SCRAMNet GT

Effective Use of SCRAMNet GT Network Interrupts

Introduction

Network interrupts are messages that can be passed among nodes on a SCRAMNet GT network. They can be used for event notification and process synchronization between different processes distributed across the network. The SCRAMNet GT API provides an efficient and powerful interface for using the network interrupt mechanisms provided by SCRAMNet GT. This document gives an overview of the available network interrupt mechanisms and discusses how to make use of them in software.

Network Interrupts

There are two types of network interrupts: broadcast and unicast. Each broadcast interrupt that is sent can be received by every node on the network. Unicast interrupts are sent to (and received by) a specific node on the network.

Each broadcast interrupt has a designated broadcast ID in the range 0 to 31. Each ID value can be used to indicate a different user-defined event took place. The ID value is especially useful for categorizing interrupts into different classes of events. For instance, all interrupts sent to signal a configuration change in the system could use broadcast ID 16.

Both broadcast and unicast interrupts include an additional user-defined 32-bit value. This value can be used to pass details about the event to the receiving node. For example, it may pass the GT memory offset and/or the buffer size of an updated GT memory region.

The reception of specific interrupts at a node can be enabled or disabled by setting the interrupt masks for the node. This is done using the scgtSetState() function or using gtmon. See below for more information.

Setting the Interrupt Masks

Network interrupt messages trigger real hardware interrupts and are subsequently processed by the SCRAMNet GT driver. Therefore, it can be beneficial to disable reception of those interrupts that are not being used by applications on a given node. This can save processing time in system configurations where network interrupt messages are heavily used. The interrupt masks allow for selection of which interrupts

CURTISS WRIGHT Controls Electronic Systems

are received and processed. This selection applies to the SCRAMNet GT device and to all processes using the device rather than on a per process basis.

The broadcast interrupt mask is a 32-bit vector used to enable/disable reception of broadcast interrupts with selected broadcast interrupt ID values (not to be confused with node ID). By default, the mask value is 0, indicating that all broadcast interrupts are disabled. To enable reception of broadcast interrupts that have ID 4, set bit 4 to 1. The mask should be set to 0x10. To receive interrupts that have ID 0 or ID 4, set bits 0 and 4 of the mask to 1. The mask value should be 0x11 in this case.

Example Broadcast Interrupt Mask
(Broadcast interrupt IDs 0 and 4 are enabled)

0	0	0	0	0	..	0	0	0	1	0	0	0	1
31	30	29	28	27	..	7	6	5	4	3	2	1	0

The broadcast interrupt mask can be set using gtmon.

```
gtmon -b 0x11
```

You can also use scgtSetState() inside of your program to set the mask.

```
scgtSetState(&handle1, SCGT_BROADCAST_
INT_MASK, 0x11);
```

To set bits in the mask without changing the enabled bits:

```
uint32 bmask;
bmask = scgtGetState(&handle[1], SCGT_
BROADCAST_INT_MASK);
scgtSetState(&handle, SCGT_BROADCAST_
INT_MASK, bmask | 0x11);
```

Set the mask value to 0xFFFFFFFF to receive all broadcast interrupts.

Unlike broadcast interrupts, unicast interrupts do not have an associated ID. They can be enabled or disabled by setting the unicast interrupt mask to 1 or 0 respectively.

To enable unicast interrupts using gtmon:

```
gtmon --uinton
```

To disable unicast interrupts using gtmon:

```
gtmon --uintoff
```

You can also use scgtSetState() to enable or disable unicast interrupts.

```
scgtSetState(&handle, SCGT_UNICAST_
INT_MASK, 1);
or
scgtSetState(&handle, SCGT_UNICAST_
INT_MASK, 0);
```

Interrupt-Self State

In usual operation, a SCRAMNet GT node does not receive network interrupts that it sent (more precisely, those sent by a node having the same node ID). However, sometimes it is useful to receive these interrupts. For example, two processes that are designed to work across the GT network on different nodes may be running on the same node. For one process to receive network interrupts from the other process (or itself), the interrupt-self state must be enabled.

To enable the interrupt-self state using gtmon:

```
gtmon --sinton
```

To disable the interrupt-self state using gtmon:

```
gtmon --sintoff
```

You can also use scgtSetState() to enable or disable the interrupt-self state.

```
scgtSetState(&handle, SCGT_INT_SELF_
ENABLE, 1);
or
scgtSetState(&handle, SCGT_INT_SELF_
ENABLE, 0);
```

Error Interrupts

Error interrupts are used to report errors occurring at the local node. An error interrupt is generated each time a link error occurs and when the driver's internal interrupt queue overflows. They are retrieved from the driver using the same mechanism as, and during, network interrupt retrieval (scgtGetInterrupt()). Error interrupts are not network interrupts, and cannot be sent.

The possible error-interrupt values are:

SCGT_LINK_ERROR - Link error occurred.

