in the free memory pool is found, a check is made to see if the range of memory covered by the MSEG being freed abuts the memory of its neighbors in the free memory pool. If this is the case, then the MSEG is merged in with its neighbor(s), resulting in a single, larger, MSEG in the memory pool.

This memory allocation scheme results in the beginning of the pool containing many smaller memory segments while the end of the pool contains the larger segments. This happens because we always start searching from the same end and will take the first segment which fullfills the allocation requirements.

When a gate allocates memory, it is placed on a list of memory owned by that gate which is anchored in the gate control block.

```
typedef struct GATE
{
    ...

  MSEG      *mem;      /* gate's reserved memory              */

    ...
} GATE;
```

This list is used to reclaim the memory a gate has allocated should that gate die or be killed. There is no particular ordering to elements on the list. For simplicity, new segments are put at the head of the list.

The first element's "prev" pointer points to the "mem" field of the GATE structure.

An allocated memory segment is actually larger than the size requested. As stated above, an MSEG precedes the block. The entire block is always padded out to the end of the current cache line. Therefore, the size of an allocated block is:

```
size + (size mod cache-line-size) + cache-line-size
```

This is the size stored in the "size" field of the MSEG header.

All this memory segment management is typically ignored by a gate. When a gate allocates memory, it recieves a pointer to the first useable location (after the MSEG). GAME wants this same pointer back when memory is freed. It is the application's responsibility to limit both read and write accesses to the allocated memory segment.

An application must NEVER access the MSEG header preceeding a memory segment. There are at least two good reasons not to do this:

1. The format may change under the application.
2. On hardware that supports tags, you'll get a tag violation (see the next topic).

## Tags

Since all addresses in GAME are logical=physical and no MMU is used, some type of memory protection was needed to help with the debugging of bad pointers. The memory protection is called 'tags'.

Tags are implemented in specialized hardware on the FRE, ASN, and ARE platforms:

Tags allow the kernel to mark each cache line with an attribute describing its readabilty or writeability. Cache lines can be marked read/write, read-only, or no-access. If an illegal access is made (e.g. writing a read-only cache line) an exception occurs.

The MSEG header is marked read-only. The intention of marking the beginning of a memory segment read-only is to catch errant code which walks past the end of the memory its allocated. This can also happen if a stack grows too large, since a stack is simply a memory segment.

Tags will not prevent a truly errant pointer from causing problems. It is possible for that pointer to miss a read-only cache line and successfully modify data. To help combat this, it is possible to have all freed memory (except the headers) marked no-access. This helps if the bad pointer happens to hit freed memory. However, this debug feature results in a performance hit as the whole memory segment needs to be walked whenever memory is allocated or freed. Obviously, this is turned off by default. See Debugging Strategies for more details. This does not help if the bad pointer points to memory owned by another gate.

One of the things to keep in mind about tags is that they are implemented by HW extraneous to the processor. Therefore, a tag violation won't occur until the data is flushed out of the data cache.

Usually (on a FRE2) this happens well after the fact and it is not possible to say where the bad access happened. To combat this, it is possible to run the processor in write-thru mode where all writes will immediately go to the memory system. When running in write-thru mode, the tag violation will occur before the program counter advances too far beyond the instruction that caused the violation. Write-thru mode is not the default and needs to be enabled. See Debugging Strategies below to see how to do this.

Note that the ASN is the only platform that reports (via the log) the memory address used to cause the tag violation.

## Ownership and Memory Sharing Implications

As mentioned above, when a gate allocates memory, that memory is added to a list of all the memory allocated by the gate. Upon gate death, all this memory is freed. This has implications with memory sharing between gates.

The preferred way to share memory is downward, where a parent owns the memory shared with its children. This works nicely because if a parent dies, all its children will also die. Sharing memory upward, where the children own memory manipulated by its ancestors, as well as memory sharing between unrelated gates, is dangerous because that memory may be freed without warning. This can leave a gate with a memory pointer to what is now free memory. Or worse, the memory may have been re-assigned to some other gate.

The chance of this can be somewhat minimized if the gate which is sharing memory maps the memory owner gate. But even a mapping isn't fool proof because you need to remember that the memory could be freed while you're pended in the middle of a function. That function may have pointers to the now freed memory cached in local variables. Using these variables becomes dangerous.

There are some ways around this. If a few children of a gate need to share memory, it is possible for them to use g_malloc_gid() to allocate memory on behalf of the parent. When doing this, the parent will become the owner of the memory segment, so all children can access it freely. The tbl, rtbl, and utbl utilities all use g_malloc_gid(). All memory is allocated in the context of the gate that creates the table.

Another method is to use g_sig_data() to move memory ownership from gate to gate.

## Syscalls

### g_malloc()/g_mfree()

The g_malloc() and g_mfree() system calls are the normal way to allocate and free memory. A gate must own the memory to be able to free it. Otherwise, g_mfree_gid() needs to be used.

#### g_malloc() Call

The g_malloc() call allocates memory segments.

**Call:**

```
void *g_malloc (u_int32 size)
```

**Values:**

"size" is the requested segment size in bytes.

**Considerations:**

The returned value points to the first usable byte in the segment.

If adequate memory is not available, the slot will be restarted due to an out-of-memory condition. Because of this, there is NO NEED to check the return value of g_malloc(). Doing so is just a waste of instruction space and CPU.

### g_mfree() Call

The g_mgree()call frees memory segments.

**Call:**

```
(void) g_mfree (void *mem)
```

**Values:**

"mem" is a pointer to a memory being freed. This must be the same value as returned by a previous g_malloc().

## g_mlen()

**Call:**

```
u_int32 g_mlen()
```

**Considerations:**

The g_mlen() call returns the size, in bytes, of the largest memory segment available. This is the largest g_malloc() request which can be satisified. If a g_malloc() call is made and not enough memory is available to satisify that request, the slot will restart due to an out of memory condition. To avoid that, this type of code can be used:

```
 if(g_mlen() > amount_I_need)
{
    pointer = g_malloc(amount_I_need);
}
else
{
    /* Couldn't get the memory I wanted.. now what? */
}
```

The only time where this is helpful is when the memory allocation is not crucial to the continuance of the application. Otherwise, the g_mlen() call isn't much help.

There is a small window on SMP systems where a processor can allocate memory between g_mlen() and g_malloc() calls executed by another processor. Therefore, a positive result from g_mlen() does not guarantee that the g_malloc() will succeed.

## g_malloc_gid()/g_mfree_gid()

The g_malloc_gid() and g_mfree_gid() syscalls work just like the g_malloc() and g_mfree() calls except that the gate ID of the owner gate can be specified. Care should be used when using these calls since its possible to abuse them. For example, a child can g_malloc_gid() memory and have its parent own the memory. If that child dies, the parent may need to clean up the memory which was allocated, The kernel isn't going to do it because the parent owns it. Failure to handle such scenarios correctly this could result in a memory leak.

### g_malloc_gid() Call

The g_malloc_gid() call allocates memory for another gate.

**Call:**

```
void *g_malloc_gid (u_int32 size, GID gid)
```

**Values:**

"size" is the size of memory segment to allocate.

"gid" is the gate id of the gate to own the memory.

**Considerations:**

The returned value points to the first usable byte in the segment.

Calling g_malloc_gid() with an invalid gid will terminate the calling gate.

The calling gate does not own the memory segment unless gid is its own gate id. The calling gate must realize this and use care when using this memory (freeing, etc.).

## g_mfree() Call

The g_mfree_gid() call frees memory owned by another gate.

**Call:**

```
void g_mfree_gid(void *mem, GID gid)
```

**Values:**

"mem" is a pointer to the memory segment to be freed. This must be the same value as returned by a previous g_malloc_gid().

"gid" is the gate id of the gate owning the memory.

**Considerations:**

Calling g_mfree_gid() with an invalid gid will terminate the calling gate.

## g_mrealloc()

The g_mrealloc() call reallocates a memory segment. It copies the contents of one memory segment to a new one, frees the old one, and returns the new one.

**Call:**

```
void *g_mrealloc(void *old_mem, u_int32 new_size)
```

**Values:**

"old_mem" is a pointer to currently allocated segment. The calling gate must own the segment.

"size" is the byte length of new segement to allocate.

**Considerations:**

The return value is the pointer to the newly allocated segment. The contents up to MIN (size, sizeof(old_mem)) from old_mem will have been copied to new_mem.

If adequate memory is not available, the slot will restart due to an out-of-memory condition.

## g_madd()

The g_madd() call is used to add a new slab to the free memory pool. This is rarely used. The only example to date is with netboot where the config file is stored in memory while the mission code starts. Once the config file has been read, its memory can then be used as a normal part of the memory pool.

**Call:**

```
void g_madd (u_int32 new, u_int32 size)
```

**Values:**

"new" is a pointer to the memory slab.

"size" is the byte length of the new memory slab.

### g_sig_data()/g_get_sig_data()

The g_sig_data() call really serves two purposes. It is a way to send multiple signals to a single gate as well as a way to move memory segments between gates.

When the receiving gate is invoked with the SIG_DATA signal, it must g_get_sig_data () to retrieve the memory.

These function calls are discussed fully in the Inter-Gate Communications chapter.

## Debugging Strategies

There are a few strategies which are useful for debugging memory problems.

### Zero Out Those Stale Pointers!

When you free memory (or send it via g_sig_data()), set that memory pointer to 0. This way, if you subsequently try to use that pointer you'll get a bus error from the NULL pointer instead of corrupting memory. A bus error is much easier to debug than memory corruption.

### debug krnl Command

The "debug krnl" command provides a few settings useful for memory debug (note that the "debug" module must be loaded).

## mem_free_check

mem_free_chk will check that memory you free is indeed memory you own. GAME normally doesn't check this. If you free memory you don't own, it is possible for parts of the gate control block to become write protected (because GAME thinks they're another MSEG header when they're not). This can lead to tag violations in strange places. This is applicable to calls to g_mfree().

## mem_full_tags

mem_full_tags will mark freed memory as no-access. It makes the box run really slow. It can be useful for catching stale pointers assuming, of course, that you're lucky enough to have the pointer point to freed memory and not some other gate's allocated memory. (If you set your freed pointers to 0 you wouldn't have this problem.) You need to restart the slot after setting this for it to take effect.

## wrt_thru

wrt_thru will cause the processor to run in write-thru mode and is the first thing you should do when debugging a tag violation. You need to restart the slot after setting this for it to take effect.

## serial

serial is only for PowerPC machines. It forces the processor to run in serial mode, which will give more accurate stacks when a tag violation occurs. You need to restart the slot after setting this for it to take effect.

### mem_all

mem_all is the he combination of mem_free_chk, mem_full_tags, and wrt_thru.

Sometimes its desirable to have the processor default to write-thru. This is especially true if the tag violation happens before the TI is up and running. This is easy to do.

For TIB:

1. cd tib
2. In Makefile, uncomment the CACHE_MODE symbol.
3. rm _tib/set_cfg_regs.o.
4. build tib set_cfg_regs.o
5. cd buildtib
6. build tib link archive -nr

For BF:

1. cd bf
2. In Makefile, uncomment the CACHE_MODE symbol.
3. build bf cache.o
4. cd buildbf
5. build bf link archive -nr

## Private Memory Management

One aspect of the GAME memory system which is not so good is that is isn't very efficient dealing with many small memory pieces. First of all, the size of each piece is always rounded up to a cache line. Then an MSEG header and guard line are included. So, for a small request (say, 4 bytes), 48 bytes of memory are actually required.

Then, there is the additional processing of allocating and freeing the memory. If many allocs and frees are done, this processing starts to be non-trivial and can really affect performance.

To help overcome these shortcomings, a series of Private Memory Managers (PMM) have been created. These are composed of a family of macros which work above GAME's MSEG allocation and allow a gate to allocate one larger segment from GAME and partition it up into little pieces.

Allocating a single segment satisifies GAMEs requirement of re-claiming all a gate's memory. Alloc and frees are then much more efficient since there is no need for cache line sized allocates or MSEG headers, since all the allocs and frees take place from within the single MSEG.

There are a few different flavors of PMMs available. Details for these can be found in include/pmm.h.

Here is a quick list of the available PMMs and where they can be used:

### Simple Private Memory Manager

Suggested When:

1. Allocations are of a fixed size (ex. table entries)
2. Space is not a concern, since slabs are not freed (slabs are freed only when PMM_S_END is called)
3. Memory utilization tends not to decrease in time
4. Freeing of segments is infrequent

### Space-Recovering Private Memory Manager

Suggested When:

1. Allocations are of a fixed size (ex. table entries)
2. Allocating and freeing segments occurs regularly
3. Memory space is to be freed back to GAME regularly

### Space-Compacting Private Memory Manager

Suggested When:

1. Allocations are of a fixed size (ex. table entries)
2. Allocating and freeing segments varies greatly
3. Most efficient use of memory resources is needed, but has these drawbacks:
    - Performance suffers due to relocating/copying segments into as few slabs as possible.
    - Segment pointers should not be cached since what is returned by PMM_C_GET() is simply a handle to the segment.
    - Client variables pointing to PMM_C segments must be declared as: <data type> **<var>; since the segment handle returned is a pointer to a pointer.
    - Requires use of PMM_C_REF(). Note, always use PMM_x_REF() if switching between PMM_C and other PMM managers.

### Pool-Of-Private-Pools Memory Manager

Suggested When:

1. Allocations are of a variable size
2. Allocating and freeing segments occurs regularly
3. Memory space is to be freed back to GAME regularly
4. The demand is a small number of popular segment sizes

### Variable Size Segment Private Memory Manager

Suggested When:

1. Allocations are of a variable size
2. Allocating and freeing segments occurs regularly
3. Memory space is to be freed back to GAME regularly
4. Requested segment sizes are randomly spread

This scheme could be slower due to its splitting, merging and chaining. It can also suffer from fragmentation, unpredictably, based on its use. The tradeoff is an increase in buckets makes for a larger hash table thus increasing PMM_V overhead space. But, this increase also reduces fragmentation and speeds up the search process when fetching for free segments.

## free_pool

Another effort overlapped the development of PMM. The files include/free_pool.h and rtl/free_pool.c implement a private memory manager similar to the "Simple Private Memory Manager" above.

## What Size memory Segments to Allocate

You only get a 'win' when reducing the size of a particular structure which is g_malloc()'d when the size is reduced below a new 16-byte boundary. For example, if a structure is 128 bytes long, and two 4-byte fields are removed, this doesn't buy anything. If 16 bytes are removed, this will (in most cases) reduce the size GAME allocates by 16 bytes.

What is the win if I allocate memory for "n" structures in one big block, as opposed to doing "n" separate g_malloc() calls? That is, how much game header overhead am I avoiding?

The overhead is 16 bytes for every separately malloc'd block. This is the memory we use to keep track of what memory has been malloc'd by a gate so that we can free it if the gate dies. In your example above, you'd save (n - 1) * 16 bytes by doing 1 big malloc instead of n smaller ones.

A friendly word of warning... don't malloc it if you really don't need it. We're liable to run out of memory just like we run out of buffers. Also, once the memory space gets fragmented, a malloc of a big chunk is more likely to fail than a malloc of a small chunk. You can use g_mlen() to see what the biggest contiguous chunk currently is.

For each GAME memory segment (MSEG) allocated, GAME allocates some additional overhead (16 bytes) to manage the memory. If an application needs lots of a certain data structure, and g_malloc()'s memory for each instance, the penalty for this overhead can be significant. In the extreme, the overhead can exceed the actual data structure size itself.

## (Shared) Free Pools

The free pool code (rtl/free_pool.c, include/free_pool.h) provide an efficient management pool of fixed sized data structures. The improvement over using GAME's native memory manager comes from amortizing the GAME MSEG overhead over a large number of the applications data structures. The free pool code will allocate larger chunks of memory from GAME (the size is programmable) and chop that into the size appropriate for the application which can be allocated from and freed to the application specific pool, rather than the GAME memory manager.

Currently these pools are not shrinkable; they only expand when the pool is depleted. The designer should be aware that a pool can suddenly grow dramatically in size, perhaps due to some transient burst of activity in the system. After the spike has passed, the over-grown pool is not subsequently shrunk, even though that memory may never again be required (i.e. if the spike happens only once in the life of the application during init, it will continue to own/consume large amounts of memory that it may never use again)

Recently, the free pool code was enhanced to allow multiple gates to access the same pool. The intent is for a parent gate to own a pool, but allow children (and or grandchildren etc.) gates to use/access/expand the pool and the parent's environment in general. This sharing of environments has proven be an effective way to improve performance. In some applications, however, it makes the gates or sub-system in question inherently "dirty" in SMP term. The benefits of the shared environment must be carefully considered, especially going forward as it appears that SMP systems may begin to dominate the scene.

# Chapter 7
# Scheduler

## Overview

The GAME gate scheduler is a simple, non-premptive, first in/first out (FIFO) scheduler. This means that a gate executes until it gives up the CPU by either pending or returning from its activation routine. It also means that gates will execute in the order in which they are placed onto the scheduler queue (with a few exceptions).

## Scheduler Queues

There are two scheduler queues in GAME. These are the Activation queue and the Idle queue. Each queue element contains a pointer to the gate control block of the gate to activate as well as the reason for theactivation.

The Activation queue is a list of gates which are ready to be run. The scheduler will walk though this list activating each gate in turn. When a running gate either pends or returns from its activation routine, the next gate in the list is activated.

Once the activation queue is empty, the system is said to go "idle". At this point in time module interrupts are handled. If there are indeed module interrupts pending this will result in some gates (such as a link driver) being added to the activation queue. Once all gates needed for interrupt processing are added to the activation queue, the contents of the Idle queue are copied to the activation queue. Then the scheduler starts executing the gates on the activation queue.

The Idle queue serves as a place for application gates to go when they want to be "fair" (or put another way, when they don't wish to kill the slot's forwarding performance). Since the scheduler is non-preemptive, it is possible for a single gate to usurp all of the processing resources of the system for a long time. This is undesirable in a system which is also trying to pass data traffic. The idle queue allows a gate to timeslice itself via the g_idle() syscall. By calling g_idle() a gate will allow more network traffic to be processed after which it will continue execution.

There is a CPU watchdog which will prevent a gate from running forever. After some large amount of time (3-4 seconds on most systems), if the same gate is still running, this gate will be killed and a "cpu hog" event will be placed in the error log. But, the CPU watchdog is really only there to prevent runaway gates from hanging the system. Packets will be dropped well before the CPU watchdog goes off, so it is up to the gate to idle itself well before the CPU watchdog limit. See "CPU Hogging" ahead. Watchdogs are discussed more in the Watchdog section.

