

DataKinetics

**System7
Software Environment
Programmer's Manual**

Revision History

ISSUE	DATE	CHANGES
1	18-Jul-95	
2	19-Feb-97	All commands in system.txt now commence with a key word. Definition of MSG made consistent with actual header file.
3	29-May-99	Description of module instance and associated library functions added. Table of default module identifiers added. Module identifiers and message types reserved for user's applications added.

IMPORTANT INFORMATION

The information in this manual is supplied without warranty as to its accuracy. DataKinetics is not responsible or liable for any loss or damage of whatever kind arising from the use of the supporting software and documentation.

©1995-1999 DataKinetics Ltd. All rights reserved. No part of this publication or the associated software may be reproduced, stored in a retrieval system or transmitted in any form or by any means without the prior written permission of DataKinetics Ltd.

Document reference: U10SSS

Contents

1 INTRODUCTION	4
2 BASIC CONCEPTS	5
2.1 Modules.....	5
2.2 Module Identifiers.....	5
2.3 Messages.....	6
2.4 Message Queues	6
2.5 Distributed Modules	7
2.6 Library Functions.....	8
3 LIBRARY FUNCTIONS.....	10
3.1 GCT_send.....	10
3.2 GCT_receive	11
3.3 GCT_grab	12
3.4 GCT_set_instance	13
3.5 GCT_get_instance	14
3.6 getm	15
3.7 relm	16
4 SYSTEM INITIALISATION	17
4.1 Introduction	17
4.2 The gctload Program.....	17
4.3 System Configuration File	18
4.3.1 Create a message queue for a Local module.....	19
4.3.2 Re-direct messages to another module.....	20
4.3.3 Spawn a process	21
4.3.4 Example Configuration File.....	22
5 OPERATING SYSTEM VARIATIONS	23
5.1 SCO Unix	23
5.2 Solaris	23
5.3 QNX	24
5.4 Windows NT.....	24
5.5 LINUX.....	25
5.6 VxWorks.....	25
APPENDIX A : MESSAGE FORMAT	26
A.1 MSG Message Structure.....	26
A.2 Header Fields	26
A.3 Parameter Field	27
APPENDIX B: DEFAULT MODULE IDENTIFIERS	28
APPENDIX C: VALUES RESERVED FOR CUSTOM USE.....	29
C.1 Reserved module identifiers	29
C.2 Reserved message types	29

1 INTRODUCTION

The DataKinetics System7 product range comprises a number of portable software modules for the realisation of Signalling System Number 7 (SS7) protocol stacks. The System7 architecture is multi-tasking, using message passing to communicate between tasks.

Each module in the system is implemented as a separate task within the chosen operating environment. A module implements either a layer within the protocol stack, a user part or some other functional entity within the system. In general, a module supports multiple internal instances within a single process (for example multiple links, multiple circuits or multiple transactions are each handled by a single process). The architecture supports multi-processor operation with modules being distributed between different processors.

For software portability, each module makes a minimum demand on the host operating system. Most modules require just 7 functions to be provided by the operating system, these are required for inter-process communication and memory allocation. This approach makes the software easy to port to different platforms and operating environments. In addition, the use of message passing between tasks, and therefore protocol layers, means that it is possible for different layers to run on different processors if required.

All that is required to port the System7 product to a new environment is an implementation of the 7 library functions tailored to the chosen operating system. These functions are available as standard products for a number of operating systems.

This document describes the operating environment for the System7 protocol modules and presents a utility program - **gctload** - which automatically creates and initialises the required environment for a number of operating systems.

2 BASIC CONCEPTS

This chapter introduces the basic System7 concepts and terminology that will be used throughout the remainder of the manual.

2.1 Modules

A module is an implementation of a particular layer in the protocol stack (eg. MTP2, MTP3), a particular user part (eg. ISUP, SCCP) or a collection of other functionality which fits together as a logical entity. A module may be part of the System7 product range or a User-supplied module.

Each module in the system runs as a separate task, process or program (depending on the type of operating system). The module is identified by a **Module Identifier** and communicates with other modules in the system by sending **Messages** to a **Message Queue** belonging to the destination module. A set of **Library Functions** is the only operating system specific code used by a module.

A module handles multiple internal instances of the functional entity associated with the module (eg. MTP2 module handles multiple signalling links, the MTP3 module handles multiple link sets and multiple routes, the ISUP module handles multiple circuits).

