Classic Protocol for Clients

Network interface to IRMs

Tue, Mar 3, 1998

The Classic protocol is used for data acquisition across the network with Internet Rack Monitor (IRM) front ends. It uses UDP port number 6800 (decimal) for all messages. The UDP datagram may contain one or more messages. The generic Classic protocol message layout is shown schematically as follows:

```
typedef struct {  /* Classic protocol message structure */
      int16 mSize;  /* size of message including mSize field */
      int16 dNode;  /* destination node number--may be 0000 */
      int16 mType;  /* hi 4 bits = message type, lo 12 bits depends on type */
      int16 mRest[];  /* rest of message depends upon type */
      } CLASSIC_MSG;

typedef CLASSIC_MSG[] CLASSIC_DATAGRAM; /* one or more messages in datagram */
```

The `mSize` word is the total number of bytes covered by the message, including the `mSize` word. It is always an even number. Multiple messages may be concatenated within the datagram, each message beginning with its own `mSize` word. There is no logical distinction made for messages that are delivered concatenated or in separate datagrams. In any case, messages are processed in the order that datagrams are received and in sequential order within the datagram.

The `dNode` is the target node# for the message. Each IRM is identified by a node#, an example of which is `0562` (hex). (At Fermilab, each such node is registered with the Domain Name Server as `node0562.fnal.gov`, for example, so that one can easily obtain the IP address given its node number.) The `dNode` field value may also be zero.

The message type number is in the most significant 4 bits of the `mType` word. Here are the values of message type number that are in use:

Msg type#	*Meaning*
0	Data reply in response to data request
2	Data request
3	Data setting
4	Analog alarm
5	Digital alarm
6	Comment alarm

The format of the rest of the message, including the least significant 12 bits of the `mType` field, depends upon the message type number. The data request and data reply message formats are described first, followed by the setting message format, followed by the alarm message formats.

Data request message

A data request message specifies a type of data requested for a list of devices. It is possible to ask for an *array* of types of data for the same list of devices, in which the reply returns a list of data for each type of data requested. Two words were coined to characterize the parameters to such a request. A **listype** specifies the kind of data requested. Each listype specification includes the listype number and the number of bytes of data to be returned for each device in the list of devices. Listype numbers are in the range 0–85 as of this writing.

An **ident** specifies a generic device. Its format comes in several types, each depending upon the listype used, but its length is always an even number of bytes. The first two bytes of any ident are the target node number for the device, so that the ident specification is global. (The only exception is used with the listype that provides for 6–character analog device name look-up, in which the ident consists only of the ascii 6–character name.) The rest of the ident format depends upon the ident type. The most common ident type is of length 4 bytes, where the last two bytes are an index number, such as an analog channel number or a binary bit number. Another common ident form is the 6-byte memory address ident, in which the 2-byte node number is followed by a 4-byte memory address. Note that whenever more than one listype is used, each must imply the *same* ident type, since each listype applies to the entire ident array. See the document called Listypes and Idents for more details.

The format of a request message includes a request id that is returned in any reply message, the reply period, the number of listypes, and the number of idents. This header is followed by the array of listype specifications and the common array of idents to which each listype applies. The reply resulting from such a request includes an array of data for each ident specified for the first listype, followed by an array of data for each ident for the second listype, etc. In case the number of bytes requested for any listype in the listype array is odd, and the number of idents is also odd, the array of data for that listype would end on an odd byte boundary. In this case, the reply data for the following listype is padded to an even byte boundary. As stated before, the total size of *any* Classic protocol message is padded to an even byte boundary, if necessary.

The following structures illustrate the layout for a request that uses an analog channel number ident:

```
typedef struct { /* listype array element structure */
      int16 lType;  /* listype#*256 */
      int16 nByte;  /* #bytes */
      } LISTYPE_SPEC;

typedef struct {  /* analog channel ident structure */
      int16 node;  /* node number */
      int16 chan;  /* analog channel number */
      } CHAN_IDENT;

typedef struct {  /* Data request message structure */
      int16 mSize;  /* Message size in bytes including mSize word */
      int16 dNode;  /* Destination node# may be zero */
      int16 mType;  /* 2000(F000) + servF(0800) + msgId(07FF) */
      uint8 period;  /* period in cycles, or zero for one-shot */
      uint8 nLtypes;  /* flags(F0) + #listypes(0F) */
      int16 nIdents;  /* #idents */
      LISTYPE_SPEC listypes[];  /* array of listype specs */
      CHAN_IDENT chans[];  /* array of idents */
      } DATA_REQ_MSG;
```

In some cases where fields are packed together, each field is associated in the comment with the relevant mask to indicate its placement. The `servF` bit is the server request flag bit that will be described later. The 12-bit `msgId` field is used to identify the request as distinct from any others sent by the same client. It is echoed in the reply message. More on this is described later.

