The GP messaging library

David Buscher
Revision: 1.1

Documentation ID: wht-naomi-9

Date: 1999/10/29 14:02:29

1 Introduction

The GP (general purpose) messaging library is a framework for sending arbitrary data packets between c40 signal processors and processes running on workstations. It is used for setting control parameters of the c40 real time (and non-real-time) system and also for sending diagnostic data from the real-time system to the diagnostic flight-recorder and thence to the workstations.

This document describes the library in a bottom-up way, i.e. it describes how it is implemented first and then how to use it at a higher level, because such an approach makes the explanation of various concepts easier, because the separations between the layers are not very well hidden in this low-level library.

2 Software structure

The software consists of an executable running on the host SPARC called `c40Comms` and a library of functions for other processes to communicate with it. One version of the library exists for the c40s (`libGPmsg_c40.a`) and another (`libGPmsg.a`) for the workstations. Although they share common features, there are differences in how they are used and implemented. The c40 version of the library deals with routing messages between c40s via commports. The `c40Comms` process looks to all the c40s as if it is just another c40, with cpu ID 0. The workstation library deals with communicating with the `c40Comms` process using the DTM protocol, and the `c40Comms` process forwards messages between the DTM ports and the c40 commports.

3 The GP message

The basic GP message is just a packet of 32-bit words which has a header and a body. The header provides the standardised parts of the message which the communications layer needs plus some additional fields which are commonly needed by higher-level functions. The body

is an application-specific sequence of 32-bit words, which can be up to `MAX_MSG_BODY_WORDS` i.e. 3300 words long. This value is slightly longer than the number of words to hold a 80x80 pixel frame at 16-bits per pixel.

Here is the definition of the header from `$STAGING/c40Comms/libsrc/gpmsg.h`:

```
typedef struct {
  int32 length;           /* Length of the entire packet in words - 1 */
  uint32 protocolID;       /* Synchronisation word */
  uint32 destAddress;      /* Destination machine + port address */
  int32 hopCount;         /* Incremented on every hop - used to trap
                             messages which never reach their destination */
  int32 command;           /* Command/acknowledge verb */
  int32 sequenceID;        /* Unique message tag for multiple messages of the
                             same type */
  uint32 replyAddress;     /* Reply-to machine + port address */
  int32 arg1;              /* Optional command argument - pads to 8 words */
} GPmsgHdr;
```

The `int32` and `uint32` identifiers are macros used to make sure that integers exactly 32 bits long are used on all platforms.

The `length` field is the length of the packet minus the length word itself. This was initially so that the protocol could be used with self-programming DMAs, but the convention has stuck although the implementation has not. There exist two macros to deal with this word, `GP_SET_BODY_LENGTH` and `GP_GET_BODY_LENGTH`. They deal with the off-by-one problem and also subtract/add the length of the rest of the header to the length of the body. Values of the packet length of less than the length of the header are invalid, whereas the body length may be zero or greater.

The `protocolID` word is a fixed word unlikely to occur in any normal bytestream - its value is defined as `0xe1ec23a1`. It acts as an identifier so that if packet synchronisation is lost, it can be regained by searching for this word in the bytestream.

The `destAddress` and `sourceAddress` are unique identifiers of the sending and receiving process. For a c40, the upper 16 bits of the address are always zero, and the lower 16 bits are the cpu ID, currently a number between 1 and 20. For a workstation process, the lower 16 bits are always zero, and the upper 16 bits are called the port number. This number is assigned by the `c40Comms` process to individual workstation processes using a method to be described later. The `destAddress` is the only required address: the `replyAddress` only makes sense if the sending process is expecting an acknowledge to the given message.

The `hopCount` field keeps a track of how many cpus have forwarded a given message. It is used to discard messages that end up in infinite loops.

The `command` field is a user-level field: the user can set it to any value, but normally it is set to the value of one of the enums in `c40Commands.h`. The basic action of the c40s is to enter and endless loop of reading GP messages and taking actions based on which enum is in the `command` field. New commands can be created by editing one of the files in `$STAGING/SharedInclude` and doing a `jam install` in that directory. Obviously some c40 code has to then be written to do something when that particular command is received.