## Activation Reasons

A gate can only be activated for seven reasons, as follows:

1. Message Delivery
2. SIG_INI
3. SIG_TMO
4. User-Defined Signal
5. SIG_IDLE
6. SIG_MAP
7. SIG_MSG

These 7 reasons are the only reasons a gate will be scheduled. There is an additional restriction that scheduling reasons do not nest. This means that a gate can appear at most one time in either the idle queue or the activation queue for each reason.

For example, the first time a gate recieves buffers, the buffers are placed on its delivery list and the gate is scheduled for message delivery. If more buffers arrive before the gate is activated (because there are many other gates ahead of it), those additional buffers do not result in another scheduling. Rather, they are tacked onto the end the existing delivery list.

With signals, the result of no nesting is somewhat different. The first time a gate's timer expires, the gate is scheduled for a SIG_TMO. If the gate doesn't get activated for that SIG_TMO before the timer expires again, the next timeout does not_result in a SIG_TMO. The gate will see only one SIG_TMO although 2 timeout periods have actually occurred.

## Message Delivery

This is the delivery of a new list of buffers for the gate to process.

## SIG_INI

SIG_INI is an initialization signal, usually the result of creating a gate with the G_SIG_INI option of g_req(). This signal can also be sent to an existing gate by using the G_REQ_INI option of g_req() (only recommended if the semantics of sending the signal is "initialize").

## SIG_TMO

A timeout signal sent to a gate when a timer set via g_tmo() has expired. Each gate can only have one timer.

## User Defined Signal

Each gate is allowed a single user defined signal. This is either the signal registered for via g_isr(), the signal being sent by g_sig_gid(), or a SIG_DATA if the gate is the target of a g_sig_data(). This was covered in detail in the Inter-Gate Communication chapter.

## Additional Reasons

There are some additional activation reasons which only apply to pended gates. When a gate pends, it does so within the context of GAME. Thus, these signals will never be seen directly by a gate as they are consumed by GAME. They are listed here only for completeness.

### SIG_IDLE

Originally, this signal was used to unpend a gate after it idled itself on the idle queue. More recently it has been used as a generic unpend signal used for such things as unpending a gate which was waiting for a semaphore token. What exactly a SIG_IDLE implies is dependent upon where within GAME a gate pends (since that is where it will resume execution when it unpends).

### SIG_MAP

Used when creating and firing mapping routines.

### SIG_MSG

Used when new buffers are delivered to a pended gate. This is utilized by the messaging system so it can collect acknowledgements or RPC replies within the context of a gate.

## Pending

When a gate pends within GAME, it waits for some event to occur. For example, when a gate does a g_idle(), it is waiting to get a SIG_IDLE in order to continue. But, what happens if that gate receives a message?

In order to be efficient, the message delivery code doesn't look at the state of each gate which is receiving a message to see if it is currently pended. It just schedules a gate for message delivery whenever it starts a new delivery list for a gate. This means that the g_idle() may actually get unpended for reasons other than a SIG_IDLE, and it needs to handle those reasons correctly.

Say we have 3 gates, A, B and C which are being scheduled (we won't worry about why B and C are on the queues). The chart below shows the state of the activation queue and the idle queue at a particular point in time. The gate at the top of the activation queue is the one which is currently running.

```
                 Activation queue            Idle queue

                   Gate    Reason          Gate    Reason

                  ------  --------        ------  --------

Running gate -->   A      SIG_INI

                   B      ?

                   C      ?
```

So gate A is running its SIG_INI. For whatever reason this takes a long time, so A needs to timeslice itself by calling g_idle(). The g_idle() call results in gate A being placed on the Idle queue for delivery of a SIG_IDLE after the system goes idle. Once this is done, the gate pends allowing the next gate to run:

```
                          Activation queue              Idle queue
                              Gate    Reason             Gate    Reason
                             ------  --------           ------  --------
Running gate -->  B      ?                         A    SIG_IDLE
                  C   ?
```

Now B runs. It sends a message to A. This causes A to be scheduled for a message delivery. Once B completes, C will run and the queues look like this:

```
                          Activation queue              Idle queue
                              Gate    Reason             Gate    Reason
Running gate -->  C      ?                         A    SIG_IDLE
                  A      message
```

Notice that the activation for A's message is on the activation queue. This means that as gates send buffers to other gates on the same slot, the activation queue never goes empty (Note that reliable messaging to the same slot will idle the sender. This is covered later). This is intentional since this is exactly what happens when we're forwarding traffic - and we want that to go fast.

Once C completes the queue looks like this:

```
                          Activation queue              Idle queue
                              Gate    Reason             Gate    Reason
Running gate -->  A      message                   A    SIG_IDLE
```

A now executes. Since it was pended, it unpends at that same point in the code. In this case, that is in the g_idle() syscall. The g_idle() unpends with a message. This isn't what it wants so it remembers that the gate received messages and pends again. The system goes idle since there are no more gates on the activation queue.

Module interrupts are now enabled. Lets assume there are interrupts pending and gate D is a driver which will handle one of those interrupts. The interrupt handler will send a g_sig(SIG_MOD0) causing gate D to be scheduled. Next the idle queue is copied to the activation queue and scheduling begins with gate D running.

```
             Activation queue              Idle queue

                  Gate    Reason            Gate    Reason

Running gate -->  D       SIG_MOD0

                  A       SIG_IDLE
```

Gate D runs to completion and then Gate A gets scheulded. Gate A unpends in the g_idle() syscall. It sees a SIG_IDLE, which is what it was waiting for so it can continue on. But, before returning to the application code, it needs to reschedule any activations it saw but didn't want, such as the message. When control returns to the g_idle() caller, the queues look like this:

```
             Activation queue              Idle queue

                  Gate    Reason            Gate    Reason

Running gate -->  A       SIG_IDLE

                  A       message
```

A is running after the g_idle(). Assuming it then returns from its activation routine it will immediately execute again, but this time for the messages it received while idle.

This may seem confusing, but applications normally don't worry about it. From an application point-of-view, it recieved a SIG_INI, idled and then recieved buffers. What applications, especially in the control path, do need to be aware of is the importance of allowing the system to go idle so that new data traffic can be processed (more on this later).

## Forwarding Path Notes

The scheduler may seem somewhat convoluted, but it is important to keep in mind that GAME was designed for the efficient forwarding of data. In the forwarding path, gates do not pend. Forwarding gates typically receive a continuous stream of buffers. Pending would cause buffers to pile up on the gate's delivery list, possibly depleting the buffer free pool on the slot.

The burden of handling pending was moved from scheduling time to unpending time, since we schedule much more than we unpend.

## Mappings

The scheduling of mapping activation routines is somewhat special. Here is the sequence of events:

1. A "target" gate is created or killed, requiring mappings to trigger. This occurs in the context of some gate (e.g., a gate calling g_req(), or GAME's MAPPER gate, which receives updates from other slots). We'll refer to this as the "triggering" gate.

```
                 Activation queue

                  Gate     Reason

                 -------  --------

Running gate -->trigger       ?

                   A          ?
```

2. The triggering gate reschedules itself at the head of the activation queue with a SIG_MAP. This is effectively a "push" onto the queue to continue the triggering gate's execution once the mappings have all run.

```
                        Activation queue

                    Gate      Reason

                   -------  --------

Running gate -->trigger      ?

                   trigger  SIG_MAP

                      A          ?
```

3. The triggering gate creates a gate (which, through some trickery, becomes a child of the MAPPER gate) which runs the map_map() routine and schedules it at the head of the activation queue. Note that many map_map gates can exist at any time. map_map gates are also responsible for the cleanup of dying gates.

```
                        Activation queue

                    Gate      Reason

                   -------  --------

Running gate -->trigger      ?

                   map_map  SIG_MAP

                   trigger  SIG_MAP

                      A          ?
```

4. The triggering gate pends, allowing the map_map gate to run.

```
                        Activation queue

                    Gate      Reason

                   -------  --------

Running gate -->map_map  SIG_MAP

                   trigger  SIG_MAP

                      A          ?
```

5. The map_map gate also reschedules itself at the current head of the activation queue with a SIG_MAP.

```
                         Activation queue

                      Gate      Reason

                     -------   --------

Running gate -->map_map     SIG_MAP

                 map_map     SIG_MAP

                 trigger     SIG_MAP

                    A           ?
```

6. The map_map gate creates the gates that run the actual mapping activation routines. Their activation routine is map_gate_act(), a kernel routine, which will call the user's mapping routine. These mapping gates are scheduled at the head of the queue, before the map_map gate's SIG_MAP position in the queue (see step 5). The scheduling reasons for these gates are also SIG_MAP.

```
                         Activation queue

                      Gate      Reason

                     -------   --------

Running gate -->map_map     SIG_MAP

                 map_act1    SIG_MAP

                 map_act2    SIG_MAP

                 :

                 map_actN    SIG_MAP

                 map_map     SIG_MAP

                 trigger     SIG_MAP

                    A           ?
```

7. The map_map gate pends, allowing the mapping gates to run.

```
                     Activation queue

                   Gate      Reason

                  -------   --------

Running gate -->map_act1  SIG_MAP

                map_act2  SIG_MAP

                :

                map_actN  SIG_MAP

                map_map   SIG_MAP

                trigger   SIG_MAP

                   A         ?
```

8. The scheduler runs the mapping gates. Note that if a mapping causes other gates to die, additional mapping activations get scheduled ahead of those already scheduled. If the mapping gate pends, it is possible for the state of the "target" gate to change. GAME does not initiate the mapping for the new state immediately. It instead stores the new state in the mapping gate's environment. This may happen multiple times during the life of a mapping. When the current mapping activation completes, the map_gate_act() routine checks for this changed state. If any exists, it re-runs the mapping with the first changed state (and repeats this for each stored state). This way, a gate will never miss a state change.

9. Once all the mapping gates have been run, the next thing on the activation queue will be the SIG_MAP activation for the map_map gate. This resumes the map_map gate's execution. If the mapping was for a dying gate, it then cleans up the gate's memory. Finally, the map_map gate kills itself.

```
                         Activation queue
                     Gate      Reason
                    -------   --------
Running gate -->map_map      SIG_MAP
                    trigger   SIG_MAP
                      A          ?
```

10. The next thing on the activation queue will now be the SIG_MAP activation for the triggering gate. This resumes the triggering gate's execution. It finishes the mapping processing (which isn't much) and returns to the application code.

```
                         Activation queue
                     Gate      Reason
                    -------   --------
Running gate -->trigger      SIG_MAP
                      A          ?
```

Note that if a mapping activation routine pends, it's continued execution can be intertwined with the activations of the base gate and other mappings in the context of the base gate. However, these other mappings can never be mappings for the SAME target gate of the still-running mapping.

## Interrupts

The details of low-level interrupt handling are specific to a given hardware platform. GAME isolates these hardware dependencies from the device driver gates and delivers interrupt events via signals.

GAME runs on various hardware platforms such as the ACE, ACE32, AFN, FRE (1, 2, 3), ASN, ARN, AN, and ARE. These hardware platforms are the name given the the processing engine that populates a slot. Each processing engine contains a main microprocessor that executes most of GAME's code and other support hardware such as timer chips, UARTS, TAGS, memory parity logic, and hardware that is used for interslot communication. Each slot also contains hardware devices such as ethernet, fddi, token ring, and synchronous chip sets that interoperate with the microprocessor on that slot.

Most of hardware platforms that GAME runs on are based on the Motorola MC680x0 microprocessor family. For the rest of this section we will talk about how the 68040 microprocessor on a FRE1/2 works.

The 68040 has three main states:

- Halted. - Awaiting a reset signal.
- Running - Executing code normally such as thru a scheduler.
- Exception - Processing an exception such as errors and external interrupts.

Exceptions can be caused by executing an instruction or by external hardware events, such as interrupts, and hardware errors. Exceptions caused by executing an instruction will be detected by the microprocessor. Some exceptions are predefined and some can be user defined. Predefined exeptions include unimplemented instruction, illegal instruction, address error, bus error and divide by zero. Motorola also defines seven prioritized levels for processing interrupt requests.

When the processor is initialized, code populates an exception vector table with addresses of routines that will run when a particular exception occurs. When an exception occurs that is not due to an interrupt request, the current running code will be preempted after it finishes its current instruction and the routine for that particular exception will execute. All of the interrupt levels, other than level 7, can be disabled. Disabling a level also disables all levels below. If an interrupt request occurs for a level that is enabled, the current running code will be preempted once the current instruction has finished. The interrupt service routine (ISR) for this interrupt level will run. If an interrupt request occurs for a level that is disabled, the exception does not occur until the level is enabled.

The FRE hardware was designed so that only two of the seven interrupt levels are used (levels 3 and 4). The initialization code populates the interrupt vector table in the following manner:

- The entry for the level 4 interrupt will contain the address of g_isr_4().
- The entry for the level 3 interrupt will contain the address of g_isr_3().
- The address of exception vector #2 will will contain the address of BusError().
- All other entries contain the address of except_entry() (there are a couple of minor exceptions).

Interrupt level 4 is always enabled. Because of this, g_isr_4() can preempt any current running code. The watchdog timer, TAGS, memory parity errors and the NMI button result in a level 4 interrupt request. Interrupt level 3 is enabled when the scheduler is idle and in between gates, but disabled while a gate is executing. This means that g_isr_3() does not always service interrupts immediately and it does not preempt gates. The RTC used by the periodic timer, the backbone, UARTs for the TI console, and the link modules, which contain devices such as the ethernet chip, all assert level 3 interrupts.

When a level 3 interrupt occurs, g_isr_3() examines the pending interrupts register and masks out interrupts that are not to be handled at the current time. This is currently used to postpone the processing of link-module interrupts when the scheduler is not idle (i.e., between gate schedulings). For each pending interrupt remaining, a g_isr() call is made to schedule the appropriate signal. As discussed in the Inter-Gate Communications section, the gate that handles the signal indicates whether it should be scheduled at the head of the activation queue (G_ISR_SIG) or the end (G_BASE_SIG).

## ARE/FRE-3 Interrupts

The PowerPC used by on the ARE and FRE-3 has a much different interrupt setup than the 68k does. The PowerPC only has a single interrupt feeding into it. There are no levels explicitely supported by the PowerPC itself.

In order to make the PPC function more like the 68k, external interrupt support HW was added. This support consists of 3 registers; IPEND, IPOL and IENABLE. All of these registers are 32 bits wide. Each interrupt source occupies one of these 32 bit positions and that position is used consistently in all registers. For example, bit 0 of all of these registers deals with the timer interrupt.

These 3 registers get combined in the following manner to generate the single interrupt which feeds into the PowerPC.

1. The interrupt pending register (IPEND) contains the interrupt state of all hardware sources.
2. The IPEND is exclusive-or'd with the interrupt polarity (IPOL)register to flip any active low interrupts to active high ones.
3. The result of this xor is anded with the interrupt enable (IENABLE) register.

4. If any bits are then set, an interrupt is sent to the processor.

Both the IPOL and IENABLE registers are software programmable. As a practical matter, the IPOL is written to once and then ignored (after all, the interrupt sources are either active high or low but not both).

The various levels found on the 68k platforms are implemented by changing the value contained in the IENABLE register. The kernel maintains a set of 3 values which are written to the IENABLE at the appropriate times.

The lowest is the watchdog level. This contains all the interrupt sources which should always be enabled. This includes such things as the watchdog interrupt as well as error interrupts like for tags. Above this are the interrupts which are enabled between gates such as timers and uarts. Finally, the highest level are those interrupts which are enabled only when we go idle such as the module interrupts. Each level contains the level below it thus the watchdog level interrups are indeed always enabled.

When an interrupt occurs, the contents of the IPEND are anded with the current IENABLE and only those interrupt are the ones which get processed.

## CPU Hogging

Since GAME uses a non-preemptive scheduler, it is very easy for a single gate to disrupt an entire slot, or even an entire box, by tying up the CPU for more than a few milliseconds at a time. When a gate or collection of gates "hog" the CPU, the scheduler may not go idle soon enough to handle the link-module interrupts. In this case, packets are dropped, and, for some reason, this makes customers unhappy.

Unfortunately, the onus is put on the application programmer to make sure their gates are well-behaved. Therefore, an analysis of each gate's execution time has to be done to ensure that the CPU is surrendered often enough.

## g_idle Call

g_idle() is the most common vehicle used to ensure a gate is not a CPU hog. Normally, g_idle(G_IDLE_POLL) is placed in an iteration loop. One example is RIP receive processing. g_idle() is used so that the CPU intensive operation of adding / deleting / updating networks does not result in drivers dropping frames.

```
void g_idle (u_int32 flag)
```

"flag" can be one of the following:

- G_IDLE_POLL - place current gate on idle queue. The gate gets rescheduled after the next time the scheduler goes idle.
- G_IDLE_CHECK - check to see if the backbone or drivers need servicing or if the watchdog count is nearing expiration. If TRUE then g_idle() acts as if G_IDLE_POLL was used as a flag. Otherwise, g_idle() returns allowing the gate to continue executing.
- flag can also take the value G_IDLE_TAIL. In older versions of GAME, this would schedule the gate at the end of the activation queue. This feature was removed because it allowed a gate to hold out module interrupts for too long. G_IDLE_TAIL now equals G_IDLE_POLL.

```
Example:
{
FOR EACH NETWORK UPDATE
{
/*
*  Process update.
*/

g_idle(G_IDLE_POLL);
}
}
```

Note that if other gate activations can access and/or modify data used by this gate (e.g., mappings or other gates in the hierarchy), the gate should ensure that the data is in a state that allows access/modification when idling (or it has to protect the data via semaphores).

Other function calls also give up the CPU. However, it is possible for the gate to regain the CPU before the slot has gone idle.

## g_fwd(), g_rpc(), g_reply() Calls

These calls all use an internal GAME function called msg_fwd(). For delivery of a message to the local slot, an explicit g_idle() call is made to allow module interrupts to run. For remote delivery, however, the gate is only pended until an ACK buffer is received from the remote slot. If the local slot is busy enough and the remote slot quickly sends the ACK, there is a chance of receiving the ACK before the local activation queue goes empty. This chance is much lower with g_rpc(), which requires a g_reply() buffer from a gate on the remote slot before the local gate unpends.

For purposes of application writing, assume that these calls will allow module interrupt service. If the frequency of the "exceptional cases" becomes a problem, the functions can be changed to do explicit g_idle() calls.

## g_delay() Call

For any parameter values greater than 16 ticks, it is fairly certain that the slot will go idle before the unpend timer expires. For a delay of 16 or smaller, there is a very slight chance of servicing the timer interrupt (between gate activations) that will unpend the gate. For purposes of application writing, assume that this call will allow module interrupt service.

## g_sema_get(), g_balloc() Calls

These calls give up the CPU only if the requested resource is not available. An application should not rely on these to perform time-slicing.

In addition, most MIB interface calls use g_rpc() to a local gate. Therefore, these MIB calls result in module interrupt service.