2.2 Module Identifiers

Each module has a module identifier (`module_id`) which is a logical number in the range 0 to 255. It is used to identify modules within the system for the purposes of inter-process communication. To send a message to another process the sending module uses the module identifier of the destination process. To receive a message from a module's own message queue it uses it's own module identifier.

Some modules operate with a fixed module identifier whilst others allow the module identifier to be specified at run-time.

The module identifiers of other modules with which a module will communicate is usually a run-time configuration option.

The module identifier is a logical value, it is not the same as the `task_id` or process id (`pid`) which is usually allocated automatically by the operating system when a process is created.

The inter-process communication mechanism usually uses the module identifier as an index to an array of message queue pointers to provide an efficient mechanism to quickly access the correct message queue. This feature can also be used to allow messages destined for a particular module to be re-directed to an alternative message queue belonging to another module.

The re direction mechanism is used when messages need to be transferred to another board in the system. Messages for all processes that run on the other board are redirected to a special local module that handles inter-board message passing. Other modules within the system do not need to know whether the modules with which they communicate are running locally or not.

A list of default module identifiers used by the System7 software is given in Appendix B.

2.3 Messages

Modules communicate by sending messages to other modules in the system. There are three types of message structure used within the System7 protocol software (MSG, T_FRAME and R_FRAME). However T_FRAMEs and R_FRAMEs are only used on embedded processors by modules running on the same processor as the physical interface. For this reason this document refers in detail only to the use of the MSG message structure.

The MSG message is a 'C' structure containing a fixed format header field (which is common also with T_FRAMEs and R_FRAMEs) and a buffer for variable length parameter data. A detailed description of the message structure is given in Appendix A.

The message header contains a message **type** field that serves to identify the meaning of the message and the format of the variable parameter area. The **id** field identifies to which internal instance of the entity handled by the module the message applies (eg the link id or the circuit id etc). The **src** and **dst** fields are the source and destination module identifiers which must be entered by the sending module prior to sending the message. The **rsp_req** field is used by the sending module to solicit confirmation that the message has been processed by the destination process. When a confirmation is requested, the destination module (after finishing processing the message) enters a value in the **status** field of the message (usually zero to imply success or non-zero otherwise), modifies the message type by clearing bit 14, and sends the message back to the source module.

2.4 Message Queues

Each module in the system has a single message queue that is used by other modules to send messages to the module. A message queue is a system buffer which stores messages (usually by reference) in first-in, first-out order.

Messages are read out of the message queue by the receiving module which typically waits until there is a message available and reads it, it then processes the message and waits until the next message is available. All input to the module is through its message queue.

2.5 Distributed Modules

Some systems require the functional entity implemented by a module to be distributed across several processors in a system. For example a module may run on several separate cards in a single computer, each card interacting with a single module running on the computer. Alternatively a user's application may be distributed across several host computers where each host interacts with a protocol module running on a single protocol server.

In both cases there is a one to many relationship between the distributed processors and the adjacent layer in the protocol stack. There is a clear requirement for the single module to be able to determine which of the distributed processors a message has been received from and to which of the distributed processors a message should be sent to. This is achieved using the concept of a module **Instance**.

The module **Instance** is a number in the range 0 to one less than the number of distributed processors. The module instance is used by the inter-board message passing process to determine which board to send the message to. When messages are received from other boards the inter-board message passing process inserts the module instance of the board from which the message was received.

The module instance is not directly accessible as a field in the message, instead the following procedures making use of the functions `GCT_set_instance` and `GCT_get_instance` are adopted:

When a module needs to send a message to an instance of a module running on one of a number of separate processors, prior to sending the message it must call the function (`GCT_set_instance`) to write the module **Instance** into the message.

When a module receives a message and needs to determine which Instance of a distributed process it came from it calls another function (`GCT_get_instance`) to extract the **Instance** number from the message.

By default the instance number is preset to zero.

2.6 Library Functions

The protocol modules are written in a portable manner and assume the existence of the following minimal set of 'system' library functions:

GCT_send	Function to send a message to another module.
GCT_receive	Function to receive a message from a module's own message queue. The function does not return until a message is ready.
GCT_grab	Function to receive a message from a module's own message queue. This function returns straight away if there are no messages ready.
GCT_set_instance	Function to insert the module instance into a message.
GCT_get_instance	Function to extract the module instance from a message.
GCT_getmem	Function to allocate a buffer from a memory partition.
GCT_relmem	Function to release a buffer back to the memory partition.