By convention, any c40 receiving a command acknowledges it by sending a packet back to `replyAddress` with `command` set to the original command plus `GP_ACK` (decimal 1000 at the moment).

The `sequenceID` field has two possible functions, only one of which is actually used at the moment. The first is exclusively for real-time diagnostic packets, and indicates the real-time frame number of when the message was created. This is used for filtering and synchronising diagnostic data. The second possible use is to set it to a unique number for each message sent by a workstation client. The c40s normally send this word back unaltered to the client in an acknowledge packet, and so this can be used to ensure an exact send/acknowledge correspondence in the case where multiple requests are outstanding. In practice, no workstation program currently extant uses this feature.

The `arg1` field is used mostly to make the header a power-of-two long, to avoid alignment problems on 64-bit workstations. However, it has a number of subsidiary functions. The first is that it is conventional that all packets sent in acknowledgement to a workstation request have the `arg1` field set to zero if the request was successfully completed and non-zero if there was an error. A further convention is that if there was an error, then the body part of the acknowledge message is an error message, formatted in 32-bit c40-style characters. The python `rpc` function assumes this protocol, but nothing else enforces it at the moment. Some older c40 software uses the `arg1` field as the first argument to the command sent in a command request packet, with subsequent arguments in `body[0]`, `body[1]` etc. This is being phased out and the `body[0]` value is used for the first argument by most new c40 functions.

4 Communications

The basic function of the GP library is to to transport packets between the sending and receiving processes as best it can. It does not guarantee delivery of any packet, nor does it even guarantee an error message if the packet cannot be delivered. This is equivalent to the UDP level of functionality within the TCP/IP protocol (however, the workstation library actually uses the TCP transport layer for part of the message transport, so in fact it has slightly better reliability than the above definition suggests).

There exist two transport/physical layers, one using c40 comports and one using DTM messages. The c40Comms process acts as a bridge between these two. We now descrine these layers in turn.

5 C40 message transport

The c40s are connected to each other by commports and most of the LSI c40 boards have an interface called an LIA that maps a c40 commport onto the VME bus where it can be read and written by the Solaris host. Thus the c40 library is mostly concerned with transporting GP messages over commports.

In principle this is quite simple. A sending c40 looks at the `destAddress` field in the GP message header and uses this as an index into a table called the `routingTable`. This table tells the c40 which commport to send the message down to get the message to its

destination. The c40 writes the words in the packet down the appropriate commport in sequence, starting with the header. The receiving c40 reads the header words, determines how many body words are to follow and then reads the body words. By examining the destination address, it can determine whether the message is for the current node or is to be forwarded. With correctly configured routing tables, the message hops from c40 to c40 to eventually reach its destination.

5.1 GP and ISR interaction

This is a very hairy section. You can skip this on a first reading...

The major complication to this process is the fact that the commports need to be used both for GP traffic and for the "LoveTrains" which form the low-latency data-traffic in the real-time interrupt service routine (ISR). The first level of solution to this is that the commports, which are all bidirectional, are used in one direction only for real-time traffic. Interrupts on these comports are used to assure synchronous interrupts across the real-time system. The opposite direction is used for GP traffic.

This scheme works in principle, but has one major Achilles' heel: the commports are not truly bidirectional in that if a commport blocks because the other cpu is not ready to accept data, it also blocks traffic in the reverse direction. This can lead to a deadlock in the following manner: if a GP message causes a commport to block and subsequently the GP-receiving cpu receives an interrupt, then the GP-receiving cpu immediately stops processing GP messages and enters the ISR. It now tries to interrupt the GP-sending cpu by sending a "cow-catcher" word down the commport in the LoveTrain direction. Because the commport is blocked, the GP-sending cpu never receives the cow-catcher, and thus never enters the interrupt state, while the GP-receiving cpu never leaves it.

Ways of avoiding this deadlock have evolved over time and it is hoped that the current implementation is the final word on this. It works by ensuring that the GP message can never block, and this is done by using "handshake" words which travel in the opposite direction to the GP messages.

When a cpu is ready to receive a message (i.e. it has finished processing the previous messages), it sets up a DMA to automatically ready any words from the incoming commport directly into a memory buffer. The DMA can read up to the maximum number of words for a single message. The receiving cpu then sends a "handshake word" down the receive comport, i.e. in the LoveTrain direction.