# Chapter 8 Watchdog

## Overview

GAME implements a simple non-preemptive, FIFO scheduler where a gate runs to completion unless it voluntarily gives up execution by calling a kernel system call that pends the gate. The kernel system calls that pend are: g_fwd(), g_rpc(), g_reply(), g_delay(), g_sema_get() (sometimes), g_balloc() (sometimes), and g_idle(). (See the Scheduler section for a discussion of CPU Hogging).

The mib interface uses many of these pending functions and many engineers have not taken this into account in their original designs.

Within the forwarding path, the currently executing gate runs its action routine to completion quickly. Non-forwarding path gates do one of the following:

1. run the current action routine to completion quickly.
2. pend themselves one or more times before completing the current action routine.
3. call g_idle() one or more times before completing the current action routine. This is a crude form of time slicing.

## Question and Answer

1. "What happens if the current executing gate is stuck in an infinite loop or appears to be in an infinite loop (it will eventually finish)"?

   The slot would hang, unless GAME's fault management system could detect this condition.

2. "Can GAME's fault management recovery system detect and correct this condition"?

   This condition can be detected by another piece of hardware other than the microprocessor.

3. "How can the fault management code execute if the scheduler is currently running"?

   Microprocessors can execute code as exceptions (interrupts), with the exceptions preempting the normal running scheduled code. So the basis for GAME's watchdog mechanism is to have a piece of hardware, other than the microprocessor, watch over the microprocessor for the purpose of detecting and correcting a hang-like condition.

## How the Watchdog Works On A FRE

If you are not familiar with how interrupts work on a FRE, review the "Interrupts" portion of the Scheduler chapter.

There is a timer chip on the FRE that expires once every second. When this timer expires, a level 4 interrupt is asserted and g_isr_4() will execute. When g_isr_4() executes, it looks at certain hardware registers located on the FRE to determine which hardware device requested the interrupt. If it is determined that the watchdog timer was the reason for asserting the interrupt, then a bit within a register on the FRE is cleared telling the FRE that we are servicing the watchdog timer. If this bit is not cleared within one watchdog timer period, then the FRE does a hardware reset (this is a "hardware watchdog").

When the scheduler idles, the watchdog detection code is disabled. When the scheduler goes from idle to non idle (g_isr_3() schedules a gate), the watchdog detection code is enabled. If g_isr_4() sees that the watchdog is disabled, then g_isr_4() just exits. Otherwise, g_isr_4() executes tmo_wdog().

tmo_wdog() checks to see if the current running gate is the same gate and same invocation of the gate as the last time that tmo_wdog() was run one second ago. If the gates differ, a limit count is set to 3, information that distinguishes this gate invocation is saved, and tmo_wdog() returns. If the gates are the same then the counter is decremented. If the counter reaches zero the slot is restarted (this is a "software watchdog"). Otherwise, tmo_wdog() returns. This means that a gate can run between 3 and 4 seconds before the slot is reset due to a watchdog.

If a gate runs more than a couple of milliseconds then something is drastically wrong. g_idle() calls should be placed into the code at determined points to allow servicing of link module interrupts.

## Platform Differences

The idea for having a watchdog timer was introduced with the FRE. The FRE, FRE2, and ASN basically work in the same manner, due to the common architecture.

The ACE25, ACE32, and AFN predate the FRE. No watchdog timer was added to the processor. Because of this, no watchdog support exists on the ACE25 or AFN. The ACE32 does implement watchdog support, but in a way very different than any other platform. The ACE32 contains two microprocessors: one for GAME processing and one for interslot communication (DMAP) (this is also true for the ACE25). The ACE exception vector table also contains routines for servicing level 7, level 6, and level 5 interrupts, with level 6 handling cascade interrupts. The DMAP processor runs code separate from GAME. When it enters its timer_isr() function, it determines whether or not the main microprocessor is hung. If the DMAP determines that the main microprocessor is hung, it creates a Late Bus Error that will result in the main microprocessor's level 6 ISR executing. Even though the ACE32 and ACE25 have some common architecture, the ACE25 could not reliably use the SYSFAIL signal to achieve like results.

The AN and the ARN both have MC68360 (QUICC) chips that internally contain a lot of programable hardware support, including timers (note that the ARN also contains a 68040 for processing). The exception vector tables on the AN and ARN are similar to each other, but differ from the FRE and ACE platforms. This exeception vector table contains a number of hardware vectored interrupts that contain their own entries in the table and are not part of the 7 prioritized interrupt levels. TIMER 1's vector entry has the address of rtc_isr_4() and is programmed for interrupt level 4. rtc_isr_4() will run every second and, unlike g_isr_4(), this routine exists only for watchdog support and executes tmo_wdog(). The MC68360's internal watchdog is not used.

The ARE uses two power pc processors, both of which run (SMP) GAME. interrupt_handler() is the main interrupt handler. When the watchdog timer expires, call_ihandler() is executed to determine which of the two processors will run the tmo_exp() routine.

# Chapter 9
# Semaphores

## Overview

Game implements a fairly straightforward semaphore capability with some additional requirements due to GAME's high-availability nature.

A semaphore is used to control access to a critical resource. This may be a shared data structure or a piece of hardware. Another use is to limit the number of instances of a certain task being performed.

Each semaphore has a number of "tokens" associated with it. Each token allows one gate to enter the critical section guarded by a semaphore. A "binary semaphore" is simply a semaphore with 1 token.

The number of tokens which a semaphore has is specified when that semaphore is created. It's possible to add or remove tokens from a semaphore while executing.

There are two types of semaphores: well-known and dynamic. These work much like gate IDs. The well-known semaphores are defined at compile time in a header file (include/known_sema.h). The dynamic semaphores are allocated at run time.

When a gate tries to get a token it will pend if one is not available. As tokens are freed, the pending gates will acquire the token and unpend. This is done in a FIFO order.

GAME tracks the ownership and creation of semaphores and tokens like any other resource and will automatically free tokens or remove un-used semaphores when gates die. In order to know which gates are using which semaphores, GAME requires a gate to register for a semaphore before using it.

Semaphores are local to a single slot and cannot be used across slots.

## Well-known vs. Dynamic Semaphores

GAME semaphores come in two flavors; well-known and dynamic. Each of these has the following characteristics:

### Well-known Semaphores

- The ID is defined in include/known_sema.h.
- It may be used by a gate without the burden of passing around a semaphore ID.
- Multiple gates can "create" the well-known semaphore. (If the number of tokens remains the same, the second and subsequent creations just become registrations.

### Dynamic Semaphores

- Created at run time by a gate.
- The semaphore ID is assigned by GAME and must be passed to any other gates which want to use that semaphore.

## Semaphore Creation and Registration

The g_sema() system call is used to create and/or register to use a semaphore.

**Call:**

```
SEMA g_sema (SEMA sema, u_int32 n)
```

**Values:**

"sema" can be:

- G_SEMA_CREATE - create a semaphore
- The handle of a well-known existing semaphore that is to be a multiple creation

"n" can be:

- The  number of tokens for a new semphore
- The new number of tokens to be associated with an existing semaphore
- G_SEMA_REGISTER - to register to use a semaphore

**Return Value:**

The return value is is the semaphore handle to use in subsequent g_sema_XXX() calls. It will be a newly allocated ID if "sema" is G_SEMA_CREATE. Otherwise, it is the same value as the "sema" parameter.

**Considerations:**

Creation of a semaphore automatically registers the creating gate to use that semphore.

If the semaphore already exists and G_SEMA_REGISTER is not specified, g_sema() will change the number of tokens associated with the semaphore to n. Adding tokens will not pend. Decreasing the number of tokens ("n" is less than in the original creation call) may pend because a token first needs to be acquired before the max count can be decremented.

As mentioned above, its possible to do multiple creations of a well-known semaphore. The second and subsequent g_sema() calls simply look like calls to change the number of tokens. If all creators initialize the number to the same value, then no change happens. The net result is that the creator is only registered for the semaphore.

## Getting a Token

The g_sema_get() call is used to obtain a semaphore token. If a token is available in the free token pool of the semaphore, one is removed from the pool and assigned to be owned by the calling gate. If there are no free tokens at the time, the calling gate PENDS until one is freed by some other gate. Note that if the caller owns all tokens, a deadlock is certain!

When there are no free tokens, multiple pending gates are served on first-come first-serve basis. This rule includes callers of g_sema() that are trying to reduce number of tokens.

Death of an owner of the token will cause the token to be returned to the semaphore it came from so that applications need not be concerned with the clean-up.

```
void g_sema_get (sema)
```

"sema" is the semaphore from which a token is desired.

## Returning A Token

The g_sema_put() call frees one semaphore token back to its free pool without any pending. If there are other gates waiting for a token (due to g_sema or g_sema_get calls), the first one is scheduled to run at the end of the current activation queue (just as a message delivery would).

A caller that has no token belonging to that semaphore is terminated.

```
void g_sema_put (SEMA sema)
```

"sema" is the semaphore to which the token is returned.

## Checking A Semaphore's State

It is often helpful to know if there are any tokens available before calling g_sema_get(). This way, a gate can avoid pending if none are available. The g_sema_state() call provides this information.

```
int g_sema_state (SEMA sema)
```

"sema" is the semaphore whose status is desired.

The returned status may be a positive number, zero, or a negative number.

A positive number indicates the number of free tokens available.

Zero and negative indicate lack of tokens and number of already pending waiters (in a sense a negative token count). That is, 0 means there are no tokens left, -2 means there are no tokens and 2 gates are already waiting for a token.

The following is guaranteed not to pend on uni-processor systems.

```
if(g_sema_state(s) > 0)

g_sema_get(s);
```

For SMP, the issue is a bit tricker. Depending upon what gates are using the semaphore and their SMP type, the above may not work. This is the case if all users of the semaphore can't be scheduled to run in parallel. To fix this for SMP would probably require the addition of a new syscall. At the moment its felt that there's not any demand for this functionality.

## Gate Death and Cleanup

Whenever gates die, any tokens it has acquired are returned to the semaphore. This will allow a waiting gates to acquire a token.

When the last user of a semaphore dies, the control block for that semaphore is also freed. This means that the semaphore will have to be re-created before it can be used again.

## Semaphores and Mappings

Mappings don't inherit a creator's registration for a semaphore. If a mapping needs access to a semaphore (even one created by its owner) it must first register to use that semaphore. The reason for this is so that if that mapping completes while still holding onto a token, that token will be returned to the semaphore.

## Are Semaphores Really Needed?

Due to GAME run to completion scheduling and SMP implementation the need for semphores is actually pretty small.

Typically a semaphore would be used to lock a data structure. On a single processor system, as long as a gate doesn't pend during its critical section while modifying the data structure) no other gate will run. So in this situation a semaphore isn't necessary. The important part is that the critical section is non-pending.

On SMP systems, the SMP type of the gates which access the data structure will control if those gates can run in parallel. If gates are configured such that they won't run in parallel, then semaphores aren't needed. Essentially, in this situation, the SMP system looks like a single processor system as far as a gate's ancestory is concerned.

SMP also provides a spin-lock capability. This may be more efficient at protecting a data structure then semaphores as described here. Especially if its unlikely that multiple concurrent accesses to the data structure will happen. However, spin-locks can only be asserted for short periods of time, and a gate cannot pend while it has one.

The primary use of semaphores in GAME applications is to protect data accesses between a gate and its mappings. This is only necessary if a gate's mapping routines perform "real work", as opposed to just sending a signal to the gate's base context. This was discussed in the Mappings section.

# Chapter 10
# Timer and Time of Day Services

## General Overview

This chapter describes two GAME services:

- The GAME Timer Service (AKA timers) provides functionality to allow gates to be periodically scheduled and to sleep.
- The GAME Time of Day Service (AKA time) provides functionality that allows gates to set and get system time.

## Timer Overview

Note that this is the *Timer* overview. There will be a *Time* overview later in this chapter.

Some applications (like those that implement the RIP, SAP, HELLO, LMI, LQR, or BOFL protocols) need to be able to execute the same code on a periodic basis.

Each gate can have one periodic timer. When the timer expires, the gate will be scheduled for a SIG_TMO signal as long as the gate is not already scheduled by a SIG_TMO signal from a previous timer expiration.

The g_tmo() kernel system call is used to start, adjust, and cancel a gate's periodic timer. Once the timer has been started, it will expire every time-out period until the timer is cancelled.

Some applications execute part of their code and then wish to sleep for a period of time before continuing on with the rest of their code. The g_delay() kernel system call or a combination of the g_idle() and g_timer_get() kernel system calls can be used to achieve these results.

## g_tmo() Kernel System Call

Applications start, adjust, and cancel a gate's periodic timer by calling the g_tmo() kernel system call.

**Call:**

```
u_int32 g_tmo (GID gid, u_int32 time)
```

**Values:**

gid" is the gate id whose timer is being manipulated.

"time" is the timeout period [1/1024 seconds]. The value G_CANCEL or 0 cancels the timer.

**Return Value:**

The return value is the timer's previous "time" value.

**Considerations:**

g_tmo() is easy to use but because you can specify any gid, you must be careful or you might start or cancel the wrong gate's timer.

The actual time used for timer expiration is not necessarily what was entered and usually is longer. The FRE, FRE-II, ASN, ACE25, ACE32, AFN, and ARE round this time up to multiples of 16 ms. The AN and the ARN round this time up to multiples of 64ms.

Because timer interrupts are serviced in between gates, some drifting can occur when gates run longer than the platform's Real Time Clock (RTC) interrupt granularity.

GAME's code refers to RTC in two different ways:

1. calendar or wallclock time.
2. A timer that increments in fractions of seconds that asserts an interrupt that GAME can use to internally manage the periodic software timers.

"RTC" within this section refers to #2 as described in the FRE spec (described in the FRE address space document /usr9/harpoon/doc/hardware/freI.txt).

Reliable messages and g_delay() save, steal, and restart a gate's timer (if it exists) which results in a timer expiration delay.

The macros G_TMO_SECONDS and G_TMO_DEFAULT can be used. They are defined in include/kernel.h.

**Example 1**

Start a timer for the current gate.

```
{
 /*
  * some existing application
  */
 ...
 ...
 g_tmo(G_SELF_ID, G_TMO_SECONDS(30));
 ...
}
```

**Example 2:**

Cancel a timer for the current gate.

```
{
 /*
  * some existing application
  */
 ...
 ...
 g_tmo(G_SELF_ID, G_CANCEL);
 ...
}
```

**Example 3:**

Start and Cancel a timer for another gate.

```
{
  /*
   * some existing application
   */
 ...
 ...

 /*
  *  Start up a test gate.
  */
 gid = g_req(G_NEW_ID, TestA, 0, G_SIG_INI);
 /*
  *  Mapping should go here.
  */
 ...
 ...
 /*
  *  Start 1 second timer for gid.
  */
 g_tmo(gid, G_TMO_SECONDS(1));
 ...
```

```
    ...
    /*
     *  Cancel gid's timer.
     */
    g_tmo(gid, G_CANCEL);
    ...
    ...
    ...
}
```

**Example 4:**

What TestA might look like.

```
TestA (env, BufList, sig)
u_int32 *env;
BUF     *BufList;
SIG     sig;
{
  if (BufList)
  {
    BUF     *CurrentBuf;
    BUF     *NextBuf;

    NextBuf = CurrentBuf = BufList;
    while ( NextBuf )
    {
      NextBuf = G_BUF_NEXT(CurrentBuf);
      /*
       *  Process  CurrentBuf.
       */
    }
  }
  else if (sig == SIG_TMO)
  {
     /*
      *  Do Periodic processing.
      */
```

```
        }
        else if (sig == SIG_INI)
        {
          /*
           *  Initialize.
           */
        }
        else
        {
          /*
           *  Other signal processing.
           */
        }
      }
```

NOTE 1. GAME will ensure that the gate will only be activated with either a buffer list OR a signal.

NOTE 2. When code is written to receive both buffers and signals, buffers must be checked for first. SIG_TMO has a value of 0.

**Example 5:**

TestB will, upon initialization, start a timer with a time value of 1 (1/1024) of a second. OldTime will return a value of 0 since no timer is started. The timer will fire 16ms or 64ms later and signal TestB with a SIG_TMO. When TestB handles the signal it will restart the timer with a time value of 2 and OldTime will return a value of 16 or 64. Using a FRE as an example, a time value of 1-16 will mean 16, 17-32 will mean 32, etc. Upon reaching 2048, the timer is cancelled.

```
/* some existing application */
{
 ...
 ...
 /*
  *  Start up a test gate.
  */
 gid = g_req(G_NEW_ID, TestB, 0, G_SIG_INI);
```

```
/*
 *  Mapping should go here.
 */
...
}
TestB (env, BufList, sig)
u_int32 *env;
BUF     *BufList;
SIG     sig;
{
  u_int32  time;
  u_int32  OldTime;
  if (BufList)
  {
    BUF     *head;
    BUF     *tail;
    BUF     *NextBuf;
    /*
     *  Find head and tail.  Then free them buffers.
     */
    NextBuf = head = BufList;
    tail    = NIL(BUF);
    while ( NextBuf )
    {
      tail    = NextBuf;
      NextBuf = G_BUF_NEXT(tail);
    }
    g_bfree(head, tail);
  }
  else if (sig == SIG_TMO)
  {
    if (*env == 2048)
    {
      /*
       * Cancel timer. Technically the g_req() would
cancel the timer.
       */
      OldTime = g_tmo(G_SELF_ID, G_CANCEL);
      g_req(G_SELF_ID, G_REQ_KILL, 0, 0);
    }
    else
    {
      /*
       *  Adjust timer.
       */
      time = *env++;
      OldTime = g_tmo(G_SELF_ID, time);
    }
  }
  else if (sig == SIG_INI)
  {
    u_int32 *NewEnv;
    NewEnv  = (u_int32 *)g_malloc(sizeof(u_int32));
    *NewEnv = 0;
    g_req(G_SELF_ID, TestB, NewEnv, 0);
    /*
     *  Start the timer.
     */
    time = 1;
    OldTime = g_tmo(G_SELF_ID, time);
  }
  else
  {
    /*
     *  Other Signals would be received here.
     */
  }
}
```

## g_delay() Kernel System Call

Applications can execute some code and then sleep before executing some more code by calling the g_delay() kernel system call.

**Call:**

```
void g_delay (u_int32time)
```

**Values:**

"time" is the timeout period [1/1024 seconds].

**Considerations:**

The actual time used for timer expiration is not necessarily what was entered and usually is longer. The FRE, FRE-II, ASN, ACE25, ACE32, AFN, and ARE round this time up to multiples of 16 ms. The AN and the ARN round this time up to multiples of 64ms.

Because timer interrupts are serviced in between gates, some drifting can occur when gates run longer than the platform's RTC interrupt granularity.