The functions GCT_getmem and GCT_relmem apply to all 3 message structures (ie MSG, T_FRAME and R_FRAME) and are accessed by 4 higher level library functions (which are supplied and do not need to be implemented in a system-specific manner). The higher level functions are as follows:

getm	Function to allocate a MSG.
gett	Function to allocate a T_FRAME (not discussed in this manual)
getr	Function to allocate a R_FRAME (not discussed in this manual)
relm	Function to release MSG, T_FRAME or R_FRAME back to the system.

Consequently only the following functions are of interest to the user:

GCT_send
GCT_receive
GCT_grab
GCT_set_instance
GCT_get_instance
getm
relm

The syntax for each of these functions is described in the following chapter. Their usage is described below:

A module wishing to send a message to another module will first allocate an MSG structure using the **getm** function. At this stage it is necessary to decide whether or not a confirmation message will be required and initialise the `rsp_req` field accordingly. Once all the message parameters have been entered into the MSG the module calls **GCT_send** to send the message to the destination module. If the `GCT_send` function fails to send the message the sending module must release the message back to the system using the `relm` function (although this will only happen when the system is incorrectly configured). When multiple destination processors are used the module sending the message must call **GCT_set_instance** prior to calling `GCT_send` in order to write the destination module instance into the message.

The destination module will receive the message from its own message queue using either the **GCT_receive** or **GCT_grab** functions (depending on whether it wishes to block or not if no messages are available). It then processes the message. When multiple source processors are used the module receiving the message should call **GCT_get_instance** after calling `GCT_receive` or `GCT_grab` in order to read the source module instance from the message.

When the receiving module has finished processing the message it carries out one of two possible courses of action depending on whether or not a confirmation is required. If no confirmation is required then the message is released back to the system using the `relm` function. If a confirmation is required then a status value is written into the message header, the message type is changed (bit 14 is cleared) and the message is sent back to the original sending module using the **GCT_send** function. On receipt of the confirmation the original sending module (after inspecting the status) releases the message back to the system using the `relm` function.

In this way it is ensured that each message will eventually always be released back to the system.

3 LIBRARY FUNCTIONS

3.1 GCT_send

Synopsis:

Function to send the message pointed to by **h** to the specified **module_id**.

Prototype:

```
int GCT_send(unsigned int module_id, HDR *h);
```

Return Value:

Returns zero on success, non-zero otherwise.

If the function does not return success then the calling program must release the message back to the system using `reim()`.

Parameters:

module_id - The destination module id. This will usually be the same as the value contained in the **hdr.dst** field of the message.

h - A pointer to the HDR structure at the start of the MSG to be sent. This parameter should always point to a buffer allocated using `getm()`.

Description:

This function uses **module_id** to determine which message queue the message should be sent to and sends the message. A success return value implies that the message has been sent to the message queue belonging to the destination process.

3.2 GCT_receive

Synopsis:

Function to wait until the next message for **module_id** is available and return a pointer to the message.

Prototype:

```
HDR *GCT_receive(unsigned int module_id);
```

Return Value:

A pointer to the received message on success, 0 on failure.

Parameters:

module_id - The module's own module id.

Description:

This function uses **module_id** to determine which message queue to receive from. If the message queue contains a message then a pointer to the first message is returned. Otherwise the function suspends the calling task until a message is available.

After processing, the message returned by the GCT_receive function must either sent back to the sending module (as a confirmation message) or released back to the system using relm().

The only difference between GCT_receive and GCT_grab is whether to block or not when no messages are available.

3.3 GCT_grab

Synopsis:

Function to determine whether there is a message ready for **module_id** and return a pointer to the message. If no message is ready then the function returns immediately.

Prototype:

```
HDR *GCT_grab(unsigned int module_id);
```

Return Value:

A pointer to the received message on success or 0 if there are no messages waiting.

Parameters:

module_id - The module's own module id.

Description:

This function uses **module_id** to determine which message queue to receive from. If the message queue contains a message then a pointer to the first message is returned. Otherwise the function immediately returns zero.