The sending cpu's reaction depends on what state it is in. If the real-time interrupts are enabled, the word coming in the LoveTrain direction triggers an interrupt. The first thing the ISR does is to check if the value of the incoming word is the handshake value (set to -1) or the CowCatcher value (set to 13, for reasons of arbitrariness). If the former, the interrupt is immediately exited after incrementing a counter in shared memory, the `handshakeCounter`. If the latter, a normal real-time ISR is entered. When real-time interrupts are not enabled, the sending cpu polls for handshake words and increments the handshake counter. Once the handshake counter has been incremented, the sending cpu knows it can send one message. It does so using a write-DMA for efficiency. The write-DMA has to be switched off during the ISR because of a second type of deadlock that can occur if the LoveTrains cause a the port to block temporarily in the reverse direction.

The receiving cpu polls the reading DMA to see if a complete message has been received. It can tell this by seeing if the number of words received matches the message length as given by the message header. Once the message is ready, it stops the DMA and processes the message. It then restarts the DMA and sends a new handshake word to continue the process.

There is one final twist, that truly bidirectional GP flow is needed during the boot process, when insufficient cpus have been booted to form a complete one-way flow diagram. Thus there is a second method of sending GP messages across commports which is truly bidirectional which is only used at boot time, and which is replaced by the handshake method once all cpus are booted. This bidirectional method is also used on the LIAs at all times, for mostly historical reasons. The bidirectional method uses programmed writes and DMA reads, because bidirectional DMAs are a pain in the extreme. The switching between GP flow methods occurs when the cpus receive the appropriate routing tables from the workstation clients. Routing tables are discussed later in the document.

5.2 c40Comms

The c40Comms process appears to the c40s as another c40 connected to the commports of 4 of the cpus in an 8-cpu ring. It reads and writes words to the LIAs using the unix `read()` and `write()` calls on the special device files `/dev/liaXXX` which are serviced by device-drivers written by LSI. It uses a unix `select()` system call to avoid using cpu time when it is waiting for messages.

5.3 Shared DRAM implementation

The LSI dbv46 card does not have an LIA, but the highest volume of data traffic, the diagnostic data, goes via this board. It uses instead a unidirectional shared-memory fifo to send data from the on-board c40 to the c40Comms process. This fifo is resident in the shared VME memory on the dbv46 board and acts like a commport to all intents and purposes, except that all the hardware features are emulated in software. The c40Comms process has to poll the shared memory to see when messages are ready, which causes a slight cpu usage penalty.

6 Workstation message transport

GP messages from any process on any workstation can be sent to the c40s using the c40Comms process as an intermediary. Messages are sent to the c40Comms process using the DTM protocol, which defines a free-form ascii header and a binary data body. The GP message in its entirety is placed in binary form in the data part of the DTM message and sent to a well-known DTM port on the c40Comms process. The DTM header of this message is just the ascii string 'GP', to allow for future upgrades. The DTM protocol takes care of the byte-order of the binary GP message, so there should be no problems handling arbitrary 32-bit data on any machine where 32-bit floats and ints have the same byte order

(all machines in the known universe?). The c40Comms process examines the GP header and forwards it to the appropriate c40.

Messages traveling in the reverse direction need some additional setting up. All GP messages from the c40s to the workstations go to cpu 0, i.e. the c40Comms process. The c40Comms process then needs to decide which workstation process the message is intended for. It does this on the basis of the port id part of the `destAddress`.

NAME

BootServices — Standard functions available on all c40s.

SYNOPSIS

```
void BootServices
(
    GPmsgBuffer *buffer,
    ExcStatus *status
);
```

PARAMETERS

GPmsgBuffer *buffer
Received buffer to process.

ExcStatus *status
Inherited status.

DESCRIPTION

Callback for BOOT_CLASS messages.

SEE ALSO

GPaddCallback(tex), GPmainLoop(tex), GPacknowledge(tex), GPforwardMsg(tex), GPsetup(tex), GPgetMsg(tex), GPsendMsg(tex), ISRsendGPmsg(tex), GPgetHandshakePtr(tex)

NAME

ConvertFromC40String — Convert a character string from C40 format, i.