The macros G_TMO_SECONDS and G_TMO_DEFAULT can be used. They are defined in include/kernel.h.

**Example 1:**

Using g_delay() to sleep for 1 second.

```
{
 /*
  * some existing application
  */
 ...
 ...
 g_delay(G_TMO_SECONDS(1));
 ...
}
```

**Example 2:**

Using g_delay() to wait for a resource and implement a form of locking.

```
{
 /*
  * some existing application
  */
 while(env->DataBaseFlag)
 {
   g_delay(16);
 }
 /*
  *  Lock data base.
  */
 env->DataBaseFlag = TRUE;
 /*
  *  Modify data base.
  */
 ...
 ...
 /*
  * Unlock data base.
  */
 env->DataBaseFlag = FALSE;
 ...
 ...
}
```

A dual processor like an ARE could have problems with the above if both processors can run the same code.

## Using g_idle() and g_timer_get() For Very Short / Accurate Delays

Sometimes applications need to sleep for time periods much shorter than GAME timer granularities, or need a timer much more accurate than GAME can provide. A combination of g_timer_get() and g_idle() can accomplish this. (The g_timer_get call is described in detail later in this chapter.)

An example of this is in the IPX protocol where the inter-packet delay of RIP and SAP packets should be set to 55 ms. g_delay() would return 6ms.

```
void ipx_delay (u_int32 delay)       /* time to delay in ms */
        {
           TBLOCK time1,time2,time3;
           if (delay < 2000 && delay >= 1)
           {
             g_timer_get (&time1);
             while (1)
             {
               g_idle (G_IDLE_POLL);
               g_timer_get (&time2);
               dsub (&time2, &time1, &time3);
               if ((((time3.sec & 0x000000ff) * 1000) +
(time3.frac / 4294968))
                       >= delay)
                  break;
             }
            }
          }
```

## Grain Tables and tmo_exp()

GAME manages each gate's periodic timer by storing the gate's control block pointer in a time grain table. There are one or more time grain tables (always a power of two - 1,2,4,8,16, ...) with each time grain table containing a maximum of "TMO_GRAIN_SIZE - 1" (currently 15) entries.

Whenever a gate calls g_tmo() to start a timer:

1. The timeout value is rounded up to to be a multiple of the hardware periodic timer's expiration time (16ms or 64ms (AN and ARN)).
2. a repetition count is calculated based upon the timeout value and the number of time grain tables.
3. an entry is added to one of the time grain tables containing the gate's control block pointer. Fields within the gate's control block pointer are filled in to depict the timeout value and the repetition count.

Entries within a grain table are added one after the other with no holes. If an entry is deleted then the grain table is reordered. If a grain table fills then the next grain table is used. If all grain tables are filled then the number of grain tables is doubled and the entries are sorted into the tables based upon the modulus of the number of tables and repetition count of the entry.

Every RTC (Real Time Clock) interrupt (16ms or 64ms) the function tmo_exp() is executed and one time grain is fully inspected. The repetion count for each entry is decremented by the number of time grain tables. If the repetion count is less than or equal to zero then the gate is scheduled for a SIG_TMO signal, unless it has not as of yet serviced a previous SIG_TMO signal. tmo_exp() restarts the timer for expired entries by re-adding an entry to a time grain table.

GAME is architected so that many timers can be handled quickly. The accuracy of the timers is not that precise when small values are used. Since most timers are in increments of seconds, a timer expiring a fraction of a second later usually does not make much of a difference.

NOTE 1. The actual time used for timer expiration is not necessarily what was entered and usually is longer. The FRE, FRE-II, ASN, ACE25, ACE32, AFN, and ARE round this time up to multiples of 16 The AN and the ARN round this time up to multiples of 64ms.

NOTE 2. Because timer interrupts are serviced in between gates, some drifting can occur when gates run longer than the platform's RTC interrupt granularity. A change has been made to tmo_exp() to detect this by using g_timer_get() and to catch up by servicing more than one timer grain table.

NOTE 3. Reliable messages g_rpc() and g_delay() save, steal, and restart a gate's timer (if it exists) which results in a timer expiration delay.

NOTE 4. When the calculated grain table is full and the next free grain table is used, the timer expiration is delayed by one RTC interrupt time for each grain table it must skip over.

NOTE 5. Usually the first expiration of a timer will occur at timeout plus the remainder of the current RTC interrupt.

NOTE 6. The timer code is flawed in that it is possible for a timer to be delayed (by RTC interrupt timer times the number of grain tables) for its first expiration, with all later expirations occuring when expected.

NOTE 7. The timer code is flawed in that it is possible for a timer to expire too soon (not greater than RTC interrupt timer times the number of grain tables) for its first expiration, with all later expirations occuring when expected. This happens mostly when timer grains are full.

## Time Overview

Applications sometimes need to be able to retrieve the system's notion of time, such as calendar and wall clock time. The kernel system call g_tget() is used to get system time. A library function time2wclk() is used to convert the returned system time into wall clock time.

A few chosen applications will need to be able to set the system time. The kernel system call g_tset() is used to set system time. A library function wclkl2time() is used to convert from wall clock time to system time.

A problem can occur if an application uses g_tget() to try to implement periodic processing. If the user sets the system time backwards, the more recent time returned by g_tget() may be less than a previous time. A kernel system call g_timer_get() can be used to ensure than time does not go backwards. g_timer_get() only keeps track of time starting from slot restart and does not include calendar time.

The include file include/wclock.h contains definitions of the structures used by the time functions.

```
/*****************************************************/
/* WALL CLOCK/CALENDAR BLOCK                          */
/
*********************************************************/
typedef struct WCLOCK
 {                        /* all are binary numbers        */
  u_int8  year;              /* 0 - 99                     */
  u_int8  month;             /* 1 - 12                     */
  u_int8  date;              /* 1 - 31                     */
  u_int8  wday;            /* 1 - 7   (1 is Sunday)        */
  u_int8  hour;              /* 0 - 23                     */
  u_int8  minute;            /* 0 - 59                     */
  u_int8  second;            /* 0 - 59                     */
  u_int8  pad;               /* 0 - 59                     */
  u_int16 msec;              /* 0 - 999                    */
  u_int16 usec;              /* 0 - 999                    */
 u_int32 t_zone;          /* 0 - 86400 local time zone     */
                      /* in seconds from date change line  */
                      /* (ex. GMT = 43200, EST = 61200)    */
 } WCLOCK;

/
*********************************************************/
/* ABSOLUTE TIME BLOCK                                */
/
*********************************************************/
typedef struct TBLOCK
 {
   u_int32  sec;        /*  g_tget()      - seconds since
                           *             midnight Jan 1, 1900
                             * g_timer_get() - seconds since
                                    *                 restart
                                         */
   u_int32  frac;                     /* fraction of sec:
bit 31 is 1/2 sec */
 } TBLOCK;
/
```

```
***********************************************************/
/* LOCAL TIME BLOCK                                       */
/
***********************************************************/

typedef struct LOC_TIME
 {
  TBLOCK  time;                /* absolute time          */
   u_int32  zone;                       /* time zone offset
[seconds]       */
  u_int32  flags;                /* flags                 */
 } LOC_TIME;
```

## Getting System Time

Applications get system time by calling the g_tget() kernel system call. Applications can convert system time to wall clock time by calling the time2wclk() library function:

### g_tget() Kernel System Call

**Call:**

```
void g_tget (LOC_TIME *tb)
```

**Values:**

"tb" is a pointer to the location where GAME can write the current system time.

**Example:**

Using g_tget().

```
        {
           LOC_TIME TimeStamp;
           g_tget(&Timestamp);
           /*
           *  Timestamp.time.sec    -  seconds since midnight
Jan 1, 1900.
            *  Timestamp.time.frac   -  fraction of seconds.
            */
        }
```

## time2wclk Library Function

**Call:**

```
WCLOCK *time2wclk (LOC_TIME *tb, WCLOCK *wb)
```

**Values:**

"tb" is apointer to system time to convert.

"wb" is a pointer to the location where GAME can write the wall clock time.

**Return Value:**

The return value equals the "wb" parameter passed in.

**Example:**

Using g_tget() and time2wclk().

```
{
   LOC_TIME TimeStamp;
   WCLOCK   WallClockTime;
   g_tget(&TimeStamp);
   time2wclk(&TimeStamp, &WallClockTime);
   /*
    *  WallClockTime.year
    *  WallClockTime.month
    *  etc.
    */
}
```

## Setting System Time

Applications set system time by calling the g_tget() kernel system call. Applications can convert wall clock time to system time by calling the wclk2time() library function:

### g_tset() Kernel System Call.

**Call:**

```
void g_tset (LOC_TIME *tb)
```

**Values:**

"tb" is a pointer to the structure containing the desired system time.

Only special applications, like the TI date command, should use this function to set system time. All slots calendar times and calendar chips are updated when using this function.

**Example:**

Using g_tset().

```
        {
           LOC_TIME TimeStamp;
           g_tset(&Timestamp);
        }
```

### wclk2time Function Call

**Call:**

```
LOC_TIME *wclk2time (LOC_TIME *tb, WCLOCK *wb)
```

**Values:**

"tb" is a pointer to the location where GAME can write the current system time.

"wb" is a pointer to the wall clock time to convert.

**Example:**

Using g_tset() and wclk2time()

```
        {
           LOC_TIME TimeStamp;
           WCLOCK   WallClockTime;
           wclk2time(&TimeStamp, &WallClockTime);
           g_tset(&TimeStamp);
        }
```

### Retrieve Time Since Slot Restart

Applications can retrieve time since slot restart by calling the g_timer_get() kernel system call.

**Call:**

```
void g_timer_get (TBLOCK *tb)
```

**Values:**

"tb" is a pointer to the location where GAME can write the time.

**Example:**

```
{
  TBLOCK TimeBlock;
  g_timer_get(&TimeBlock);
  /*
 *   TimeBlock.sec     - seconds since slot restart.
   *    TimeBlock.frac     - fractions of seconds
   */
}
```

## Summary of How Timers and Time Are Implemented

GAME runs on various hardware platforms such as the ACE, ACE32, AFN, FRE, FRE-II, ASN, ARN, AN, and ARE. These hardware platforms are the name given the the processing engine that populates a slot. Each processing engine contains a main microprocessor that executes most of GAME's code and other support hardware such as timer chips. The granularity, accuracy, and reliability of the timer services provided by GAME will all be dependent on the hardware that GAME is running on.

The FRE, FRE-II, and ASN are similar and will be referred to as the FRE.

The ACE, ACE32, and AFN are similar and will be referred to as the ACE.

The ARN is similar to the AN and will be covered under the AN.

The ARE is similar to the FRE (somewhat).

The Watchdog Timers are covered in a separate section.

## Periodic Timer

The FRE contains a fixed real time clock (RTC) timer that increments 256 times per second and updates the single byte STAMP register (see tib/tib_pri.h). Every 16 ms (64 times per second), a level 3 interrupt will be asserted, if level 3 interrupts are enabled, or as soon as level 3 interrupts are enabled. The scheduler enables level 3 interrupts by calling g_poll() in between gates and by calling g_wait() when the scheduler is idle. When this RTC level 3 interrupt is asserted, g_isr_3() will execute. When g_isr_3() executes, it looks at certain hardware registers located on the FRE to determine which hardware device requested the interrupt. If it determines that the RTC was the reason for asserting the interrupt, a bit is cleared telling the FRE that we have handled the RTC. The function tmo_exp() is executed.

The ACE contains six programmable timers. One is set to expire after 16 ms (64 times per second). When this interval timer expires, the TMRB1 pending bit is set in the ACE Status Register (ASR; see ace/ace_pri.h). The expiration of TMRB1 will be checked by the scheduler in between gates by executing g_poll() and when the scheduler is idle by executing g_wait(). If TRMB1 is set, clock_isr() is executed. Within clock_isr(), the timer chip is reprogrammed to expire after 16 ms and the function tmo_exp() is executed.

The MC68360 (QUICC) chip used on an AN contains four programmable timers. Timer number 2 is programmed to expire every 64 ms (16 times per second). When timer number 2 expires, a bit is set in the CPM Interrupt Pending Register (CIPR). The CIPR is checked by the scheduler in between gates by executing g_poll() and when the scheduler is idle by executing g_wait(). If timer 2 did expire, the function tmo_exp() is executed.

## Time - g_timer_get()

An application calls the function g_timer_get() to retrieve the amount of time since the slot restarted. g_timer_get() uses information stored in GAME's environment by g_poll(), g_wait(), and ISRs, to derive seconds. g_timer_get() calls g_timer_read() to read a hardware timer to retrieve fractions of seconds.

On the FRE, seconds are incremented in g_isr_4() when servicing the watchdog interrupt. To derive fractions of seconds, the RTC timer is read. The granularity of the RTC timer is 4ms.

On the ACE, seconds are incremented by either g_poll() or g_wait() calling clock_isr(). clock_isr() checks the ASR for the TMRA2 expiring. TMRA2 is set each second by the free running clock. To derive fractions of seconds, the free running clock is read to determine how much time has elapsed since the last second. The granularity of the free running clock is 1/64000 of a second.

On the AN, seconds are incremented in rtc_isr_4() when servicing the watchdog interrupt. Timer number 1 increments 65104 times per second and asserts a level 4 interrupt each second. Fractions of seconds are derived by reading timer 1.

## Time - Calendar Chip

All platforms, with the exceptions of some older ANs, contain a battery backed up calendar chip that also contains 2 KB of non-volatile storage. The AFN, AN, ASN, and ARN contain this chip on the mother board. The ACE and ACE32 that run within the VME chassis contain this chip on the SYSCON board. Each FRE and ARE contain one of these chips.

At some point of time, the TI date command will be used to reset calendar time. When this happens, the calendar chip will be updated. Year, month, date, week day, hour, minute, and second can be set and retrieved from this chip. The chip then independently keeps track of time even if AC power is not applied to the system.

NOTE 1. This chip does not keep fractions of seconds or timezone information.

NOTE 2. The passwords for TI's Manager and User are stored in this chip's non-volatile memory.

NOTE 3. The reason the original AN's did not contain a calendar chip was due to cost cutting procedures. However, this backfired in many ways because it caused heartaches for customers and software engineers.

## Time - g_tget(.

Reading the calendar chip is not cheap. Because of this, the calendar chip is usually only read when a slot restarts and the retrieved calendar time is stored. Whenever g_tget() is called, it calls g_timer_get() and adds the output from g_timer_get() to the stored calendar time.

### Internal Wallclock Service

Keeping accurate wallclock/calendar time on the various platforms running GAME is not trivial. One problem that occurs is that the FRE and ARE platforms contain one calendar chip per slot and time must be synchronized between slots. A second problem occurs in that the calculated calendar time kept by a slot can drift.

To work around the first problem, the LOADER gate creates a master timekeeper soloist gate (GID_MASTERTIMEKEEPER; see include/known_id.h). This gate sends its time to the other slots when they start, and the receiving slots sets their time to the time that was sent by the master slot. The soloist also sends the time to the other slots every 12 hours.

To work around for the second problem, the LOADER gate creates a timekeeper gate per slot (GID_TIMEKEEPER). This gate receives messages from the GID_MASTERTIMERKEEPER gate (FRE and ARE only) and, through a varying periodic timer, it will adjust the wallclock time on its slot if needed. The periodic timer initializes to 1 minute, and then is set to one hour. If, at timer expiration, no adjustment is needed, then the timeout doubles to maximum of 24 hours. If time adjustment is needed, the timeout halfs to a minimum of 1 hour.

## Other

Backbone BOFL's are highly tied into tmo_exp(). They are not covered here.

# Chapter 11
# Miscellaneous Function Calls

## Overview

This chapter covers function calls that were not covered throughly in other sections.

## g_appbase() - Returns Base Load Address

The linker that builds dynamically loadable images (files labelled .exe) does not preserve relocation information. When an image is loaded into memory at run-time by the Dynamic Loader, any pointer or memory reference contained within the image is not adjusted to reflect the actual base address of that image. The result is that after load-time, all pointers that are not relative to the PC location will be relative to location 0, just as they appear in the image before load-time.

An example of this is an array of compile-time initialized literals:

```
char *strings[3] = { "one", "two", "three" };
```

In this case, the array elements will be pointers to the literals which are stored in the literal section of the image, and each pointer will be relative to 0. Another typical example of this can be seen with a Finite State Machine implemented using arrays of function pointers to represent action routines.

To compensate for this, the pointers must be adjusted by the application at run-time. g_appbase() returns the location in memory where the image is loaded (its base address). The returned address must be added to each pointer before it is used.

**Call:**

```
u_int32 *g_appbase (int8 *app_name)
```

**Values:**

"app_name" is a pointer to a string with the application name (as defined in loader/ld_exec_names.c) or NIL to signify the current application.

**Considerations:**

This function is used to find the base address for a code segment (*.exe). The return value is this address.

This function call is not very efficient. It needs to walk a list of all loaded applications, performing a string compare at each entry. For that reason, the caller should perform this call once at initialization and store the results in their local environment.

**Example:**

This example uses the load address to offset an entry in an FSM table.

```
/* fetch the base address of where we're loaded */
u_int32 *appbase = g_appbase("isdn.exe");

:

:

/* adjust the pointer table by our load address */
( (pfi) ( (int) (table->EventFunc) + (int)appbase) ) ();
```

## g_bcfg() - Environment Configuration

**Call:**

```
#include "kernel.h"

void g_bcfg (G_BCFG_BLK *bcfg)
```

**Values:**

"bcfg" is a structure defined in include/kernel.h. It contains a collection of system information maintained by the GAME kernel.

**Considerations:**

Although originally billed as a call to allow applications to influence this information, this call only allows examination of the parameters. The most common use is by device drivers, which check to see if the local buffer size is big enough (if not, they crash or log a message and exit).

## g_buf2mem(), g_mem2buf() - Copy a Buffer's Contents to Memory / Back To a Buffer

These functions are only used in the application-level version of Priority Queueing. They help to implement congestion control for DLS, providing a place to temporarily hold data other than in a buffer. Their use is discouraged unless needed for a similar purpose (i.e., don't use this casually).

### Copy a Buffer's Contents to Memory

**Call:**

```
#include "kernel.h"
u_int32 g_buf2mem (BUF *buf, u_int32 *mem, u_int32 mem_len)
```

**Values:**

"buf" is the pointer to buffer to copy to memory.

"mem" is the pointer to memory where buffer is to be copied. "mem" must be word aligned.

"mem_len" is the number of bytes available after "mem" to save the buffer. The minimum this may be is:

G_BUF_END(buf) - G_BUF_START(buf) + G_BUF2MEM_PAD

**Return Value:**

The return value is the number of bytes actually used to save the buffer.