SCGT_DRIVER_MISSED_INTERRUPTS – The GT device received interrupts faster than could be processed, which caused the interrupt queue to overflow.

The scgtInterrupt Structure

The scgtInterrupt structure is used to store information about a network interrupt. It specifies a network interrupt to send when passed to the scgtWrite() function or a network interrupt received when returned from the scgtGetInterrupt() function.

The scgtInterrupt structure is defined as follows:

```
typedef struct scgtInterrupt
{
    uint32 type;
    uint32 sourceNodeID;
    uint32 id;
    uint32 val;
} scgtInterrupt;
```

Before sending an interrupt, set the members of scgtInterrupt as described in the following table. The sourceNodeID member is determined by the GT hardware and does not need to be set.

scgtInterrupt member	Unicast	Broadcast
type	SCGT_UNICAST_INTR	SCGT_BROADCAST_INTR
id	Destination node ID [0.255]	Broadcast ID [0.31]
val	User-defined value	User-defined value

After receiving an interrupt, the members of scgtInterrupt are as described in the following table.

scgtInterrupt member	Unicast	Broadcast	Error
type	SCGT_UNICAST_INTR	SCGT_BROADCAST_INTR	SCGT_ERROR_INTR
sourceNodeID	Source Node ID	Source Node ID	Reserved
id	Reserved	Boardcast ID	Reserved
val	User-defined value	User-defined value	Error code

Note that the id member is used differently depending on the type of interrupt. When sending or receiving broadcast interrupts, the id member is set to the interrupt ID of the broadcast interrupt (value in the range [0,31]). When sending a unicast interrupt, this member is set to the destination node ID. The id member is not valid when receiving unicast or error interrupts.

The val member is used to send a user-defined 32-bit value along with a broadcast or unicast interrupt. If an error interrupt is received, this member holds the error code associated with the error.

Sending Interrupts

To send a network interrupt, use the scgtWrite() function. If a data buffer is specified, the scgtWrite() function will write the data to GT memory before sending the network interrupt.

To send a broadcast interrupt:

- Set type to SCGT_BROADCAST_INTR
- Set id to the desired interrupt ID (in the range [0,31])
- Set val to the desired 32-bit value.

The following code demonstrates how to send a broadcast interrupt that has broadcast ID 20.

```
scgtInterrupt intr;

intr.type = SCGT_BROADCAST_INTR;
intr.id = 20; /* Broadcast interrupt
ID */
intr.val = 1234; /* set this to your
desired value */

scgtWrite(&handle, 0, NULL, 0, 0,
NULL, &intr);
```

To send a Unicast interrupt:

- Set type to SCGT_UNICAST_INTR
- Set id to the desired destination node ID.
- Set val to the desired 32-bit value.

The following code demonstrates how to send a unicast interrupt to the SCRAMNet GT node with node ID 5.

```
scgtInterrupt intr;

intr.type = SCGT_UNICAST_INTR;
intr.id = 5; /* destination node ID
*/
intr.val = 1234; /* set this to your
desired value */

scgtWrite(&handle, 0, NULL, 0, 0,
NULL, &intr);
```

To send a Unicast interrupt:

- Set type to SCGT_UNICAST_INTR
- Set id to the desired destination node ID.
- Set val to the desired 32-bit value.

The following code demonstrates how to send a unicast interrupt to the SCRAMNet GT node with node ID 5.

```
scgtInterrupt intr;

intr.type = SCGT_UNICAST_INTR;
intr.id = 5; /* destination node ID
*/
intr.val = 1234; /* set this to your
desired value */

scgtWrite(&handle, 0, NULL, 0, 0,
NULL, &intr);
```

Receiving Interrupts

The SCRAMNet GT driver stores network interrupts that were received by the GT device in a circular queue in system memory. Each process that retrieves network interrupts from the queue has its own scgtIntrHandle. This handle is used to mark a place inside the queue to indicate which interrupts a process has retrieved using scgtGetInterrupt(). This method of queuing allows multiple applications sharing a single SCRAMNet GT device to have access to all incoming network interrupts. This method also guarantees that the most recent interrupt messages are not missed and that notification of old interrupts that were missed can be passed to the user.

Before we can retrieve network interrupts from the SCRAMNet GT driver, we must first call scgtGetInterrupt() with an intrHandle value of -1. This tells scgtGetInterrupt() to initialize our interrupt handle so we can retrieve interrupts starting from this point in time.

For example:

```
scgtIntrHandle intrHandle = -1;
scgtInterrupt intrBuff[100];

scgtGetInterrupt(&handle, &intrHandle,
intrBuff, 0, 0, &numIntrRet);
```

In this example, the scgtGetInterrupt() function will return immediately without retrieving any interrupts. A non-zero timeout and numInterrupts may be specified during the first call to scgtGetInterrupt() if desired. After the call, the interrupt handle is initialized.