SYNOPSIS

```
int ConvertFromC40String
(
    int *wordString,
    char *charString,
    int maxChar,
    ExcStatus *status
);
```

PARAMETERS

int *wordString
Input string, null terminated.

char *charString
Output string.

int maxChar
Maximum number of characters to convert, including terminating null.

ExcStatus *status
Inherited status.

DESCRIPTION

E. 32-bit chars. The output string is always null-terminated.

RETURNS

Number of characters converted, including the terminating null.

SEE ALSO

ConvertToC40String(tex)

NAME

ConvertToC40String — Convert a character string to C40 format, i.

SYNOPSIS

```
int ConvertToC40String
(
    char *charString,
    int *wordString,
    int maxChar,
    ExcStatus *status
);
```

PARAMETERS

char *charString
Input string, null terminated.

int *wordString
Output string.

int maxChar
Maximum number of characters to convert, including terminating null.

ExcStatus *status
Inherited status.

DESCRIPTION

E. 32-bit chars. The output string is always null-terminated.

RETURNS

Number of characters converted, including the terminating null.

SEE ALSO

ConvertFromC40String(tex)

NAME

DiagPackFloat — Pack a vector of floats into a GP packed-float format

SYNOPSIS

```
int DiagPackFloat
(
    float *in,
    int inSize,
    int exponent,
    int *outBuffer,
    ExcStatus *status
);
```

PARAMETERS

float *in
Vector of floats.

int inSize
Size of vector.

int exponent
Divide data by 2êxponent before pack.

int *outBuffer
Output buffer.

ExcStatus *status
Exception status.

DESCRIPTION

Pack a vector of floats into a GP packed-float format.

RETURNS

The size in 32-bit words of the packed message.

NOTE

The vector size must be even, or the results are undefined.

SEE ALSO

DiagPackInt(tex)

NAME

DiagPackInt — Pack a vector of ints into a GP packed-float format

SYNOPSIS

```
int DiagPackInt
(
    int *in,
    int inSize,
    int exponent,
    int *outBuffer,
    ExcStatus *status
);
```

PARAMETERS

int *in
Vector of ints.

int inSize
Size of vector.

int exponent
Divide data by 2êxponent before pack.

int *outBuffer
Output buffer.

ExcStatus *status
Exception status.

DESCRIPTION

This routine is provided so that int data can be sent in the same format as float data, without the overhead of converting to floats first.

Clearly, on the receiving side, these ints will be converted into floats, but this is not an issue given that the dynamic range of the format is only 16 bits, so the allowable values will always fit into a 24-bit float.

RETURNS

The size in 32-bit words of the packed message.

NOTE

The vector size must be even, or the results are undefined.

SEE ALSO

DiagPackFloat(tex)

NAME

GPacknowledge — Acknowledge a GP message.

SYNOPSIS

```
void GPacknowledge
(
    GPmsgBuffer *buffer,
    ExcStatus *status
);
```

PARAMETERS

GPmsgBuffer *buffer
Partially completed reply.

ExcStatus *status
Exception status.

DESCRIPTION

Expects a reply message in buffer or an error status in status. It formulates a reply buffer and sends it back to the workstation client.

If there is no error condition set in status, buffer->header.arg1 is set to zero to indicate success and the message is sent to the originating address (buffer->header.replyAddress).

If an error condition in set in status, buffer->header.arg1 is set to -1, and the error string in status to be copied into the reply buffer. The reply buffer is sent to the original sender and also to PORT_STDERR.

If no message is to be sent back, there must be no error condition set in status, and buffer->header.length must be set to an invalid value, i.e. Less than MSG_HDR_WORDS.

Buffer is freed irrespective of status.

NOTE

This routine has ignores the status convention. It also clears status on return.

SEE ALSO

GPaddCallback(tex), GPmainLoop(tex), BootServices(tex), GPforwardMsg(tex), GPsetup(tex), GPgetMsg(tex), GPsendMsg(tex), ISRsendGPmsg(tex), GPgetHandshakePtr(tex)

NAME

GPaddCallback — Add/replace a message handler to the list of message handlers

SYNOPSIS

```
void GPaddCallback
(
    int command,
    void (*callback)(GPmsgBuffer *buffer,ExcStatus *status),
    ExcStatus *status
);
```

PARAMETERS

int command
Command/class enum from c40Commands.h.

void (*callback)(GPmsgBuffer *buffer,ExcStatus *status)
Message callback function.