After processing, the message returned by the GCT_grab function must either sent back to the sending module (as a confirmation message) or released back to the system using relm().

The only difference between GCT_receive and GCT_grab is whether to block or not when no messages are available.

3.4 GCT_set_instance

Synopsis:

Function to write the module **instance** into the message pointed to by **h**.

Prototype:

```
int GCT_set_instance(unsigned int instance, HDR *h);
```

Return Value:

Returns zero on success, non-zero otherwise (currently no failure conditions are defined).

Parameters:

instance - The destination module instance.

h - A pointer to the HDR structure at the start of the MSG.

Description:

Writes the destination module instance into the message. This function should be called prior to calling GCT_send by the module sending the message.

The destination module instance is used when messages are sent from one processor to another processor. It determines which destination processor the message is sent to.

Examples of the use of this function are as follows:

a) When sending messages to one of several boards. In this case the module instance is the board_id.

b) When sending messages to one or other Signalling Interface Unit (SIU) from an SIU pair. In this case the module instance is 0 (SIUA) or 1 (SIUB).

3.5 GCT_get_instance

Synopsis:

Function to recover the module instance from the message pointed to by **h**.

Prototype:

```
unsigned int GCT_get_instance(HDR *h);
```

Return Value:

Returns the module instance read from the message.

Parameters:

h - A pointer to the HDR structure at the start of the MSG.

Description:

Recovers the source module instance from a received message. This function should be called after return from GCT_receive or GCT_grab.

The source module instance is used when messages are received from a number of processors by the local module. It identifies the source processor at which the message originated.

Examples of the use of this function are as follows:

a) When receiving messages from one of several boards. In this case the module instance is the board_id.

b) When receiving messages from one or other Signalling Interface Unit (SIU) in an SIU pair. In this case the module instance is 0 (SIUA) or 1 (SIUB).

3.6 getm

Synopsis:

Function to allocate an MSG and initialise given fields in the message header.

Prototype:

```
MSG *getm(unsigned short type, unsigned short id,  
          unsigned short rsp_req, unsigned short len);
```

Return Value:

A pointer to the allocated message buffer message or 0 if no buffers available.

Parameters:

type - The message type, this is written to the **hdr.type** field of the message before the function returns.

id - The id value, this is written to the **hdr.id** field of the message before the function returns.

rsp_req - The rsp_req value, this is written to the **hdr.rsp_req** field of the message before the function returns.

len - The number of bytes that will be used in the parameter area of the message. This is written to the **len** field of the message before the function returns.

Description:

This function allocates a message buffer from the buffer pool and initialises the type, id, rsp_req and len fields of the message to the specified values.

The function is used to allocate a message for subsequent inter-process communication where it will be sent to the destination process. On return from the function it is the calling functions responsibility to initialise the **hdr.src**, and **hdr.dst** fields and the parameter area of the message prior to calling GCT_send().

3.7 relm

Synopsis:

Function to release a message that has previously been allocated by either `getm()`, `gett()` or `getr()` back to the system.

Prototype:

```
int relm(HDR *h);
```

Return Value:

Zero on success; non-zero otherwise.

Parameters:

h - Pointer to the message buffer to be released.

Description:

Returns a message buffer allocated by `getm()`, `gett()` or `getr()` to the system buffer pool.

Each message allocated must be returned once (and only once) to the system. It does not need to be returned by the same process that allocated it.

Note that because this function can be used for `MSG`, `T_FRAME` and `R_FRAME` messages it is necessary to cast the pointer to the message to type `HDR`.

4 SYSTEM INITIALISATION

4.1 Introduction

System initialisation requires first that a pool of message buffers is created for subsequent inter-process communication. Secondly, that a message queue is created for each module that will run and that any message re-direction for modules that are running remotely is initialised. Then all process can be started.

A program **gctload** exists to handle this initialisation sequence. It reads input from a text file called system.txt, carries out all system initialisation and starts up all processes. It then remains dormant until it receives a signal from the operating system to shutdown. Then it terminates all processes that it started and releases any system resources back to the system in a controlled manner.

This chapter describes the basic operation of the **gctload** program and the format of the text file it uses. **gctload** is available for a number of operating systems. Any detail specific to a particular operating system is described in a later section.