**Considerations:**

Applications must not modify the saved buffer image. The saved image includes the BUF header and all of the data between the start and end offsets.

### Copy Contents of Memory to a Buffer

**Call:**

```
#include "kernel.h"
void g_mem2buf (BUF *buf, u_int32 *mem)
```

**Values:**

"buf" is a pointer to the buffer that will receive the data.

"mem" is a pointer to memory set up by a previous g_buf2mem() call.

**Considerations:**

This call restores the saved buffer image to a buffer. The calling gate will be terminated if the save buffer has been corrupted.

## g_env(), g_env_gid - Returns Environment of a Gate

### g_env() Call

**Call:**

```
u_int32 g_env ()
```

**Considerations:**

This function returns the current environment for the running gate. The return value needs to be cast to whatever the environment represents.

### g_env_gid Call

**Call:**

```
u_int32 g_env_gid (GID gid)
```

**Values:**

"gid" is th Gate ID of the gate whose environment is desired.

**Considerations:**

This utility returns the current value of the environment of a gate on the local slot, given a GID. Using another gate's environment is generally a dangerous thing to do and extra care must be taken. See the Memory section for a discussion of memory sharing between gates.

## g_i_die(), g_u_die() - Commit Suicide / Kill Another Gate

These functions are morbidly referred to as the suicide and murder functions. g_u_die() was created during a debugging session when someone wanted to set a breakpoint when any gate was killed. g_i_die() was created as a shorthand. All that each routine does is call g_req() with the proper parameters.

### g_i_die Call

**Call:**

```
void g_i_die()
```

**Considerations:**

This terminates the calling gate.

### g_u-die Call

**Call:**

```
void g_u_die (GID gid, void (*act) (void *, BUF *, u_int32),
void *env, SIG sig)
```

**Considerations:**

The parameters match exactly what is passed to g_req(). However, the only parameter actually needed (or used) is "gid".

## g_load_archive() - Archive Loading

**Call:**

```
u_int32 *g_load_archive (char *archive_name)
```

**Values:**

"archive_name" is the name of the archive in the boot image (e.g., "atmc.exe", "dict.str")

**Return Value:**

The return value is the address where archive has been loaded. This is an memory segment which is owned by the calling gate. Zero is returned if there were any errors.

**Considerations:**

This call allows an application to load an archive segment from the boot image. The archive segment is placed into memory which is owned by the calling gate and thus may be freed when the application is finished with it (g_mfree). The call handles retrieval of local or remote archives and will also take care of image decompression.

Only the body of the archive is returned by g_load_archive(). This implies that if the caller needs to know additional information about the data (i.e. its size) there needs to be an application specific header within the body.

Since the memory is owned by the caller and may be freed at any time, it is up to the caller to perform any caching which may be required for performance reasons. Every call to g_load_archive() will result in the boot media being read.

It is recommended that a new extension be created for different archive types. This will serve to keep it clear to us and to customers what type of data is contained in each segment. Perhaps ".MIC" for microcode?

The archive which is loaded by g_load_archive() should be created by the ldgen_compress utility. This utility has the following command line arguments:

```
%ldgen_compress input-file output-file
```

This utility takes the input-file, compresses it and generates an archive header for it. No special format is required of the input-file. The filename stamped in the archive header is the same as output-file. After a call to g_load_archive(), the caller will have an exact duplicate of input-file.

## g_memop() - Special Memory Operation

The g_memop() system call is used to perform a memory operation that may fail (e.g., bus error). The role of this syscall is to ensure that if the operation does fail, it does so in a silent manner so that the caller doesn't get killed due to a bus error. One use of is to probe a memory location to find out whether or not a piece of hardware was installed.

Currently, this silent failure is only implemented on the FRE and ARE. The call can still be made on other platforms, but no protection is provided.

**Call:**

```
#include "kernel.h"

u_int32 g_memop (u_int32 type, u_int32 size, void *addr, void
*data)
```

Values:

"type" is the type of operation. Select one of the following:

- G_MEMOP_RD: perform a read
- G_MEMOP_WR: perform a write

"size" size of the data to read/write. Select one of the following:

- G_MEMOP_8: 8 bits
- G_MEMOP_16: 16 bits
- G_MEMOP_32: 32 bits

"addr" is the address to read or write.

**Return Value:**

The return value is TRUE if the operation failed and FALSE if it succeeded.

**Considerations:**

The size of the data parameter must match the size of the operation. Failure to do so could result in some unexpected return values (i.e., if data is a u_int32 and the op is a byte, that byte will get loaded into the _top_ of the u_int32).

## g_myid() - Returns Caller Gate ID

**Call:**

```
GID g_myid ()
```

**Considerations:**

This function returns the gate ID of the running gate. If this is called in a mapping context, the GID of the base gate is returned, not the GID of the temporary mapping gate.

## g_platform() - Gets Platform Type

**Call:**

```
#include "platform.h"

u_int32 g_platform()
```

**Return Values:**

This routine returns the platform type on which GAME is running. The values are defined in include/platform.h. The values as of this writing are:

| | |
|---|---|
| `PLATFORM_UNKNOWN` | `No clue...` |
| `PLATFORM_SIM` | `Simulator` |
| `PLATFORM_FRE` | `FRE-I` |
| `PLATFORM_FRE2` | `FRE-II` |
| `PLATFORM_ACE` | `ACE (VME hardware)` |
| `PLATFORM_ACE32` | `68030 ACE` |
| `PLATFORM_FNS` | `AFN (68030 ACE, single-board)` |
| `PLATFORM_IN` | `AFN special (single-board plugs into many vendor's hubs)` |
| `PLATFORM_PIR` | `AN (Piranha, QUICC-based)` |
| `PLATFORM_CUDA` | `ASN (Barracuda)` |
| `PLATFORM_BF` | `ARE (Bluefish)` |
| `PLATFORM_BF_5000` | `5000 (Blackfish)` |
| `PLATFORM_NEPT` | `ARN (Neptune) [next-gen AN, 040-based]` |

## g_reset() - Restarts Slot(s)

**Call:**

```
void g_reset (GH gh)
```

**Values:**

"gh" is a bit-map of slots to reset, in gate-handle format.

**Considerations:**

This function call will cause each of the indicated slots to reset (i.e., restart GAME). The only application that has a legitimate reason to call this is a management application (e.g., TI, BCC).

## g_slot() - Returns Caller Slot Number

**Call:**

```
u_int32 g_slot ()
```

**Considerations:**

This function returns the local slot number.

## g_src() - Retrieves Source of Reliable Message

**Call:**

```
GH g_src (BUF *buf)
```

**Values:**

"buf" is a pointer to the buffer to examine.

**Return Values:**

When "buf" points to a buffer that was received via a reliable transport primitive, this function returns the gate handle of the sending gate instance.

If "buf" points to a buffer that has not yet been delivered or was received via a non-reliable transport primitive, FINGER (0) is returned.

## g_stk() - Saves Current Stack in System Log

**Call:**

```
#include "kernel.h"
void g_stk (u_int32 level, u_int32 opt, TBLOCK *time)
```

**Values:**

"level" is the maximum number of stack frames logged, but fewer may be saved if stack is not deep enough.

"opt" is the dumping option, as follows:

- G_STK_DBG: Print saved events on the debug port.
- G_STK_GAME_STK: dumps GAME stack that is always linked below gate stack (by default, only gate stack is dumped)

"time" is optional. If not NIL, the TBLOCK referenced by time will be used to time-stamp all stack dump entries (by default, current time is used).

**Considerations:**

This utility generates several entries in the system log based on the current stack. All stack un-roll events are "TRACE" level and carry the same time stamp for easy spotting. The same stack dump utility is also used by the CRASH macro and other fatal exception handlers.

## get_unqid() - Get a Unique ID

**Call:**

```
u_int32 get_unqid (u_int32 bits)
```

**Values:**

"bits" is the size in bits of unique ID to return. Must be between 21 and 24.

**Return Value:**

This call returns an ID which is unique across all Bay systems. The number is usually related to a serial number. If a unique number cannot be obtained, zero is returned.

# Chapter 12
# Fault Management

## Types of Faults and System Reactions

### Hardware Reset

The following faults cause a slot to do a hardware reset:

- Hardware watchdog
- NMI button (pressed for greater than one second)

Hardware reset implies that the slot goes through the cold start process (diags, boot, GAME). The diagnostics run a full set of tests upon cold start, one of which is a DRAM memory test. This test wipes out the system log, so when you come back from this type of crash, there is nothing left in the log, making it particularly tough to debug.

**Debug Hint:**

If you hit the NMI button for less than one second during diagnostics, it interrupts the current test and gets you to the diagnostics prompt (you obviously have to have something plugged into the diag port to see this). If you hit it _before_ the DRAM memory test is run and then type "boot", the log will remain intact. It can then be viewed when GAME comes up.

Note that a short (less than 1 second) push of the NMI button while GAME is running will not cause a hardware reset. As indicated above, this gets you to the diagnostics prompt

### GAME Reboot

The following faults cause GAME to "restart" or "reboot":

- software watchdog
- memory parity error
- tag violation

Here, "restart" or "reboot" means that the bootstrap is executed, which re-loads GAME (because it may have been code space that was corrupted). Diagnostics are not run, so the log is preserved in this case.

### Gate Termination or GAME Reboot

The action taken for the following errors depends on whether a gate is executing or GAME is running (e.g., in the scheduler). In the former case, only the offending gate is terminated. Otherwise, GAME restarts.

- processor error (illegal instruction, divide by 0...)
- illegal memory reference
- GAME detected error (e.g., bad parameters to a function call)
- VBM error (PPC only)
- Page fault (PPC only)

## "Problem" Gates

If an application has a persistent bug that causes its gate or gates to repeatedly crash, GAME detects this and takes actions to protect the other applications on the slot.

GAME keeps track of gate crashes in three timescales. If a gate dies too many times within a time period, GAME will not restart the gate. It instead terminates the parent (If you can't control your children, GAME comes after you.) The timescales and the allowable number of crashes are:

```
                time       crashes allowed
               ------      ---------------
short term      1 sec            1
medium term     1 min            5
long term       30 min           10
```

GAME also keeps track of crashes on a subsystem basis. Each time the parent gate of a subsystem dies (the gate started by the loader), or when game has to kill a gate's parent for violating one of the crash limits discussed above, GAME records a subsystem failure. GAME compares the number of these failures against limits for three timescales. Whenever the number of failures exceed the limit, the subsystem is terminated and not restarted. The timescales and the allowable number of failures are:

| | time | failures allowed |
|---|---|---|
| short term | 2 sec | 2 |
| medium term | 4 min | 5 |
| long term | 1 hour | 10 |

There are a couple of special cases:

- If the MIB subsystem dies, the entire box immediately restarts. The life of all other subsystems depend on a live and healthy MIB.
- If the DP service exceeds the number of failures allowed for a timescale, the entire box restarts. No packets can be forwarded without DP, so there is no value in keeping everything else alive.

The hope here is that if a gate constantly crashes, killing its parent may remove the reason (e.g., corrupt data in the parent's environment) that is causing the crash. If that doesn't work, the subsystem is eventually shut down.

Obviously, information is put into the log when any of this happens.

# Chapter 13
# System Event Log

## Overview

Event logging can be used for debugging and network management. Each slot that contains a processor (ACE, FRE, etc.) maintains its own fixed size event log that is located at a fixed location within volatile memory (DRAM). Applications only write to the physical log located on the slot the application is executing on. This log will survive system reboots, software restarts, and crashes unless the hardware is (re)initialized. The hardware is initialized during power up, hot swaps, diagnostics, hardware reset, and certain hardware specific failures (on a FRE, the ISR handing the Watchdog timer must clear the Watchdog pending bit before the next Watchdog timer interval or else the FRE hardware will reset).

Events that are written into the log vary in size. Most events that contain strings have a defined code that is stored in the log entry instead of the string. This practice allows for more events to be written into the fixed size log. When the log is viewed, the code is replaced by a string from other entities such as string services or Site Manager. Strings still can be written directly into the log, but this practice limits the number of events that can be stored in the log. When the log is full and a new event needs to be added, the oldest entry or entries are removed and replaced with the most recent entry.

It is NOT a good practice to log a message by using a sprintf to format a string and use a "generic" EDL event code. Unless you are adding a message that will be removed when a defined problem is fixed, modify the EDL file to add a new EDL event code. "generic" EDL event codes waste too much log space.

## Log Entry Format

```
+-------+--------+--------+--------+
| Size  | Type   | Entity | Event  |      4 bytes
+----------------------------------+
|                                  |      8 bytes
|        Time Stamp                |
+----------------------------------+
|   Sequence Number          |Slot|      4 bytes
+----------------------------------+
|                                  |
x            Data                 x     0 - 251 long words
|                                  |
 ---------------------------------
```

Size - Size of log entry in long words (4 - 255).

Type - Log entry type. Will be one of the following values:

- 1 - DEBUG message
- 2 - INFO message
- 4 - WARNING message
- 8 - FAULT message
- 16 (0x10) - TRACE message

Entity - Who logged the event (see include/edl_types.h).

- 0 - TI_EDL
- 1 - LB_EDL
- 2 - IP_EDL
- 3 - SNMP_EDL
- ...

- 12 - OSPF_EDL
- ....
- 30 - IPX_EDL
- ...

Event - Log message number within entity ( see edl/*.edl; TI.edl, LB.edl, IP.edl, etc ).

Time Stamp - Time that event was logged.

- First 4 bytes - number of seconds since January 1, 1900.
- Second 4 bytes - fraction of seconds. (The number of bits used is hardware specific and left justified.) See Timer chapter.

Sequence Number - Sequence number of event on Slot.

Slot - Slot number event occured on.

## Quick Example of EDL

Each numeric event code is defined using the "Event Definition Language" and a preprocessor tool. Entity specific log messages are created by adding the entity to "include/edl_types.h" and a corresponding "<entity>.edl" file to the "edl" directory. For example, when NetBios over IP was added to the system

```
#define NBIP_EDL        77
```

was added to the file "include/edl_types.h" and the file "edl/NBIP.edl" was created. The contents of "NBIP.edl" is:

```
-----------------------------------------------------
/* @(#)WSCCS c/edl-NBIP.edl 1.1 6/27/94 */

RECORD NBIP_EDL

NBIP_CRASH          FAULT_MSG        "System error, service
attempting restart."


NBIP_BAD_PKT       WARNING_MSG    "invalid NetBIOS over IP
packet received"

NBIP_BAD_NAME      WARNING_MSG    "invalid NetBIOS over IP
name"

NBIP_INIT           INFO_MSG        "Service
initializing."
NBIP_IF_UP          INFO_MSG        "Interface
%d.%d.%d.%d up."
NBIP_IF_DOWN        INFO_MSG        "Interface
%d.%d.%d.%d down."
NBIP_TERM           INFO_MSG        "Service
terminating."

NBIP_UNK_PKT        DEBUG_MSG       "unknown
protocol received"
NBIP_CACHE_INIT     DEBUG_MSG       "initializing
NetBIOS name cache"
NBIP_CACHE_DOWN     DEBUG_MSG       "killing
NetBIOS name cache"
NBIP_CACHE_FULL     DEBUG_MSG       "NetBIOS name
cache is full"
NBIP_AGE_NAME       DEBUG_MSG       "aging from
NetBIOS name cache"
NBIP_GENERIC_DBG     DEBUG_MSG       "%s"

END_REC NBIP_EDL
-----------------------------------------------------
```

From "NBIP.edl" and "edl_types.h", the preprocessor tool will create the file NBIP_edl.h. The contents of "NBIP_edl.h" is

```
-----------------------------------------------------
ifndef NBIP_EDL_H

#define NBIP_EDL_H


#include "edl_types.h"


#define NBIP_CRASH      (u_int32)(FAULT_MSG | (NBIP_EDL <<
8) | 1)

#define NBIP_BAD_PKT   (u_int32)(WARNING_MSG | (NBIP_EDL <<
8) | 2)

#define NBIP_BAD_NAME  (u_int32)(WARNING_MSG | (NBIP_EDL <<
8) | 3)

#define NBIP_INIT      (u_int32)(INFO_MSG | (NBIP_EDL << 8)
| 4)

#define NBIP_IF_UP     (u_int32)(INFO_MSG | (NBIP_EDL << 8)
| 5)

#define NBIP_IF_DOWN   (u_int32)(INFO_MSG | (NBIP_EDL << 8)
| 6)

#define NBIP_TERM      (u_int32)(INFO_MSG | (NBIP_EDL << 8)
| 7)

#define NBIP_UNK_PKT    (u_int32)(DEBUG_MSG | (NBIP_EDL <<
8) | 8)

#define NBIP_CACHE_INIT (u_int32)(DEBUG_MSG | (NBIP_EDL <<
8) | 9)

#define NBIP_CACHE_DOWN (u_int32)(DEBUG_MSG | (NBIP_EDL <<
8) | 10)

#define NBIP_CACHE_FULL (u_int32)(DEBUG_MSG | (NBIP_EDL <<
8) | 11)

#define NBIP_AGE_NAME   (u_int32)(DEBUG_MSG | (NBIP_EDL <<
8) | 12)
```

```
#define NBIP_GENERIC_DBG        (u_int32)(DEBUG_MSG | (NBIP_EDL
<< 8) | 13)


#endif /* NBIP_EDL_H */
-----------------------------------------------------------
```

NOTE 1. The preprocessor tool only allows 255 TOTAL messages per entity, not 255 DEBUG messages, 255 WARNING messages etc.

NOTE 2. All new messages MUST be added to the END of the ".edl" file. So if you add a FAULT_MSG:

NBIP_OHNO FAULT_MSG "Oh NO"

you would add this after NBIP_GENERIC_DBG, not NBIP_CRASH. The reason for this is that newer versions of tools that format the log (like Site Manager) would get mixed up when reading a log from an older version of router software.

## g_log() System Call

Applications add entries to the log by calling g_log system call:

**Call:**

```
void g_log (u_int32 code, u_int32[] args)
```

**Values:**

"code" is the numeric event code.

"args" is the variable length array of event arguments.