ExcStatus *status
Exception status.

DESCRIPTION

The callback will be called from GPmainLoop(3) with with the buffer argument containing the message which triggered the callback, and holding on return a reply message, if any (see GPacknowledge(3)).

If command is a specific command enum, only messages with that command in the header will trigger the callback. If command is a message class enum, all messages with command enums within that class will trigger a callback to the supplied function, providing a specific callback for that command enum has not also been registered.

SEE ALSO

GPmainLoop(tex), GPacknowledge(tex), BootServices(tex), GPforwardMsg(tex), GPsetup(tex), GPgetMsg(tex), GPsendMsg(tex), ISRsendGPmsg(tex), GPgetHandshakePtr(tex)

NAME

GPallocBuffer — Allocate a buffer for a general-purpose message.

SYNOPSIS

```
GPmsgBuffer *GPallocBuffer(ExcStatus *status);
```

PARAMETERS

ExcStatus *status
Inherited status.

DESCRIPTION

This routine allocates a buffer containing a message body (maximum size) and its associated header, plus extra pointers to allow easy use in linked lists.

NOTES

Use this routine in preference to malloc(3) because it uses the same pool of buffers used by the GP message system. If you use malloc(3), this pool will fill up when you use GPsendMsg(3), which automatically does a GPfreeBuffer(3).

Returns a pointer to the new buffer, or NULL if no free buffers are left.

SEE ALSO

GPfreeBuffer(tex)

NAME

GPcloseReturnPort — Close the current GP reply channel

SYNOPSIS

```
void GPcloseReturnPort(void);
```

DESCRIPTION

Sends a UNMAP_GP_DTM message to the server where appropriate and destroys the DTM port, where possible.

SEE ALSO

GPsetup(tex), GPshutdown(tex), GPopenServerPort(tex), GPcloseServerPort(tex), GPgetReturnDTMport(tex), GPgetReturnAddress(tex), GPopenReturnPort(tex), GPforwardMsg(tex), GPsendMsg(tex), GPgetMsg(tex)

NAME

GPcloseServerPort — Closes the DTM port to the GP message server.

SYNOPSIS

```
void GPcloseServerPort(void);
```

DESCRIPTION

Closes the DTM port to the GP message server.

SEE ALSO

GPsetup(tex), GPshutdown(tex), GPopenServerPort(tex), GPgetReturnDTMport(tex), GPgetReturnAddress(tex), GPopenReturnPort(tex), GPcloseReturnPort(tex), GPforwardMsg(tex), GPsendMsg(tex), GPgetMsg(tex)

NAME

GPdequeue — Dequeue a message: take it off the head

SYNOPSIS

```
GPmsgBuffer *GPdequeue
(
    GPmsgQueue *queue,
    ExcStatus *status
);
```

PARAMETERS

GPmsgQueue *queue
Queue to take message from.

ExcStatus *status
Inherited status.

DESCRIPTION

Dequeue a message: take it off the head.

RETURNS

NULL if the queue is empty, otherwise returns a pointer to the head. Does *not* set status if the queue is empty, since this is not an exception, but rather the only way to tell if the queue is empty.

SEE ALSO

GPnewQueue(tex), GPdestroyQueue(tex), GPsetQueueSize(tex), GPenqueue(tex)

NAME

GPdestroyQueue — Free space allocated for a queue and free any buffers on it

SYNOPSIS

```
void GPdestroyQueue
(
    GPmsgQueue **queue,
    ExcStatus *status
);
```

PARAMETERS

GPmsgQueue **queue
Pointer to queue pointer; on return this is set to NULL.

ExcStatus *status
Exception status.

DESCRIPTION

Free space allocated for a queue and free any buffers on it.