4.2 The gctload Program

The program **gctload** is run to initialise the system environment. It reads the system configuration from a file called system.txt. This is a user configurable file containing details of all the module identifiers known to the system, details of whether they are local modules or remote modules accessed by a local module (message redirection) and lists the command line for all processes to be started by gctload.

gctload builds a pool of message buffers for subsequent use by the getm() and relm() functions. The user may modify the size of this buffer pool by the use of a command line parameter if required.

gctload creates a message queue for each of the local module identifiers. (It subsequently expects a process to service each message queue otherwise messages will be written to message queues and never read causing a loss of system messages).

It initialises the message queue look-up table so that messages destined for modules that do not exist locally are re-directed to a message queue for a module which does exist locally.

Having created the system environment, gctload proceeds to spawn all processes listed in the system configuration file in. It then remains dormant until it receives a signal from the user to shutdown the system.

To shut down the system it terminates any processes that it created and releases all system resources back to the operating system.

4.3 System Configuration File

The system configuration file is a text file used by **gctload** to configure the software environment.

The file syntax permits the use of comments to improve the readability of the file. Comments are inserted into the file by using an asterisk (*); all characters on the line after the asterisk are ignored.

Numbers can be entered in either decimal or hexadecimal format. Hexadecimal numbers should be prefixed with 0x. For example the value eighteen can be entered in either of the following formats:

```
0x12      * (Hexadecimal)
18        * (Decimal)
```

The System Configuration File commands allow local modules to be declared (each local module requires a message queue), messages for non-local modules to be redirected to local modules (no message queue but a reference to a message queue which must already exist) and command lines for processes to be started up to be listed. The syntax of each command is listed in the following sections.

4.3.1 Create a message queue for a Local module

Synopsis:

Command to create a message queue for a given module identifier which will be serviced by a local module.

Syntax:

```
LOCAL      <module_id>
```

Example:

```
LOCAL      0x20      * Create message queue for module_id 0x20
```

Description:

This command causes gctload to create a message queue and associate the queue with the given module_id.

These commands should appear at the start of the System Configuration file. One entry should appear for each local module that will run in the system. The module identifier, <module_id>, must be in the range 0 .. 255 and must not have already been declared. Usually the module_id is entered in hexadecimal format.

4.3.2 Re-direct messages to another module

Synopsis:

Command to cause messages for a given module identifier to be redirected to an alternative message queue.

Syntax:

```
REDIRECT <new_module_id> <existing_module_id>
```

Example:

```
REDIRECT 0x22 0x20 * Redirect messages for 0x22 to module 0x20
```

Description:

This command causes messages destined to <new_module_id> to be redirected to <existing_module_id>. The <existing_module_id> must have already been declared as a local module.

Messages for many module identifiers may be re-directed to a single module. A separate command line should be used in each case.

Typical use for this command is to redirect messages intended for processes that are running on a remote board via a local process which is responsible for transferring the message to the remote board.

4.3.3 Spawn a process

Synopsis:

Command to spawn process.

Syntax:

```
FORK_PROCESS <process_path_name> { <parameters> }
```

Example:

```
FORK_PROCESS /mydir/BIN/myproc * Startup my process
```

Description:

This command will cause the specified process to be spawned once the system environment has been created. Under some operating systems it is possible to specify command line parameters for the process to be forked.

A process does not have to be spawned in the configuration file providing it is run after **gctload** and its module identifier has been declared as local. The advantage of using the configuration file is that all processes spawned by gctload can be shutdown in a controlled manner using a single command.

4.3.4 Example Configuration File

The example shown creates and assigns a message queue to module_id's 0x10 and 0x20, and redirects messages for module_id 0x30 to module_id 0x20. It then starts up processes to service the message queues for the local modules:

```
*
*   Example System Configuration File - system.txt
*
*
LOCAL      0x10      * Process_A's Module ID
LOCAL      0x20      * Process_B's Module ID
*
REDIRECT   0x30  0x20 * Redirect msgs for Process_C to Process_B
*
FORK_PROCESS Process_A
FORK_PROCESS Process_B
```

5 OPERATING SYSTEM VARIATIONS

5.1 SCO Unix

The **gctload** program should be executed from the directory containing the System Configuration File.

An alternative System Configuration File filename (instead of system.txt) can be selected using the command line option `-c<filename>`.