**Example 1:**

In order to log the message "invalid NetBIOS over IP packet received", the following lines can be added to the appropriate function:

```
          #include "NBIP_edl.h"

          g_log(NBIP_BAD_PKT);
```

Phyically, 16 bytes would be consumed by this log entry.

```
         0x04 04 4d 02      16 bytes (4 long words) WARNING
NBIP  code 2
          0xb6 58 88 d4      12/09/96  13:11:48
          0x80 00 00 00      .5 sec
          0x00 00 12 34      Sequence Number 291  Slot 4
```

**Example 2:**

In order to log the message "this is boring", the following lines can be added to the appropriate function:

```
          #include "NBIP_edl.h"

          char my_msg[80];

          sprintf(my_msg, "this is boring");

          g_log(NBIP_GENERIC_DBG, my_msg);
```

Physically, 32 bytes would be consumed by this log entry.

```
0x08 01 4d 0c     32 bytes (8 long words)  DEBUG  NBIP code 12
          0xb6 58 88 dc      12/09/96  13:11:56
          0x01 00 00 00      4 ms
          0x00 00 12 44      Sequence Number 292 Slot 4
          0x74 68 69 73      t h i s
          0x20 69 73 20      <sp> i s <sp>
          0x62 6f 72 69      b o r i
          0x6e 67 00 00      n g <null> <null>
```

If a new EDL event code was added to display this message, only 4 long words of log space would be consumed instead of 8. Even if the text string length was much larger, only 4 long words would be used instead of a much larger length.

**Example 3:**

In order to log the message "Interface 1.0.0.1 down.", the following lines can be added to the appropriate function:

```
#include "NBIP_edl.h"

u_int32  ip_address;

ip_address = 0x01000001;

g_log(NBIP_IF_DOWN,

       ( (ip_address >> 24) & 0xff),

       ( (ip_address >> 16) & 0xff),

       ( (ip_address >>  8) & 0xff),

       ( ip_address & 0xff) );
```

Physically 32 bytes would be consumed by this log entry.

```
0x08 02 4d 06   32 bytes (8 long words)  INFO  NBIP code 6
   0xb6 58 88 e4         12/09/96  13:12:04
   0x02 00 00 00         8 ms
   0x00 00 12 54         Sequence Number 293 Slot 4
   0x00 00 00 01         1
   0x00 00 00 00         0
   0x00 00 00 00         0
   0x00 00 00 01         1
```

## System Event Logger Gate

Applications directly add entries to the log through the kernel system call g_log(). The System Event Logger Gate is a well-known gate that runs on each slot. The primary purpose of this gate is to handle requests for retrieving events from the log so that the log can be viewed or stored.

TI, TI_RUI, TFTP, FTP, and SNMP all communicate with the System Event Logger Gate on one or more slots by using g_rpc(). The gate requesting the log entries will receive replies from one or more slots and sort the log entries received via the timestamp field of each log entry. These gates may also perform filtering so that entries physically contained in the log do not have to be viewed or stored. Filtering can be done by date, time, entity, severity (event type), and code (event code). Slot filtering can also be done, but in this case the g_rpc() just sends the request to one slot.

The complete log cannot usually fit in one g_reply(). Because of this, numerous g_rpc()s will be sent from the requesting application to the System Event Logger Gate. The data portion of the g_rpc() contains a field that is a requested sequence number. The Event Logger Gate returns entries greater than the requested sequence number and not greater than the log's current sequence number. When the requesting gate sends the first g_rpc(), the sequence number is usually set to zero so that every entry starting at the logs lowest sequence number will be returned. The g_reply() will contain a number of log entries and the sequence number that the next g_rpc() should use. The log can also be polled for only new log events by not always using zero as the initial sequence number. This procedure is used by SNMP for traps and optionally can be used by TI's log command.

## How the Log Becomes Useless at Times

The 5 series OS had a small log that was resident only on slot 2. Each entry had a fixed size of about 80 bytes. The system had no fault management and debug log messages did not really exist. The log did not survive reboots, but it could be periodically saved to floppy.

When GAME applications are designed, a number of DEBUG messages are typically added. These DEBUG messages are not documented and, by default, the TI log command filters out DEBUG messages so that they are not seen. Two problems arise from using this procedure. First, customers, and even engineers, can have a hard time figuring out what these debug messages mean (they are often very cryptic). Second, the DEBUG messages still take up log space, so they limit what can be physically placed in the log.

Some applications are much too chatty (they log too much). When it became necessary for routers to scale to a large number of interfaces per slot (precipitated by the release of the MCT1 link module) the log started to wrap frequently during certain critical periods (like boot time) and the useful information in the log was lost.

IPX and some other protocols allow the user to set a filter via the MIB to control which log messages are written into the log, but most applications do not have this functionality.

Another form of log filtering was added to the system for debugging purposes. This log filtering filters out the g_log() kernel system call so that the message is not written physically into the log. This was accomplished by increasing the log header that manages the log to add a bitmask that allows each severity type for each entity to be filtered.

Some important GAME messages cannot be filtered.

**Example 1:**

Exclude

```
$ log -x        /* Exclude all log messages all slots       */

$ log -x -s2     /* Exclude all log messages on slot 2       */

$ log -x -s2 -eLAPB  /* Exclude all LAPB log message on slot
2     */

$ log -x -s2 -eIP -fd    /* Exclude all IP DEBUG messages on
slot 2    */
```

**Example 2:**

Include

```
$ log -i         /* Include all log messages all slots       */

$ log -i -s2     /* Include all log messages on slot 2       */

$ log -i -s2 -eIPX  /* Include all IPX log messages on slot
2     */

$ log -i -s2 -eIP -ffw   /* Include all IP FAULT and WARNING
messages  */

/*               on slot 2                            */
```

NOTE 1. When the log is saved, a template is printed to show how the filters are set.

NOTE 2. The filters are active until they are modified or until the hardware is reset.

NOTE 3. From the TI, "log -z" is used to display the current filter settings.

## Log Crash Points

Sometimes, debugging problems that occur on-site infrequently becomes a long and tedious affair. A crash dump tool was developed for saving a slots complete memory image at the time that an application panics or experiences a system fault.

The Log Crash Points feature was added to the system so that the application would indirectly PANIC upon calling g_log() if the g_log event code matched a predefined filter. Previously, to get the same effect, you'd have to recompile the code with the PANIC added.

**Example 1:**

Set Log Crash Points

```
$ debug slcp 2 NBIP 8  /* Set a log crash point on slot 2 */
/* for NBIP code 8                   */
/* NBIP_UNK_PKT                      */
```

**Example 2:**

Clear Log Crash Points

```
$ debug clcp 2 NBIP 8   /* Clear a log crash point on slot 2 */
/* for NBIP code 8                     */
/* NBIP_UNK_PKT                        */
```

**Example 3:**

List Log Crash Points

```
$ debug llcp 2       /* List log crash points on slot 2   */
0x00004d08 NBIP           Event : 8
```

NOTE 1. The debug system does not have to be loaded to use log crash points.

NOTE 2. Log crash points are one-shots. They are cleared upon taking the PANIC.

NOTE 3. 8 log crash points can be set per slot.

NOTE 4. The interface requires you to have the EDL files handy.

## Choosing The Appropriate Event Severity

The following are the definitions of the severity levels that you can assign to a log event:

- FAULT - Something is about to crash
- WARNING - Recoverable error that should be flagged for the user, i.e. something potentially dangerous occurred, but the box stayed up e.g. link module not verified with diagnostic)
- INFO - Normal operations that user should know about (e.g. Spanning Tree is up)
- TRACE - Events that happened as a result of network activity (e.g. DECnet adjacency up)
- DEBUG - Events that aid in debugging problems.

### FAULT Messages

Every entity must have an event of the following type defined in its edl file :

```
xx_CRASH    FAULT_MSG    "System error, service attempting
restart."
```

where "xx" is the entity string.

When you decide to PANIC for any reason in your code, you must use the macro CRASH(xx_CRASH). This causes the above FAULT event to go into the log immediately before the crash, making it evidently clear which application lost its cookies.

You may choose to log other events before crashing, to aid in debugging. These must be DEBUG events. The only FAULT events in the log should be xx_CRASH events, along with PANICs, bus errors, tag violations, etc.

## WARNING Messages

This is a judgment call. If you detect something bad that doesn't cause a FAULT, but that you feel is important enough to call the user's attention to it, log a WARNING event. Examples from the current revision include duplicate IP address detection, file system corruption, diagnostic failures, unreadable config file, ethernet carrier loss.

## INFO Messages

INFO events should be kept consistent across all applications, meaning that DECnet coming up should look very similar to IPX coming up, CSMACD lines register the same events as FDDI lines, etc. This goes all the way down to exact wording of universal events. Obviously, not every entity in the box fits the mold exactly, but please make an effort to adhere the existing styles.

Another goal is to keep the number of INFO events down to a manageable level.

Guidelines for logging application INFO events:

1. At the beginning of your init strip for your entity, log one of the following events:

```
event_name  INFO    "Protocol initializing."

event_name  INFO    "Service initializing."
```

2. When your entity terminates for any reason (even if it is bouncing right back up again), log one of the following events:

```
event_name  INFO    "Protocol terminating."

event_name  INFO    "Service terminating."
```

3. When your entity comes up on a given circuit, log the following event:

```
name     INFO     "Interface <??> up on circuit <n>."
```

where: ?? is your identifier for the interface on that circuit, (e.g. 192.32.1.56 for IP, NIL (empty string) for LB) and "n" is a %d for circuit number.

4. When your entity goes down on a given circuit, log the following event:

```
name     INFO     "Interface <??> down on circuit <n>."
```

where: ?? is your identifier for the interface on that circuit, (e.g. 192.32.1.56 for IP, NIL (empty string) for LB) and n is a %d for circuit number.

These should be the only INFO events you log. Again, not every application fits the mold exactly, but this is the model.

## DEBUG Messages

There are no guidelines for DEBUG messages. Your DEBUG events are your own, but remember that the memory reserved for logging events is a limited resource. Don't go wild filling up the log with DEBUG events and cause it to wrap, thereby losing potentially important information.

Also, remember that although DEBUG events are not documented, customers can see them. Maintain a professional tone and and provide enough coherent information so that a customer can use the information when talking with customer support (i.e., don't just dump a bunch of hex numbers!).

## Logging Tips & Miscellaneous Info

Physical log sizes:

- FRE, FRE2, ASN are 64k
- ACE25, ACE32, AFN are 64k (Some older revs 32k)
- ARE is 64k
- ARN is 32k
- AN > 2MB DRAM is 32k
- AN 2MB DRAM is 16k

At many sites the log wraps quickly during certain failures. Much of this wrapping is due to applications being too chatty.

Some customers who have free memory have requested that the log size be increased to a size as large as 4 MB.

A common mistake made is to save the log too quickly after a failure. Unless the System Event Logger gate is up, the log cannot be retrieved from that slot.

The wallclock time kept between slots is not totally in sync. When following an event that crosses slots it is possible that for the log too show them out of order absolute time wise.

# Chapter 14
# System Loader

Prior exposure to the Bay development environment, build process, and GAME concepts (gates, mappings, etc.) is helpful.

## Background

GAME and its applications were originally linked as a single slab of code (like the simulator). This became unweildy as more and more software was developed for the router. Therefore, a mechanism was needed to separate applications from the kernel and each other.

The following goals were established:

- mechanism for conditionally (via configuration) loading/ spawning applications
- provide fault isolation/recovery in conjunction with the kernel
- extensible to easily support new kernel elements and applications
- minimize DRAM memory consumption on all slots
- allow for tailored S/W image to reduce file system memory consumption, and only ship the specific software modules which customer ordered
- hooks for releasing software modules independently, if we ever decide to do so

### Linking/Loading Options

A couple of options were considered to meet the above goals:

### Memory Reclamation (5-Series Method)

Image is still linked as a slab, but unconfigured applications would have their code space reclaimed and placed in the dynamic allocation pool at run-time. This option was not chosen.

### Dynamic Loading

Dynamic loading was chosen for the following reasons:

- GAME's dynamic config capability disqualified memory reclaimation because an entity could be loaded at any time.
- targeted a separate-linking approach where the kernel is linked statically (as a slab) and applications (drivers, routers, etc.) are linked as their own executables.
- wanted code to be relocatable so it could run anywhere in memory.
- Oasys compiler supported position-independent code (PIC) where all offsets are calculated relative to the PC.
- access to kernel system calls via jump tables.
- linking loader option was considered, but deemed overkill for an embedded environment due to concerns about increased image size (because of reloc info), performance (depending on implementation), and boot time (re-linking); also, the modified image is not easily servable to neighbors because it's not virgin.
- archive file format holds all the executables in a single file (bn.exe, ace.exe, etc.).

## Loader Operation

The System Loader consists of a Kernel Loader and a Dynamic Loader.

## Kernel Loader

After the bootstrap acquires the kernel image, GAME initializes the hardware and itself, and then starts the Kernel Loader.

The Kernel Loader is really just a gate spawner that works in two phases:

- Phase 1 - "core" kernel services are brought up first (GAME, file system, MIB/Emanate, loader gates, timekeeper). MIB must obtain config and initialize first before any other subsystems can start.
- Phase 2 - system services are then brought up ( DP, event logger, kernel MIBs, etc) and finally the dynamic loader is launched.

One of the gates spawned in phase 2 is an image server gate, which serves the kernel and application images to remote boot clients

## Dynamic Loader

The Dynamic loader is a mechanism for conditionally (via configuration) loading/spawning applications.

The dynamic loader retrieves its configuration records (wfLinkModules wfDrivers, wfProtocols) from the MIB.

Applications are loaded on a per-slot basis, as dictated by the configuration records

The Dynamic loader monitors dynamic changes to the MIB records so it can load or unload applications on demand.

### Acquiring Application Executables

For each application that's configured, the loader spawns a downloader gate which attempts to acquire the application image.

The downloader gate first tries to load the image from a neighbor slot (straight from DRAM) by sending broadcast messages to the image server gate.

To expedite the boot process, each image server can serve multiple downloader clients simultaneously.

If no neighbor slot has the desired image, then loader attempts to get it from the active boot image on the file system (flash on BN).

A file system control gate serializes access to FS to minimize disk thrashing.

Executable files which come from the FS are compressed, so the loader must decompress them. Images obtained from a neighbor slot are already decompressed.

Each image has a compressed & uncompressed checksum that the loader validates.

The dynamic loader supports image revision checking to ensure that the kernel and application images are from the same release. It enforces this check on all 'rel', 'int', 'fix', etc. images; however, it allows anything with a 'dev' stamp to run with anything else so developers can make workspaces and debug in the lab.

On platforms that support TAG protection, the loader sets the code section to read-only to prevent inadvertant corruption. The data section can't be protected because that would require it to be 'uncachable'. This would have a detrimental effect on performance.

## Jump Tables

Kernel system calls and inter-module API calls go through a central kernel dispatch table (the GAME dispatch table). The magic structure 'game_hdr' is the place-holder for the dispatch table.

In the kernel, the 'game_hdr' structure is declared in the game subsystem and linked into the kernel image. Each application which links independently has its own copy of 'game_hdr', which is declared in the subsystem's '<subsys>_hdr.c' file.

The loader plugs the address of GAME's dispatch table pointer into each application's 'game_hdr' structure at load-time.

Each GAME system call is defined as a macro in the include/game.h.

```
    header file:
#define g_req      (GID) (DsP (G_REQ))     /* GID gid, void
(*act) (),
 u_int32 env, u_int32 ini */
where DsP is defined as:
#define DsP( call_num )  (* (game_hdr.dispatch [call_num]))
```

The "call_num" is simply a constant from 1 to G_END_SCALL, which represents each system call's location in the dispatch table.

The following example compilation of a call to g_req shows four args being pushed, the jump table pointer being loaded, and eventually a JSR through the function pointer:

```
 ld_app.c: 168                  /* Log message and start gate */

  ld_app.c: 169                           gid = g_req(gid,
init_act, init_env, signal);

  787  9:00000120  2F04                    MOVE.L  D4,-(SP)

 788                           *               STACK OFFSET 4

 789  9:00000122  2F2E0014              MOVE.L  20(A6),-(SP)

 790                           *               STACK OFFSET 8

 791  9:00000126  2F2E0010              MOVE.L  16(A6),-(SP)

 792                          *               STACK OFFSET 12

  793  9:0000012A  2F02                    MOVE.L  D2,-(SP)

 794                          *               STACK OFFSET 16
```

```
   795  9:0000012C  2053                          MOVE.L  (A3),A0
  796  9:0000012E  20680010                      MOVE.L  16(A0),A0
  797  9:00000132  4E90                              JSR      (A0)
```

The system services (mib, tbl, etc.) and dynamically loaded applications (ip, tcp, etc.) use a second level of indirection through the jump table to accomplish function calls; by convention, sys service calls are defined in <subsys>.h (mib.h, tbl.h, etc.) and app service calls are defined in <subsys>_dsp.h (ip_dsp.h, tcp_dsp.h, etc.). Notice the extra level of indirection required to load the function pointer.

```
      (3 MOVE.L instead of 2):

#define mib_get_new_inst   ( u_int32   ) (AppDsP (MIB_INDEX,
MIB_GET_NEW_INST))
#define AppDsP(index, call_num) (* (((int
((**)()))(game_hdr.dispatch [index])) + call_num))
 ld_get_cfg.c: 79           if (mib_get_new_inst(obj_id,
inst_id);) {

 575  9:00000076  2F0C                          MOVE.L  A4,-(SP)
 576                          *                    STACK OFFSET 4
 577  9:00000078  486EFFBC                         PEA     -68(A6)
 578                          *                    STACK OFFSET 8
 579  9:0000007C  2053                          MOVE.L  (A3),A0
 580  9:0000007E  20680200                    MOVE.L  512(A0),A0
 581  9:00000082  20680014                      MOVE.L  20(A0),A0
  582  9:00000086  4E90                              JSR      (A0)
and for a dynamically loadable application:

#define ip_register          (u_int32) (DynDsP(IP_INDEX,
IP_REGISTER))

#define DynDsP(index, call_num) (* (((int
((**)()))(game_hdr.dispatch [index+G_END_SCALL])) +
call_num))
```

```
tcp_mgr.c: 756                      ret = ip_register (
&twait_env->local_ip, (u_int32)NULL,....

                                                       ...
2483  9:000004E4  48780001                       PEA     $00000001
2484                          *                    STACK OFFSET 40
2485  9:000004E8  2F2A0028                  MOVE.L  40(A2),-(SP)
2486                          *                    STACK OFFSET 44
2487  9:000004EC  42A7                            CLR.L    -(SP)
2488                          *                    STACK OFFSET 48
2489  9:000004EE  486A0020                         PEA      32(A2)
2490                          *                    STACK OFFSET 52
2491  9:000004F2  207B017000000002                     MOVE.L
(game_hdr,PC),A0
2492  9:000004FA  2068027C                   MOVE.L  636(A0),A0
2493  9:000004FE  20680020                    MOVE.L  32(A0),A0
2494  9:00000502  4E90                            JSR      (A0)
```

## API Calls Between Loadable Modules

An application may publish a public jump table (example, TCP).

The loader plugs the app jump table pointer into the appropriate location in the second level dispatch table and "relocates" the pointer address.

Clients which make calls through a dynamically loaded app's jump table must synchronize with that application. Note that the code for the API function may be unloaded at any time by modifying the configuration.