SEE ALSO

GPnewQueue(tex), GPsetQueueSize(tex), GPenqueue(tex), GPdequeue(tex)

NAME

GPenqueue — add it to the tail of a queue

SYNOPSIS

```
void GPenqueue
(
    GPmsgBuffer *buffer,
    GPmsgQueue *queue,
    ExcStatus *status
);
```

PARAMETERS

GPmsgBuffer *buffer
: New buffer to add.

GPmsgQueue *queue
: Queue to add message buffer to.

ExcStatus *status
: Inherited status.

DESCRIPTION

If adding a message to the queue would cause it to exceed its maximum length, it is silently sent to GPfreeBuffer(3). GPfreeBuffer calls this function, so it must be careful...

SEE ALSO

GPnewQueue(tex), GPdestroyQueue(tex), GPsetQueueSize(tex), GPdequeue(tex)

NAME

GPforwardMsg — Queue a *completed* message buffer for transmission.

SYNOPSIS

```
GPmsgBuffer *GPforwardMsg
(
    GPmsgBuffer *buffer,
    ExcStatus *status
);
```

PARAMETERS

GPmsgBuffer *buffer
Message to send. Must have been allocated with GPallocBuffer(3).

ExcStatus *status
Exception status.

DESCRIPTION

The message buffer must be complete including header. In some implementations, this function queues the packet to be sent, and.

RETURNS

Immediately. This is in keeping with the philosophy of the packet transport layer in that it does not guarantee delivery. The buffer is freed once the message has been successfully transmitted. For the above reasons, no reference to the packet contents should be made after the function returns successfully. Returns a NULL buffer pointer if the buffer will be freed by the sending routine, or a pointer to the input buffer if it could not be sent and was not destroyed. This simplifies calls to GPfreeBuffer(3).

SEE ALSO

GPsetup(tex), GPshutdown(tex), GPopenServerPort(tex), GPcloseServerPort(tex), GPgetReturnDTMport(tex), GPgetReturnAddress(tex), GPopenReturnPort(tex), GPcloseReturnPort(tex), GPsendMsg(tex), GPgetMsg(tex)

NAME

GPfreeBuffer — Free a used GP message buffer.

SYNOPSIS

```
GPmsgBuffer *GPfreeBuffer
(
    GPmsgBuffer *buffer,
    ExcStatus *status
);
```

PARAMETERS

GPmsgBuffer *buffer
Buffer to release.

ExcStatus *status
Inherited status.

DESCRIPTION

Release a buffer returned by GPallocBuffer(3) or GPgetMsg(3).

RETURNS

NULL if successful.

NOTES

Do not use free(3) to release such a buffer since the buffer must be returned to the pool of free buffers, not the general memory pool.

This routine ignores the status value. If buffer is NULL, as it might be if status was set on attempting to allocate the buffer, does nothing.

Buffers freed in this way will be used to replenish the list for allocation by interrupt service routines and to replenish a pool of free buffers for use in GPallocBuffer(3).

Returns a NULL pointer to assign to the freed pointer (can simplify pointer bookkeeping).

SEE ALSO

GPallocBuffer(tex)

NAME

GPgetHandshakePtr — Get a pointer to the counter which is incremented when a handshake word

SYNOPSIS

```
int *GPgetHandshakePtr
(
    int portNum,
    ExcStatus *status
);
```

PARAMETERS

int portNum
Port number to get handshake pointer from.

ExcStatus *status
Exception state.

DESCRIPTION

Arrives.

SEE ALSO

GPaddCallback(tex), GPmainLoop(tex), GPacknowledge(tex), BootServices(tex), GPforwardMsg(tex), GPsetup(tex), GPgetMsg(tex), GPsendMsg(tex), ISRsendGPmsg(tex)

NAME

GPgetMsg — Get the next general-purpose message from the incoming message queue.

SYNOPSIS

```
GPmsgBuffer *GPgetMsg
(
    int timeout,
    ExcStatus *status
);
```

PARAMETERS

int timeout
Timeout in milliseconds. Set to -1 to block indefinitely.

ExcStatus *status
Exception status.

DESCRIPTION

This function returns the next available incoming message on the input port set by GPopenReturnPort(3). If no messages are available it waits for timeout milliseconds for a message. If the timeout is set to -1, the function blocks indefinitely. If timeout is set to 0, the function returns immediately if no messages are waiting.