To shutdown the system it is first necessary to determine the process id (pid) of the gctload process. Then enter the following command:

```
kill <gctload_pid>
```

Command line arguments can be used following the FORK_PROCESS command although there may be some limitations to the symbols that are permitted.

The number of messages available to the system is limited by the number of kernel message headers. Attempting to use more messages may cause the system to halt. Additional message headers can be allocated by modifying and rebuilding the kernel (see the appropriate manual for your Unix system).

Typically the following System V IPC values must be changed:

MSGNMI	Number of Message Queue Identifiers
MSGTQL	Number of System Messages Headers

The default values for these are given in `/usr/include/sys/msg.h`.

The new values for these parameters should be set to at least the following values. There may be other users of these resources so the actual value may need to be greater than the values shown.

MSGNMI = At least the number of 'LOCAL' entries in system.txt.
MSGTQL = At least the number of MSGs in the system.

5.2 Solaris

As for SCO Unix except that it is not necessary to rebuild the kernel when modifying the System V kernel parameters. Instead add the following lines (with appropriate values) to the file `/etc/system`:

```
set msgsys:msginfo_msgmni=50  
set msgsys:msginfo_msgtql=200
```

The values are read by the kernel at boot time so there is no need to re-build the kernel, just reboot the system.

5.3 QNX

The **gctload** program should be executed from the directory containing the System Configuration File.

An alternative System Configuration File filename (instead of system.txt) can be selected using the command line option `-c<filename>`.

To shutdown the system either determine the process id (pid) of the gctload process. Then enter the following command:

```
kill <gctload_pid>
```

or use the QNX specific command `slay` which removes the need to determine the process id:

```
slay gctload
```

Command line arguments can be used following the FORK_PROCESS command although there may be some limitations to the symbols that are permitted.

5.4 Windows NT

The **gctload** program should be executed from the directory containing the System Configuration File. An alternative System Configuration File filename (instead of system.txt) can be selected using the command line option `-c<filename>`.

To run the system in a separate console use:

```
start gctload -csystem.txt
```

To run the system within the current console use:

```
gctload -csystem.txt
```

The mechanism to shutdown the system depends on how the software was started up. If the system was run in a separate console all processes can be stopped in one operation by pressing CTRL-C. If the system was run in the current console enter CTRL-C to stop gctload and then stop each other process individually using the Task Manager.

Command line arguments can be used following the FORK_PROCESS command although there may be some limitations to the symbols that are permitted.

Note that the filename of the binary file on each FORK_PROCESS command line must include the .exe file extension.

Currently gctload does not run as an NT service so it is necessary to restart it manually whenever the system is re-booted.

5.5 LINUX

The **gctload** program should be executed from the directory containing the System Configuration File.

An alternative System Configuration File filename (instead of system.txt) can be selected using the command line option `-c<filename>`.

To shutdown the system it is first necessary to determine the process id (pid) of the gctload process. Then enter the following command:

```
kill <gctload_pid>
```

Command line arguments can be used following the FORK_PROCESS command although there may be some limitations to the symbols that are permitted.

5.6 VxWorks

The gctload program should first be loaded into memory. It should then be executed by entering:

```
gctload "system.txt"
```

The system can be halted by executing a separate utility program supplied with the distribution.

APPENDIX A : MESSAGE FORMAT

A.1 MSG Message Structure

A message consists of a fixed header field and a variable length parameter field:

```
typedef struct msg
{
    HDR                hdr;
    unsigned short    len;
    long              param[80]
} MSG;
```

hdr: The message header

len: This field indicates the number of bytes in the parameter area of the message. Some messages do not contain any data in the parameter area in which case **len** is set to zero.

param: The parameter area of the message. The contents of this field are dependent on the message type. This field is normally accessed via a pointer obtained using the macro `get_param()`, which returns an unsigned character pointer to the param field.

The meaning of each field for a given message type is described in the individual application message specification. (See Programmer's Manual for each module).

NOTE: Users of systems where the structure 'MSG' is already defined for other purposes should use the alternative definition 'MSF'.