Synchronization is accomplished by mapping the parent gate of the service-providing application.

Ensuring that the mapping routine DOES NOT PEND will leave your code free of race conditions. Note that if your gate is pended inside an API call, and the API owner and its code are unloaded, and your mapping routine pends, your gate may resume execution inside the API code space that has already been unloaded.

A fairly simple, correct mapping routine:

```
void tnc_tcp_mapper(gh, new_gh)
GH      *gh;
GH       new_gh;
{
   TNC_ENV *tnc_env;
   if ( GH_CEASED_LOCAL(*gh, new_gh) ) {
      tnc_env = (TNC_ENV *)g_env();
      tnc_env->state = TNC_EXIT;
      g_log(TNC_TCP_DOWN);
      g_i_die();
   }
   *gh = new_gh;
}
```

## Application Requirements

Application requirements are as follows:

- no global, writable data (.BSS)
  - globals are not very "clean"
  - 5-series code was riddled with bugs that resulted from mis-managed global variables. Globals don't work across slots in a true distributed system. Globals don't work in the current implementation of the multislot GAME simulator.
  - BSS location/size info not carried in the image header
- all code and data "PIC-able"
- do use jump tables to publish APIs

## Application Interfaces

Applications may load an executable module from the boot image archive via the g_load_archive() system call. This is typically used by drivers which must download a coprocessor (cop) image. Note that the caller owns the memory, which it may free at its own discretion.

Applications may also load an 'overlay' version of an executable module. This enables multiple calling gates to share a single copy of loadable code, rather than each gate loading its own version.

## Fault Management

Loader maintains a mapping on each gate it spawns so it can restart any gate that PANICs or crashes.

Two system gates are special because they provide shared memory pointers to their clients: the MIB and DP. If they ever crash, the entire slot restarts because apps are not coded to deal with the loss of these services (stale pointers).

Game/loader maintain a history of each subsystem's crashes, and if the subsystem appears to be 'broken' it will not be restarted. This keeps mis-configured or broken gates from hogging the CPU.

Historical data is maintained for:

- children of subsystem - if a child or multiple children are 'broken', then the subsystem will be restarted.
- subsystem itself - if subsystem is 'broken', then it will not be restarted. 'Broken' is defined by the number of crashes which occur in a given time period (see the Fault Management section)

## Shortcomings

The Loader is restrictive (no .BSS, all code "PIC-able", etc.). At the time, most of our code was built in-house, and the requirements seemed reasonable. Unfortunately, we now port a lot more code 3rd party code, so the requirements have become an impediment.

The simulator was not addressed. Image is tailored via stubs.c file

### BCC Work

The above requirements were deemed too restrictive for an application which is data-driven and is largely composed of 3rd party code.

Added hooks to allow .BSS section.

Relocated non-PICable data structures directly in .DATA section and marked the image as "not servable" to other slots.

Ultimately, they want a more standard-OS approach (i.e. linking loader support). They're working on a true run-time linking loader.

## Process Issues

### Builds

These issues may go away or be modified, due to the conversion to clearcase.

The kernel must be re-linked in the global build directories (buildtib, buildace, etc.) whenever one of its modules has been re-compiled; a new archive file is automatically created when the kernel is linked.

When a module within a dynamically loadable application is modified, the application must be re-linked in its own subsystem directory.

After re-linking an application, a new archive file (bn.exe, asn.exe, ace.out, etc.) must be generated in the global build directory; this must be done manually when an application has been re-linked.

For a list of kernel subsystems, see buildtib/Makefile, and look at the PROGLIBS list; applications can be found in the PROGEXES list.

```
**********************************************
** EXAMPLE 1 - rebuilding a kernel subsystem **
**********************************************
** Compile kernel module **
intruder->cd loader

intruder->touch ld_boot.c
intruder->build tib -nr

Mon Dec 30 11:15:18 EST 1996

make  -r TOOL=ghs TARG=tib PLAT=m68k GROUP=tib

loader: Mon Dec 30 11:15:23 EST 1996
gcc68 -OLA -Z54  -c -ANSI -68040 -ga -unsignedptr -Xredefine
-useDS -align4 -X89 -X325 -X380 -Z551 -Onounroll    -I../
include -I../edl/_tib -I../cdl/_tib -I../mdl_inc
-DTIMEKEEPER -DTIB_ONLY  -o _tib/ld_boot.o ld_boot.c
C-68000 1.8.7 Copyright (C)
1983,1984,1985,1986,1987,1988,1989,1990,1991,1992,1993,199
4 Green Hills Software, Inc. All rights reserved.
lib68 -crvy _tib/libloader.a _tib/ld_boot.o
 Deleted file: ld_boot.o
 Added file: _tib/ld_boot.o

Mon Dec 30 11:15:43 EST 1996

** Re-link TIB kernel **

intruder->cd ../buildtib

intruder->build tib -nr

Mon Dec 30 11:17:02 EST 1996

make  -r TOOL=ghs TARG=tib PLAT=m68k GROUP=tib
```

```
game: Mon Dec 30 11:17:05 EST 1996
wsp=`echo ${WSPACE} | sed 's/^\(.*\)router[0-9]*\///' | sed
's/^\(.*\)harpoon\///'` ; echo "char Image_directory[] =
\"${wsp}\";" > _tib/stamp.c
echo 'char Image_date[] = "'`date`'";' >> _tib/stamp.c
gcc68 -OLA -Z54  -c -ANSI -68040 -ga -unsignedptr -Xredefine
-useDS -align4 -X89 -X325 -X380 -Z551 -Onounroll    -I../
include -I../edl/_tib -I../cdl/_tib -I../mdl_inc -DTIB_ONLY
-o _tib/stamp.o _tib/stamp.c
C-68000 1.8.7 Copyright (C)
1983,1984,1985,1986,1987,1988,1989,1990,1991,1992,1993,199
4 Green Hills Software, Inc. All rights reserved.
l68 _tib/
game.cfe-z-g-y-y1:14:16-y3-t:\"start\"-s1:9:x-s1:13:d-s1:1
4:b-U:14,_tib/kernel.map,_tib/game.crf=_tib/start.o _tib/
game_boot.o _tib/stamp.o _tib/libgame.a /rte11/harpoon/dev/
tpearson/11/loader/_tib/libloader.a /rte11/harpoon/dev/
tpearson/11/game/_tib/libgame.a /rte11/harpoon/dev/
tpearson/11/hwf/_tib/libhwf.a  ....
_tib/liblast.a ./gamelink.dir
*** WARNING *** -S1 IGNORED, CONFLICTS WITH SECTION CONTENTS:
14
mapconv.pl _tib/kernel.map > _tib/kernel.nm
cd _tib;        \
        cofftoexe -K -r11.00 -i game.cfe -o game_bn.exe -k
TIBFRES ; \
        ldexe_compress game_bn.exe krnl_bn.arc

Parsing Input File: game.cfe
Program execution address space:
--------------------------------
Load Address: 0x30020000  Rom Address: 0x30020000 Size:
0x0017C644 Bytes  Entry point:  0x30024000

Input file information:
-----------------------
Input file:    game.cfe
File type:     Kernel file.
Tool name:     Oasys Linker
```

```
Output file information:
------------------------
Image Name:     dev/tpearson/11
Output file:    game_bn.exe
Platform Key:   (0101000B) BB M68000 MotherBoard (FRE FRE2
FRE2_60)
Revision:       11.00
Date Created:   Monday December 30 11:18:00 1996

Compressing ldapp.nohdr to ldapp.cmp
Using LZSS Encoder
..........................................................
......................Input bytes:        1558084
Output bytes:       806321
Compression ratio:  49%
cd _tib ; \
tib_cat bbdcmp.exe krnl_bn.arc krnl_bn.exe
cd exes; archive -av bn.exe  krnl_bn.exe snmp.exe pcap.exe
fsi.exe tms380.exe drs.exe osi.exe vines.exe lapb.exe x25.exe
xns.exe ipx.exe ip.exe fr.exe atm_dxi.e...

Creating new archive: bn.exe

Platform: BB
 -- Adding krnl_bn.exe                        FRE FRE2 FRE2_60
 -- Adding snmp.exe                                  FRE FRE2
ASN FRE2_60 ISP_60

    ....

 -- Adding hdwanlm.exe                        FRE FRE2 FRE2_60
 -- Adding de100.exe                               FRE2 FRE2_60
 -- Adding hdwancop.exe                                HDWANLM
 -- Adding mct1cop.exe                                MCT1_COP

** End EXAMPLE 1 **
```

```
*******************************************
** EXAMPLE 2 - rebuilding an application **
*******************************************

** Compile and re-link application **
intruder->cd ilacc

intruder->touch ilacc_ctrl.c
intruder->build tib -nr

Mon Dec 30 11:07:19 EST 1996

make  -r TOOL=ghs TARG=tib PLAT=m68k GROUP=tib

ilacc: Mon Dec 30 11:07:22 EST 1996
gcc68 -OLA -Z54  -c -ANSI -68040 -ga -unsignedptr -Xredefine
-useDS -align4 -X89 -X325 -X380 -Z551 -Onounroll   -pic32
-pid32 -I../include -I../edl/_tib -I../cdl/_tib -I../mdl_inc
-DTIB_ONLY  -o _tib/ilacc_ctrl.o ilacc_ctrl.c
C-68000 1.8.7 Copyright (C)
1983,1984,1985,1986,1987,1988,1989,1990,1991,1992,1993,199
4 Green Hills Software, Inc. All rights reserved.
lib68 -crvy _tib/libilacc.a _tib/ilacc_ctrl.o
 Deleted file: ilacc_ctrl.o
 Added file: _tib/ilacc_ctrl.o
l68 _tib/
ilacc.cfe-z-g-y-y3-t:\"ilacc_entry\"-sl:9:x-sl:13:d-sl:14:
b-U:14,_tib/ilacc.map,_tib/ilacc.crf=_tib/ilacc_hdr.o _tib/
libilacc.a /rte11/harpoon/dev/tpearson/11/hwf/_tib/libhwf.a
/rte11/harpoon/dev/tpearson/11/snmp/_tib/libsnmp.a /rte11/
harpoon/dev/tpearson/11/tib/_tib/libtib.a /rte11/harpoon/
dev/tpearson/11/pcap/_tib/libpcap.a /rte11/harpoon/dev/
tpearson/11/prioq/_tib/libprioq.a /rte11/harpoon/dev/
tpearson/11/rtl/_tib/librtl.a  ./ilacclink.dir
mapconv.pl _tib/ilacc.map > _tib/ilacc.nm

Parsing Input File: ilacc.cfe
```

```
Program execution address space:
--------------------------------
Load Address: 0x00000000  Rom Address: 0x00000000 Size:
0x0000825C Bytes  Entry point:  0x00002140

Input file information:
-----------------------
Input file:    ilacc.cfe
File type:     Loadable Application file.
Tool name:     Oasys Linker

Output file information:
------------------------
Image Name:    dev/tpearson/11
Output file:   ilacc_ucmp.exe
Platform Key:  (0101000B) BB M68000 MotherBoard (FRE FRE2
FRE2_60)
Revision:      11.00
Date Created:  Monday December 30 11:07:46 1996
Compressing ldapp.nohdr to ldapp.cmp
Using LZSS Encoder
..........
Input bytes:        33372
Output bytes:       18823
Compression ratio:  44%

Mon Dec 30 11:07:50 EST 1996

** Regenerate the archive file **
intruder->cd ../buildtib

intruder->build tib -nr archive

Mon Dec 30 11:09:41 EST 1996

make  archive -r TOOL=ghs TARG=tib PLAT=m68k GROUP=tib
```

```
cd exes; archive -av bn.exe  krnl_bn.exe snmp.exe pcap.exe
fsi.exe tms380.exe drs.exe osi.exe vines.exe lapb.exe x25.exe
xns.exe ipx.exe ip.exe fr.exe atm_dxi.exe wan.exe llc.exe
at.exe bgp.exe egp.exe ospf2.exe rarp.exe tcp.exe dls.exe
appn_cp.exe appn_ls.exe sdlc.exe nbase.exe tftp.exe lnm.exe
tn.exe ppp.exe debug.exe tnc.exe nbip.exe wcp.exe ntp.exe
isdn.exe lm.exe ping.exe atm.exe atmsig.exe atm_le.exe
igmp.exe dvmrp.exe ftp.exe quicsync.exe arp.exe xm.exe
sysl.exe crm.exe bgprs.exe st2.exe nsc_100m.exe ipex.exe
rredund.exe npt.exe run.exe ip6.exe sh_csmac.exe sh_sync.exe
sh_tcp.exe sh_tftp.exe sh_snmp.exe sh_fr.exe sh_ip.exe
munich.exe fmpb.exe pim.exe hwcomp.exe bot.exe hwf.exe
fddi.exe dsde2.exe dst.exe dtok.exe enet2.exe qenet.exe
qsync.exe hdlc.exe hssi.exe ilacc.exe lance.exe ds2180.exe
ds2181.exe e1.exe t1.exe hfsi.exe mct1e1.exe atmalc.exe
atmalcop.exe hdwanlm.exe de100.exe hdwancop.exe mct1cop.exe

Creating new archive: bn.exe

Platform: BB
 -- Adding krnl_bn.exe                      FRE FRE2 FRE2_60
 -- Adding snmp.exe                              FRE FRE2
ASN FRE2_60 ISP_60
 -- Adding pcap.exe                              FRE FRE2
ASN FRE2_60 ISP_60
 -- Adding fsi.exe                               FRE FRE2
ASN FRE2_60 ISP_60

    ...

 -- Adding hssi.exe                         FRE FRE2 FRE2_60
 -- Adding ilacc.exe                        FRE FRE2 FRE2_60
 -- Adding lance.exe                        FRE FRE2 FRE2_60
 -- Adding ds2180.exe                       FRE FRE2 FRE2_60
 -- Adding ds2181.exe                       FRE FRE2 FRE2_60
 -- Adding e1.exe                           FRE FRE2 FRE2_60
 -- Adding t1.exe                           FRE FRE2 FRE2_60
 -- Adding hfsi.exe                         FRE FRE2 FRE2_60
 -- Adding mct1e1.exe                       FRE FRE2 FRE2_60
 -- Adding atmalc.exe                           FRE2 FRE2_60
 -- Adding atmalcop.exe                                ATMALC
 -- Adding hdwanlm.exe                      FRE FRE2 FRE2_60
-- Adding de100.exe                                    FRE2 FRE2_60
 -- Adding hdwancop.exe                                     HDWANLM
 -- Adding mct1cop.exe                                     MCT1_COP

** End EXAMPLE 2 **
```

## Debugging

There are some special considerations for debugging dynamically loadable applications because their load address is not known until they are actually loaded.

The application load address must be read from the TI console via the 'loadmap' command (after the loader has loaded it), and then fed into the debugger when you load the application symbol table:

```
[2:TN]$ loadmap 2

----------------------
Loadmap from SLOT 2:
----------------------
--> arp.exe            0x304ecdd0  0008944
--> tcp.exe            0x30508df0  0057776
--> tftp.exe           0x304ef0d0  0020488
--> snmp.exe           0x304ff730  0030360
--> tn.exe             0x304f40f0  0038424
--> ip.exe             0x304c0f70  0179780
--> hdlc.exe           0x30491560  0058368
--> lance.exe          0x30522f00  0008840
--> dsde2.exe          0x305251a0  0005232
```

The GDB-PPC Debugger used with the FRE had been modified to query the target for the load address. However, the image still needs to be loaded first.

Alternatively, this step can be avoided by linking the application into the kernel slab for debugging purposes.

Loadmaps are available on-demand from the TI; they are also dumped into the system log so that dynamic addresses found in stack dumps can be resolved post-mortem.

The 'stkscan' and 'logscan' tools assist in the post-processing of log information:

- Cut/paste the loadmap info into a temporary file (/tmp/stk)
- Move to the directory that containing the linker map files $ cd buildtib/maps
- stkscan the faulting address $ stkscan /tmp/stk 0x327c11a8

The output looks something like:

```
intruder->stkscan /tmp/stk 0x327c11a8

0x327c11a8   [fsi     @ 0x5888 ] == fsi_xmt_oper_act+0x0
```

## Software Release

We have the potential to release each executable module independently, and then have a compatibility matrix in the loader to enforce compatibility rules.

While the benefit of this software release model is great, it presents nightmare-ish test and processing matrices for the SQA and Manufacturing departments.

## Adding a New Subsystem

### Kernel Subsystems

Add code to spawn the kernel application in ld_phase2.c.

```
/* Create the Data Path gate */

ld_svc(GID_DP_INI, LDF_NONE, dp_init_act, 0);
```

This reference to the subsystem's entry point causes the subsystem to be linked into the kernel.

Add subystem to PROGLIBS line in the global build directories (buildtib, buildace, buildpir, ...).

### Application Subsystems (Dynamically Loadable)

Make sure application conforms to requirements listed above.

Modify the Makefile in your subsystem to specify PIC and any PROGLIBS required for linking.

Add your subsystem to the PROGEXES line in the global build directories (buildtib, buildace, buildpir, ...).

<subsys>_hdr.c file must be updated (see doc list below)

"Register" application with dynamic loader by grabbing an index in the loader's global include file, and then adding the subsystem name to the loader's.

Add attribute to loader MIB record and add code to loader files to load/unload new application.

See 'dyn_load_user.ps' document below for all the details. Also, see http://www.enghost/clearcase/router/add_exe.html.

## Related Documentation

```
/rte1/harpoon/doc/sysman/dyn_load.ps

/rte1/harpoon/doc/sysman/dyn_load_user.ps

/rte1/harpoon/doc/sysman/Email-archive/
dynamic_loadr_rel.txt (somewhat outdated)
```

# Chapter 15
# Symmetric Multi-Processing

## Symmetric Multi-Processing

Symmetric Multi-Processing (SMP) was added as part of the Bluefish (ARE) project. In order to meet the agressive forwarding rates needed for Bluefish, it was determined that a single processor wouldn't work. Therefore, Bluefish was designed as a dual processor system. To date, only Bluefish and its derivatives (Blackfish, FRE-3) support SMP.

The challenge in adding SMP to GAME was figuring out how to do it without having a huge impact on the 2 million or so lines of code already in existence.

The major problem when applying SMP to an existing code base is how to protect data that may be modified by both processors at the same time. This means you either need to add locks to all data stuctures or you can't concurrently schedule gates which modify the same data structure.

Obviously, adding locks to all data structures would have a huge impact on the existing code, not to mention all the new deadlock bugs it would introduce.

Since GAME already organizes gates into family trees, this seemed to be a logical way to make an educated guess about who shares data structures. For example, it is not likely that IP and Appletalk share memory. This is the approach that was used.