RETURNS

A pointer to a message buffer which should be freed with GPfreeBuffer(3) when no longer needed. The size of the received message can be derived from the length field of the message header. If no messages are waiting or arrive within the timeout interval, returns a null pointer.

NOTE

The timeout facility is not available on all architectures. In this case, all timeout values apart from zero behave the same as a value of -1, i.e. If no messages are waiting the function blocks indefinitely.

SEE ALSO

GPsetup(tex), GPshutdown(tex), GPopenServerPort(tex), GPcloseServerPort(tex), GPgetReturnDTMport(tex), GPgetReturnAddress(tex), GPopenReturnPort(tex), GPcloseReturnPort(tex), GPforwardMsg(tex), GPsendMsg(tex)

NAME

GPgetReturnAddress — Get our own address in GP space.

SYNOPSIS

```
unsigned int GPgetReturnAddress(ExcStatus *status);
```

PARAMETERS

ExcStatus *status
Not Documented.

DESCRIPTION

The address is a machine:port pair encoded into an integer.

SEE ALSO

GPsetup(tex), GPshutdown(tex), GPopenServerPort(tex), GPcloseServerPort(tex), GPgetReturnDTMport(tex), GPopenReturnPort(tex), GPcloseReturnPort(tex), GPforwardMsg(tex), GPsendMsg(tex), GPgetMsg(tex)

NAME

GPgetReturnDTMport — Get the handle for the DTM return port for GP messages

SYNOPSIS

```
int GPgetReturnDTMport(ExcStatus *status);
```

PARAMETERS

ExcStatus *status
Not Documented.

DESCRIPTION

This is useful for applying DTM functions such as DtmSelectRead() or DtmAddXinput() to the GP message stream.

SEE ALSO

GPsetup(tex), GPshutdown(tex), GPopenServerPort(tex), GPcloseServerPort(tex), GPgetReturnAddress(tex), GPopenReturnPort(tex), GPcloseReturnPort(tex), GPforwardMsg(tex), GPsendMsg(tex), GPgetMsg(tex)

NAME

GPmainLoop — Event loop for handling GP messages

SYNOPSIS

```
void GPmainLoop(ExcStatus *status);
```

PARAMETERS

ExcStatus *status
Not Documented.

DESCRIPTION

Listens for GP messages and dispatches the appropriate callback (as set up by GPaddCallback(3)). On return from the callback, calls GPacknowledge(3) with the returned message buffer.

SEE ALSO

GPaddCallback(tex), GPacknowledge(tex), BootServices(tex), GPforwardMsg(tex), GPsetup(tex), GPgetMsg(tex), GPsendMsg(tex), ISRsendGPmsg(tex), GPgetHandshakePtr(tex)

NAME

GPnewQueue — Create and initialise a new message queue

SYNOPSIS

```
GPmsgQueue *GPnewQueue(ExcStatus *status);
```

PARAMETERS

ExcStatus *status
Exception status.

DESCRIPTION

Create and initialise a new message queue.

RETURNS

A pointer to the allocated queue structure.

SEE ALSO

GPdestroyQueue(tex), GPsetQueueSize(tex), GPenqueue(tex), GPdequeue(tex)

NAME

GPopenReturnPort — Opens an input port from the C40 server.

SYNOPSIS

```
int GPopenReturnPort
(
    int GPport,
    ExcStatus *status
);
```

PARAMETERS

int GPport
GP port number.

ExcStatus *status
Exception status.

DESCRIPTION

If DTMport is PORT_USER, a free port is requested and returned from the c40 comms server, otherwise the numbered "well known" DTM port is opened.

RETURNS

The GP address actually connected or -1 on error.

SEE ALSO

GPsetup(tex), GPshutdown(tex), GPopenServerPort(tex), GPcloseServerPort(tex), GPgetReturnDTMport(tex), GPgetReturnAddress(tex), GPcloseReturnPort(tex), GPforwardMsg(tex), GPsendMsg(tex), GPgetMsg(tex)

NAME

GPopenServerPort — Opens the DTM port to the GP message server.

SYNOPSIS

```
void GPopenServerPort(ExcStatus *status);
```

PARAMETERS

ExcStatus *status
Not Documented.