Now, to retrieve network interrupts from the driver, call scgtGetInterrupt() supplying a non-zero maximum number of interrupts to return. Here we supply 100 as the maximum number of interrupts to retrieve.

```
uint32 ret;
scgtInterrupt intrBuff[100];

ret = scgtGetInterrupt(&handle,
&intrHandle, intrBuff, 100, 250,
&numIntrRet);
```

The scgtGetInterrupt() function will retrieve interrupts from the driver's interrupt queue that have not been previously retrieved. If there is at least one interrupt in the queue, the scgtGetInterrupt() function will return immediately allowing the calling program to process the interrupt(s). If there are no interrupts in the queue to retrieve, scgtGetInterrupt() will wait for interrupts for up to the specified timeout value. The timeout value is 250 milliseconds in our example. During the wait time, the CPU is released for other processes (or threads) to use until an interrupt arrives. After retrieving one or more interrupts from the driver the scgtGetInterrupt() function automatically updates the interrupt handle to reflect the new location in the driver's interrupt queue.

Sender/Reciever Communication Example

In the following example, a sender of a point-to-point transfer writes a buffer of size 0x2000 at offset 0x1000. After the write operation completes, a unicast interrupt is sent to node 5. The reader process (running on node 5) waits until it receives a unicast interrupt and then reads the data from GT memory. The val member of the interrupt contains the GT offset from which to read. Here, the size of the data is hardcoded for both processes, but the size of the buffer could be stored in the first element at the offset indicated by val, for example.

Note that the receiver code has to be started before the sender so it sees the network interrupt.

Sender code

```
uint32 gtOffset = 0x1000;
scgtInterrupt intr;

intr.type = SCGT_UNICAST_INTR;
intr.id = 5; /* destination node ID
*/
intr.val = gtOffset; /* set this to
your desired value */

scgtWrite(&handle, gtOffset,
dataBuffer, 0x2000, 0,
&bytesTransferred, &intr);
```

Here we used a feature of sending interrupt automatically after transfer completion.

Receiver code

```
scgtIntrHandle intrHandle = -1;
uint32 numIntrRet = 0;
uint32 done = 0;
scgtInterrupt intrBuff[1];

/* initialize intrHandle */
scgtGetInterrupt(&handle, &intrHandle,
&intrBuff, 0, 0, &numIntrRet);

while (!done)
{
  scgtGetInterrupt(&handle,
  &intrHandle, &intrBuff, 1, 250,
  &numIntrRet);
  if (numIntrRet != 0)
  {
     if (intrBuff[0].type == SCGT_
     UNICAST_INTR)
     {
        gtOffset = intrBuff[0].val;
        done = 1;
     }
  }
}

scgtRead(&handle, gtOffset,
dataBuffer, 0x2000,
0, &bytesTransferred);
```

Additionally, we could use the sourceNodeID field in the interrupt structure to look for the interrupt from a specific node in the receiver code.

Dedicated Interrupt Thread

In systems that utilize asynchronous even notification, network interrupts can be received at any time during a program's execution. If the program must respond to the interrupt in a timely manner, spawning a thread that is dedicated to interrupt handling can be advantageous. The thread would continuously call scgtGetInterrupt(). For each network interrupt retrieved, the thread would then call a desired function or perform some task.

While there are no incoming interrupts, the scgtGetInterrupt() call is using very little CPU time waiting for an incoming interrupt to arrive. The main application thread could continue its work while interrupt handling occurs in the background.

Here is a simple function that can be spawned as a thread to implement a callback system.

```
void intrThread()
{
  scgtIntrHandle intrHandle = -1;
  uint32 numIntrRet = 0;
  uint32 done = 0;
  scgtInterrupt intrBuff[100];
  int i;

  /* initialize intrHandle */
  scgtGetInterrupt(&handle,
  &intrHandle, &intrBuff, 0, 0,
  &numIntrRet);

  while (!exitThread)
  {
    /* get up to 100 interrupts ...
    The number of interrupts to get
    should be tuned to your specific
    application. */

    scgtGetInterrupt(&handle,
    &intrHandle, &intrBuff, 100, 250,
    &numIntrRet)

    /* call the interrupt handling
    function for each interrupt */

    for (i = 0; i < numIntrRet; i++)
    {
      /* the interrupt can be filtered
      here to dispatch it to
      different functions based on
      type or source node etc. */
      handle_the_
      interrupt(&intrBuff[i]);
    }
  }
}
```

This method can also be used to preserve legacy software structures that used signals or callback functions for event notification. The interrupt thread can simply call the equivalent signal handler or callback function.

Conclusion

Network interrupts provide an efficient and practical way to send event and synchronization information across the network between processes. The SCRAMNet GT API provides simple and powerful mechanisms for using network interrupts, allowing the system designer flexibility when incorporating network interrupts into distributed systems. This efficiency and flexibility can be an enabling factor in many distributed system designs.