There is also one critical observation which can be made: In order to meet the Bluefish performance goals. Is isn't necessary to have all gates running in parallel. Only the forwarding path really needs to be SMP. If the control path isn't that optimal, it is still ok.

## Gate Classification

Out of the above came the notion of classifying gates into one of 5 types based upon how they share memory. This, in conjunction with a gate's ancestry, allows the SMP scheduler to avoid scheduling two gates which may modify shared memory concurrently. A gate's ancestry starts with the first gate in a family tree created by the loader.

A gate's classification is set by an option to the g_req() syscall. This may be set when the gate is created or by the gate itself. A gate can change its classification by another call to g_req(). This change takes effect the next time a gate gets scheduled.

Here is the list of the different gate classifications, the associated g_req() option, and a description.

### Global/G_REQ_GLOBAL

The most excusive category. A Global gate will be the only gate executing in the system. The 2nd processor will be held in a tight idle loop. This is used for gates such as the MIB (which shares memory with practically every application on the box). Applications are strongly discouraged from declaring their gates as Global.

### Ancestor exclusive/G_REQ_ANCESTOR

This is the default gate type. An Ancestor exclusive gate will not run with any other ancestor exclusive gate. This is the default type because we can't be sure which gates do or don't share memory outside of their ancestory. These gates are assumed to share memory outside their ancestory. The first step to SMPize a subsystem is to determine if it ever goes outside its ancestory. If it doesn't, its type can be changed to Clean Ancestor.

### Clean Ancestor/G_REQ_CLEAN_ANCESTOR

Clean Ancestor gates do not share memory outside of their own ancestory. Therefore, it is ok to run them with other Clean Ancestor or Ancestor Exclusive gates, provided those gates come from a different ancestory. Ideally, most of the control path would be of this type.

### Clean Reader/G_REQ_CLEAN_READER

The Clean Reader type was created to aid in making the forwarding path efficient. The Clean Reader should be used when the gate only reads data structures owned by the rest of its ancestory (i.e. the forwarding table). A Clean Reader will run concurrently with another Clean Reader from the same ancestory. This is ok as both are only reading data. A clean reader will not execute if a non-Clean Reader from the same ancestory is running. This is because the other gate may be modifying the shared data structure.

### Clean/G_REQ_CLEAN

This is the ideal gate for the forwarding path. A Clean gate is free to run with any other gate (except for Global). It doesn't share any data or the data it does share is read only and never modified. Clean gates can achieve their 'cleanliness' by using the atomic operations described below. But since Clean usually implies the datapath, you usually need to ensure that a clean gate doesn't get blocked for any substantial period of time.

## SMP Scheduler

The SMP scheduler makes use of a single activation queue and idle queue as the standard scheduler does. Each processor decides what gate it wants to run next by looking at all the gates available in the activation queue as well as what gate the other processor(s) is (are) currently running (if any). When it finds a gate which satisifies the scheduling requirements, it executes that gate.

This means that, unlike single-processor GAME, gates may not execute in the strict order they appear in the activation queue. However, all gates in the activation queue still need to be executed before the slot goes idle (allowing module interrupts).

If a processor can't find a gate to execute, possibly because it would conflict with the gate already running on the other processor, it idles itself waiting for the other processor to complete. Once the other processor completes, one of the processors (which one depends upon which acquires the lock first) will start running the next gate on the activation queue (the next gate will always be eligible as both of the processors will have been idle).

The scheduling rules for two-processor SMP are summarized in the following chart.

The SMP types are:

| | |
|---|---|
| G | Global |
| A | Ancestor exclusive |
| CA | Clean Ancestor |
| CR | Clean Reader |
| C | Clean |

Across the top of this chart is the type of gate currently running on the other processor. Down the side is the type of gate the current CPU would like to run. A 'Y' indicates that, yes, the gate being scheduled will execute in parallel with the currently running gate. A '-' means the gate will not run. A '*' indicates the other scheduler will be idle, so this state will never happen.

In places where it matters, it is indicated whether the gate being scheduled is in the same or different ancestry. If "same" or "diff" isn't indicated, then it doesn't matter.

| Gate Being Scheduled | Running Gate | | | | |
|---|---|---|---|---|---|
| | G | A | CA | CR | C |
| G | * | - | - | - | - |
| A same | * | - | - | - | Y |
| A diff | * | - | Y | Y | Y |
| CA same | * | - | - | - | Y |
| CA diff | * | Y | Y | Y | Y |
| CR same | * | - | - | Y | Y |
| CR diff | * | Y | Y | Y | Y |
| C | * | Y | Y | Y | Y |

## The Kernel Lock

The kernel is one place where memory sharing across processors is very likely. This could happen if the gates running concurrently happen to make overlapping system calls.

To prevent problems here, the kernel is protected by one lock. Only one processor may be in the kernel code at any point in time. This includes the scheduler, implying that only one processor will be picking a gate to run at any time.

## Interrupts

With multiple processors, interrupt handling becomes more interesting. GAME solves the issue by requiring the kernel lock before entering the interrupt processing code. Therefore, only one CPU can handle interrupts at a time.

Interrupts are only enabled when the CPU is in the scheduler. In order to be in the scheduler, the processor must first have acquired the kernel lock. Each processor will enable interrupts between each gate it executes.

On the interrupt handler side, the kernel lock must be acquired before interrupt processing makes its way into the kernel. This is necessary because some error interrupts will be seen by both processors and we need to serialize their handling.

This all can work because the kernel lock is special - it can nest. The owner of the lock is monitored, so when a CPU goes to request the lock, the lock code knows if that CPU already owns the lock. This information returned from the locking call informs the caller as to whether or not a nested lock has occurred. This lets the caller knows whether or not it should free the lock when it is done. If the CPU already had the lock, it doesn't free it.

The only time the interrupt code executes under a non-nested lock on a FRE is for level 4 interrupts. Level 3 interrupts are always serviced with a nested lock because the CPU has to enable the interrupts.

The sequence of events is as follows for "between-gate" interrupts:

1. Get the kernel lock.
2. Enter scheduler.
3. Call g_poll() to enable between gate interrupts.
4. Interrupt occurs.
5. Enter interrupt_handler.
6. Get the kernel lock, finding out that it is currently owned by this CPU.
7. Call the interrupt service routine.
8. Return from interrupt processing (note the lock was not freed).
9. Pick the next gate to run.

10. Free the kernel lock.
11. Run the selected gate.

## Gate Creation, Death, and Mappings

Since the kernel lock has to be obtained to enter kernel code, it is impossible for multiple processors to create or kill a gate at the same time. However, it is possible for a gate on one processor to kill the gate that is currently active on the other processor. This race is handled by making the map_map() gate a Global gate.

As explained earlier, when a gate is killed, the head of the scheduler's activation queue is modified such that the first entry is the map_map gate. When the map-triggering gate pends, the scheduler runs and sees that the first entry on the queue is a Global gate. It therefore idles the CPU, waiting for the other CPU (which may be running the newly killed process) to finish. Once the other CPU finishes, one of the CPUs runs the map_map gate (while the other idles), which cleans up the dead gate's resources and schedules the mapping activations.

## Atomic Locks

There are two RTL routines which implement atomic operations. These are atom_incr_int32() and atom_update_int32(). These can be used to make gates clean even if they share memory.

The atom_incr_int32() is used to update MIB stats. It would be a shame to not be able to mark a gate Clean only because it needs to count stats. The atom_incr_int32() provides an atomic increment so that multiple Clean gates can update the same stat.

The atom_update_int32() can be used to perform an atomic update of a value. This can be used to implement a busy-wait loop to serialize access to a data strucure.

Under the PowerPC architecture, atomic operations are not performed by doing an atomic read-modify-write operation on the bus as you may expect. Rather, the PPC has what it calls a 'reservation'. To do an atomic operation, you first perform a load with reservation. This causes the PPC to remember which cache line your load came from. Once a new value is ready to write, be it an increment or setting of a lock, the processor does a store w/ reservation. Unlike other stores, this store will only complete if some other processor hasn't modified the reserved cache line. If the store fails, another load/store cycle needs to be done. All this work is what the atom_incr_int32() and atom_update_int32() are doing.

Notice how the reservation happens on a cache line boundary. This means that in order to get the highest likelyhood for the store to complete, that cache line shouldn't be in high use.

To help with atomic locks, a number of macros have been defined in include/atom.h. A brief summary of these is:

```
SMP_LOCK_ALLOC(lock_ptr)
```

Allocates a block of memory and returns the lock_ptr which will be properly aligned within it for atomic operations.

```
SMP_LOCK_UNALLOC(lock_ptr)
```

Frees the memory acquired by SMP_LOCK_ALLOC.

```
SMP_LOCK_ACQUIRE(lock_ptr)
```

Uses atom_update_int32() to facilitate a busy-wait binary lock. This will return only after the lock has been acquired. But, it is a busy-wait, NOT a pending call. Therefore, while you own the lock, "Thou shalt not pend!"

```
SMP_LOCK_RELEASE(lock_ptr)
```

Releases the lock acquired by SMP_LOCK_ACQUIRE().

## SMP Operations On Non-SMP Systems

Applications are free to use the SMP g_req() options, atomic routines, etc, on non-SMP systems. These are all appropriately stubbed out. For example, atom_incr_int32() will still perform an increment, but not atomically (since there is only one processor).

# Appendix A
# Performance and Scalability

## Performance

Every gate affects the forwarding/zero-drop numbers/system performance.

Since the same CPU is used for forwarding and other processing*, the attitude of "My gate is not in the forwarding path, so I don't need to optimize it very well" is dangerous. Any single gate can kill the performance of the box.

Under GAME, once a gate is running, it can hold the CPU indefinitely. On FRE hardware, a four-second watchdog timer will terminate the execution of a runaway gate. However, four seconds goes far beyond the time necessary to invalidate most service guarantees. It is even much too long to provide a decent level of ASAP FIFO service, since the link drivers, the backbone drivers, and the pieces of forwarding code are not allowed to run.

The author has observed (and been involved in) re-design efforts of software that used too many buffers or too much memory. Since the penalties for misuse of these resources ("buffer hog" log messages, "out of memory" faults) are much more apparent than the misuse of CPU, the trend has been to use more CPU and less buffers and memory. This results in gates that need to run long periods of time, but do not chew up buffers or transient memory. Most developers are under the impression that if their gates do not cause watchdog timers on the FRE hardware, they are not causing any harm. This is definitely not true.

*Traditionally this has been true of Wellfleet routers. Frame/Cell switching and future router systems will do more forwarding in hardware, removing the CPU from the forwarding path.

Since GAME does not provide the equivalent of "CPU hog" messages or enforce any time-slicing, it is totally the responsibility of the application developer to ensure that their gates act in a manner to realize a "well-behaved system". A well-behaved system is one that does not dramatically reduce the forwarding performance while performing CPU intensive non-forwarding operations (i.e., allows those gates involved in forwarding to run at regular intervals).

Whether this tuning gets done or not usually depends on how much scaling or zero-drop testing is performed. Unfortunately, even for the conscientious developer, it is impossible to realize this goal. This is largely due to two reasons:

- A gate cannot get access to enough information to make an intelligent decision of whether or not to give up the CPU at any given time.
- Even if it had access to this information, the CPU is not given to the device drivers until ALL gates on the slot have idled. So, even if one gate is being a good citizen, it does not make for a well-behaved system. It is the aggregate action of ALL of the gates on the scheduler queue that determines when interrupts are processed.

As a case study, lets look at one small piece of the BGP routing protocol implementation: the connection transmit gates. Each connection transmit gate is responsible for sending routing updates to the router at the other end of the connection. Many (tens) of connection gates can exist on a single slot.

When a router learns 10,000 new routes, a connection transmit gate has to examine the new routing entries, process each against the announce policy configured for their particular connection, possibly include the route in an UPDATE message and update its "advertised routes" table, and send the UPDATEs. This obviously takes a lot of time for 10,000 routes and without any explicit surrendering of the CPU, the gate will watchdog on a FRE.

The problem was made workable (i.e., it doesn't watchdog, but it's not really a well-behaved system) by inserting g_idle calls into the code. The G_IDLE_CHECK flavor was used, which will pend the calling gate whenever there is driver or backbone activity to be processed. This approach does not work well for a few reasons:

- The g_idle calls are not free.
- Knowing when and where to insert g_idle calls is a black art, since there is no way to determine how long a gate has been hogging the CPU.
- Since all transmit connection gates tend to act on the same routes at the same time, the number of connections on a slot affects how well-behaved that slot is. g_idle placement that works when only one connection gate is active can cause massive problems if five gates are active, because all of the gates are going to be on the scheduler queue and each will grab about the same size slice of CPU time. This therefore requires over-g_idle-ing the code, which leads to the next problem.
- Just because a driver or backbone interrupt is pending doesn't necessarily mean that the CPU has to be surrendered. DAS traces of the BGP code show that some g_idle calls pend the calling gate to service only one received buffer from the backbone. Previous research has shown that running such a code strip for a single buffer is very inefficient, as it causes unnecessary thrashing of the instruction cache.

The indeterministic allocation of the CPU resources appears to be the largest problem in regard to providing service guarantees. If we cannot guarantee that the gates involved in forwarding (drivers, decaps, encaps, protocol forwarders) will run at a regular frequency, we cannot guarantee anything. A method of making this more deterministic could have benefits for the normal ASAP FIFO traffic (better zero-loss numbers) as well as real-time traffic.

## Avoid Transient Resource Allocation

Most routing protocol applications are designed so that the gates that transmit and receive routing information packets are separated from the gate or gates that maintain the routing tables*. Therefore, some method of relaying this information is necessary.

Scaling becomes a huge problem if resources have to be allocated temporarily (i.e. 'transient' resources) to accomplish the inter-gate communication, whether the resource is buffers or memory. Huge updates will kill the box because of the (potentially large spikes in) transient resource usage.

The most successful approach to this problem has been to have the routing table gate be a parent of the transmit/receive gates and allow the child gates to modify the parent's memory. The tbl and rtbl subsystems have been modified to make sure that any additional memory allocated is always owned by the parent gate.

This approach obviously requires additional care by the programmer. The shared resource (routing table structures) usually need to be modified in a mutually exclusive fashion.

*Similar requirements are often found in designs for other (non-routing types of) applications which must maintain some kind of shared common central database that gets modified in response to network activity.

## Use RTBL/UTBL, not TBL, for Non-Forwarding/Filtering Tables

TBL works best for forwarding tables. These forwarding tables usually do filtering simultaneously. This is what TBL was designed for. Also, these forwarding tables should contain the least number of entries necessary (i.e., it should be a cache). Once you get beyond several hundred entries, the modification times for the table grow to problematic lengths.

For everything else, RTBL and UTBL should be used. These services were designed for faster modification. The lookup time is pretty good too.

## Image Size

Generally speaking, keeping the image size small is a goal. Excessive image size can lead to excessive memory consumption, which can have negative impact on scalability of a hardware platform/product.

Of particular concern are embedded strings for use in logged (g_log()) messages. All these log strings should be defined in your .edl file, and the only arguments to the g_log() function should be the message identifier and any necessary numeric values that the log message in question calls for. If you pass a string as an argument to g_log, you not only have increased image size and memory consumption to store the string, you have also consumed CPU time/power to format the string in question (especially if you do this via sprintf()).

Furthermore, strings take up a lot of space in the log. Excessively long log messages with embedded strings causes the log to fill up and wrap faster/more often. This can lead to a loss of important log data. In the extreme, the log wraps so fast you never catch the event that you really need to see that will clue you into the real problem in a system.

## Accessing Memory, Hitting/Missing Cache Etc.

Accessing (i.e. reading./writing) memory is generally to be minimized in designs. Today's processor speeds have reached levels that make memory bandwidth (or access time) the bottleneck for system performance. Accessing memory means the CPU has to go "off-chip" across a bus to access registers in some other memory chip.

Imagine an algorithm that is going to add a constant offset to each element in an array. Furthermore assume the constant offset is part of a data structure X:

```
typedef struct X {
    /* other cells in data structure X */
    u_int32 offset;
    /* other cells in data structure X */
} X;
```

One possible encoding of the algorithm might be:

```
void
add_offset ( x, array, size)
X    *x;                /* pointer to struct X with offset */
u_int32 array[];        /* array of 32 bit unsigned integers */
u_int32 size;           /* number of entries in array */
{
    for ( i = 0; i++ ; i < size )
    {
     array[i] += x->offset;
    }
}
```

In the above instance, the CPU will execute the expensive fetch from memory for the offset (i.e. the contents of x->offset) for each iteration of the for loop.

Now consider this alternative:

```
void
add_offset ( x, array, size)
X           *x;       /* pointer to struct X with offset */
u_int32      array[]; /* array of 32 bit unsigned integers */
u_int32     size;     /* number of entries in array */
{
    register u_int32 offset;
    offset = x->offset;
    for ( i = 0; i++ ; i < size )
    {
          array[i] += offset;
    }
}
```

The algorithm has the same order of complexity as the first (both are O(N) time-complex), but this second alternative will execute significantly faster in real time. The reason is that the fetch from memory for the constant offset is only done once in the second alternative, rather than once per loop. So while both algorithms execute the loop the same number of times, the first algorithm does an extra memory fetch/read per loop.

While some of the 'smarter' compilers may invalidate the above analysis, optimizing correctly to eliminate the cited waste, such optimization has been turned off in at least some of the compilers currently being used for GAME development at the time of this writing. The recommendation here is to not rely on the compilers for this kind of optimization and to write the code itself in such an optimal fashion so as to get good performance regardless of what compiler (options) are being used.

An even worse situation would arise from use of some of the macros defined to access the contents of the GAME buffer (BUF) header. Some of these macros make references to multiple fields of the BUF structure, multiplying the penalty. Buffer memory does not get cached, making the problem even uglier. The bottom line is that any references to buffer headers/contents should be done once and stored in a (hopefully register) variable for future reference (using registers like this is also basically a good idea for any memory references that can be similarly managed).

Memory reads are generally more expensive than writes. The speed of the memory access also depends on which memory is being accessed. Platforms have different kinds of memory, and different sized caches. Hitting the cached memory is always faster than having to go out on the bus to fetch form non-cached memory.

Memory access speeds, from slower to faster, on traditional Wellfleet platforms is as follows:

- un-cached DRAM
- dual-port (protocol headers in buffers)
- cached DRAM
- And, of course, a register is fastest.

These differences are especially important on the PowerPC based platforms, where a slow memory fetch costs lots and lots of unused CPU cycles.

Buffer memory is never cached.

A couple of things to do to make your code run faster:

- Organize your data structures so that the fields you access within a particular routine are all located in one or two 16-byte cache lines. Putting comments in the *.h file explaining that you did this will help keep someone from screwing it up later.
- If you are going to access something repeatedly within a buffer, put it into a stack variable instead. This way you will do a cached access.

# Index