DESCRIPTION

Opens the DTM port to the GP message server.

SEE ALSO

GPsetup(tex), GPshutdown(tex), GPcloseServerPort(tex), GPgetReturnDTMport(tex), GPgetReturnAddress(tex), GPopenReturnPort(tex), GPcloseReturnPort(tex), GPforwardMsg(tex), GPsendMsg(tex), GPgetMsg(tex)

NAME

GPsendMsg — Send a general-purpose message to a given destination.

SYNOPSIS

```
GPmsgBuffer *GPsendMsg
(
    GPmsgBuffer *buffer,
    int bodyLength,
    int destAddress,
    ExcStatus *status
);
```

PARAMETERS

GPmsgBuffer *buffer
Message to send. Must have been allocated with GPallocBuffer(3).

int bodyLength
Number of words in body.

int destAddress
Destination machine/port.

ExcStatus *status
Exception status.

DESCRIPTION

Takes a message buffer, fills in the message header and sends the resulting packet using GPforwardMsg(3).

RETURNS

A NULL buffer pointer if the buffer will be freed by the sending routine, or a pointer to the input buffer if it could not be sent and was not destroyed. This simplifies calls to GPfreeBuffer(3).

SEE ALSO

GPsetup(tex), GPshutdown(tex), GPopenServerPort(tex), GPcloseServerPort(tex), GPgetReturnDTMport(tex), GPgetReturnAddress(tex), GPopenReturnPort(tex), GPcloseReturnPort(tex), GPforwardMsg(tex), GPgetMsg(tex)

NAME

GPsetQueueSize — Change the maximum number of entries in a queue.

SYNOPSIS

```
void GPsetQueueSize
(
    GPmsgQueue *queue,
    int newLength,
    ExcStatus *status
);
```

PARAMETERS

GPmsgQueue *queue
Queue to change length.

int newLength
Length to set queue to.

ExcStatus *status
Exception status.

DESCRIPTION

Invalid arguments fail silently.

SEE ALSO

GPnewQueue(tex), GPdestroyQueue(tex), GPenqueue(tex), GPdequeue(tex)

NAME

GPsetup — Initialise the GP communications system

SYNOPSIS

```
void GPsetup(ExcStatus *status);
```

PARAMETERS

ExcStatus *status
Exception status.

DESCRIPTION

Initialise the GP communications system.

SEE ALSO

GPshutdown(tex), GPopenServerPort(tex), GPcloseServerPort(tex), GPgetReturnDTMport(tex), GPgetReturnAddress(tex), GPopenReturnPort(tex), GPcloseReturnPort(tex), GPforwardMsg(tex), GPsendMsg(tex), GPgetMsg(tex)

NAME

GPshutdown — Close down the GP communications system

SYNOPSIS

```
void GPshutdown(void);
```

DESCRIPTION

Close down the GP communications system.

SEE ALSO

GPsetup(tex), GPopenServerPort(tex), GPcloseServerPort(tex), GPgetReturnDTMport(tex), GPgetReturnAddress(tex), GPopenReturnPort(tex), GPcloseReturnPort(tex), GPforwardMsg(tex), GPsendMsg(tex), GPgetMsg(tex)

NAME

ISRsendGPmsg — Send a GP message from an interrupt service routine (ISR).

SYNOPSIS

```
void ISRsendGPmsg
(
    GPmsgBuffer *buffer,
    int bodyLength,
    int destAddress
);
```

PARAMETERS

GPmsgBuffer *buffer
: Message to send. Must have been allocated with GPallocBuffer(3).

int bodyLength
: Number of words in body.

int destAddress
: Destination machine/port.

DESCRIPTION

Takes a message buffer, fills in the message header and queues the resulting packet for the attention of the GP system running in the non-ISR portion of the code. Thus it avoids non-reentrant code invoked in the standard GPsendMessage(3).

The buffer is freed once the message has been successfully transmitted.

Does not use the standard status system because it is of limited use in an ISR.

SEE ALSO

GPaddCallback(tex), GPmainLoop(tex), GPacknowledge(tex), BootServices(tex), GPforwardMsg(tex), GPsetup(tex), GPgetMsg(tex), GPsendMsg(tex), GPgetHandshakePtr(tex)