A.2 Header Fields

All application messages start with a common header which is used to determine the message type, the source and destination module identities and status information. This header structure is defined as follows:

```
typedef struct hdr
{
    unsigned short    type;
    unsigned short    id;
    unsigned char     src;
    unsigned char     dst;
    unsigned short    rsp_req;
    unsigned char     hclass;
    unsigned char     status;
    unsigned long     err_info;
    hdr              *next;
} HDR;
```

type: The type field is used to distinguish between different messages. It uniquely identifies the format of the remainder of the message and in particular the format of the message parameter area.

id: The id field allows modules which handle multiple internal instances of a single entity (such as a signalling link) to distinguish which entity the message is destined for.

src: The src field contains the module identity of the module which issued the message.

dst: The dst field contains the module identity of the module which the message is destined.

rsp_req: The rsp_req field is used by the originator of a message to indicate whether or not it requires confirmation from the receiving module that the message has been received.

If the sending module requires confirmation it sets a bit in the rsp_req field prior to sending the message. Which bit to set is determined by the least significant nibble of the module's own module id (as written in the src field). For example if the module id is 0x36 and message confirmation is required, the user would set bit 6 in the rsp_req field so rsp_req would equal 0x0040.

If message confirmation is not required then the rsp_req field should be set to zero.

The confirmation message takes the same format as the request message but uses a different **type** value. The **type** value for a confirmation message is derived from the **type** value in the request message by clearing bit 14.

hclass: This field is reserved for future use it is assigned by getm() and must not be modified.

status: This field is used for confirmation messages and indications to indicate the status associated with the message. A value of 0 in a confirmation message usually indicates success.

err_info: This field is used in some confirmation and indication messages to supplement the status field and provide additional information

next: This field is reserved for future use and should not be used.

A.3 Parameter Field

The parameter field can contain 0 to 320 bytes of data. The data is stored in the parameter field in a host independent format. The contents and format of messages parameter field are defined in the various Programmer's Manuals.

APPENDIX B: DEFAULT MODULE IDENTIFIERS

The default module identifiers used by System7 software are listed in the following table. In some systems these default values may be changed when the system is run up so care should be taken to understand that the module identifier is not necessarily fixed to the default value.

Module identifiers with a least significant nibble set to 0x0d are reserved for user generated applications. Although the values may be used in example applications supplied by DataKinetics.

Module identifiers with a least significant nibble set to 0x0c are reserved entirely for user applications. These 16 module identifiers will not be used in any DataKinetics products and are therefore available for use by the user in custom applications in the knowledge that they will not conflict with future DataKinetics applications.

Value	Mnemonic	Description
0x00	DVR_TASK_ID	Driver module for 68302, 68360, MC860
0x10	MVD_TASK_ID	Physical switch & clock driver (per-board)
0x20	SSD_TASK_ID	Physical board interface module
0xb0	RSI_MOD_ID	RSI socket based interface
0x21	CONG_TASK_ID	Congestion module
0x71	SS7_TASK_ID	MTP2 protocol module
0x22	MTP_TASK_ID	MTP3 protocol module
0x32	RMM_TASK_ID	Internal DataKinetics use
0x23	ISP_TASK_ID	ISUP protocol module
0x33	SCP_TASK_ID	SCCP protocol module
0x14	TCP_TASK_ID	TCAP protocol module
0x15	MAP_TASK_ID	MAP protocol module
0x4a	TUP_TASK_ID NUP_TASK_ID	TUP/NUP protocol module
0x0d	APP0_TASK_ID	User's application module
0x1d	APP1_TASK_ID	User's application module
0x2d	APP2_TASK_ID	User's application module
...
0xdd	APP13_TASK_ID	User's application module
0xed	APP14_TASK_ID	User's application module
0xfd	APP15_TASK_ID	User's application module
0x8e	MGMT_TASK_ID	General management module
0xdf	SIU_MGT_TASK_ID	Internal DataKinetics use
0xef	REM_API_ID	Remote (users) application
0xff		Invalid module_id - do not use!

APPENDIX C: VALUES RESERVED FOR CUSTOM USE

In some cases users may wish to add their own modules and messages to a system. To facilitate this a range of module identifiers and message types have been reserved specifically for this purpose and will not be used in any DataKinetics products.

C.1 Reserved module identifiers

All module_id values containing 0x0c as the least significant nibble are reserved for use in user applications.

C.2 Reserved message types

A total of 1024 message types are reserved exclusively for use in user's own applications.

The reserved message types are of the following 4 formats where the nibbles identified by a question mark can be set to any value:

0x?cc?

0x?cd?

0x?ce?

0x?cf?