The period byte specifies the period between replies, expressed as a count of cycles. A period value of zero implies a one-shot request, to which only a single reply is given. The flags nibble is used for options, only one of which is presently used. The most significant flags bit (mask=`80`), if set, indicates that the period byte is an 8-bit clock event number, so that replies to the request are to be sent only when the clock event occurs. (In practice, clock events are sampled once per 10–15 Hz cycle, so a reply can occur no more often than every cycle no matter how frequently the clock event occurs.) The number of listypes indicates the number of entries in the listype specs array. Note that all listypes used in a single request must relate to the same ident type, since the same ident array is used for processing each listype in the array.

When a request is initialized, it is "compiled" into a form that optimizes the speed of building replies. The first reply occurs as soon as possible; subsequent replies occur in synchronism with the cyclic operation of the system. (For the clock event case, there is no immediate reply.) Normal system operation begins each 10–15 Hz interrupt-triggered cycle by updating the local data pool, including executing any local application activities appropriate for that cycle. After the data pool is updated, which often takes only a few milliseconds, reply messages are built for each active data request for which a reply is due on that cycle. All reply messages built for each requesting client as can fit are concatenated into one or more datagrams for delivery.

For a one-shot request, as soon as the reply is delivered to the network, the request is automatically cancelled, freeing all related resources. For a periodic or clock event request, reply messages are delivered until the requester cancels the request. A cancel message is formatted much like an invalid data request. Build a request that specifies the same `mType` word, but that indicates a zero value for both `nLtypes` and `nIdents`. No array of listypes or idents need be included. When such a cancel message is received, all resources for that request are freed. It is possible that the requester will see one more reply message if it has already been queued to the network.

Note that every data request sent from the same source socket must specify a different request id, as long as it is active. A request id should not be re-used until some time after cancellation of a request using that request id. This is because there may be a lagging reply message yet to be delivered to the client, or replies may continue to be sent if the cancel was unsuccessful. If a request id is re-used in a new request message too soon after a cancel message for a previous request is sent, a reply to the previous request may be misinterpreted as a reply to the new request. Use request ids in the range `0001–07EF` only.

If the client receives a reply message for which the request id is unknown, a cancel message should be sent to the socket that sourced the reply message. This technique covers the case that a cancel message was lost, or the case that a host client "went away" without delivering a cancel to an active request. In case the client host receives a reply destined to an unknown *socket*, the host system is expected to deliver a "port unreachable" ICMP error message. Upon reception by the server, this will cause all active requests from that client socket to be cancelled.

Data request message—server style

The `servF` bit in the `mType` word, when set, indicates a server-style request, which is used when a data request includes an array of idents that may include at least one node number that is not the server node number. Upon reception of a data request with the server flag bit set, the server will act as a client and forward the request to a multicast destination, if appropriate, collect the replies from any nodes represented in the ident array, arrange the reply data in order appropriate for the original request message, and deliver the complete reply to the original client. In order for periodic replies to server-style requests to be valid, the nodes participating in such requests must be operating synchronously, driven by the same cyclic interrupt, normally decoded from a suitable clock event. Operating synchronously, all nodes will update their respective data pools simultaneously, and all will deliver replies to active requests at about the same time. A server node, acting to fulfill a server-style request from a client, receives all related partial replies and inserts the data according to the original request order. At a certain time in the cycle, traditionally at 40 ms past the beginning of the cycle, all active server-style requests for which replies are due are delivered. At this deadline, by which time all partial replies for a given request must be received by the server node, the server node can deliver a coherent reply to the original client. The purpose of this server-style support is to make it easier for a host to process requests that cross multiple nodes.

Upon reception of a server-style request, the server node examines the request, separates out the part of the ident array that matches its own node number, and forwards the abbreviated request to a destination that depends upon the number of non-local nodes it finds in the ident array. If there is only one such, it forwards the request to that specific node; if there is more than one, it forwards to a multicast IP address that will reach all nodes that are part of one project. The forwarded request message does not have the server bit set. (It also uses a new request id established by the server node.) Each node in the project that receives this forwarded request message checks for any node numbers in the ident array for which the node number matches the local node number. If any matches, a request is initialized to deliver replies for its own idents; all non-local idents are ignored. (If there are no matches, the receiving node ignores the entire request.) The partial replies are delivered at the usual time early in the cycle, to be received by the server node before the deadline, at which time the server node delivers the composite reply, using the original client's request id in the reply message. Thus, a number of nodes may operate synchronously to produce the aggregate reply returned to the client by the server node.

For a one-shot server-style request, or for the first reply due from a periodic (but non-clock event) request, there is no delay of the reply until the deadline time. As soon as the first partial reply has been received from all nodes participating in the request, the server node delivers the reply to the client. When a server-style request is cancelled, the server flag bit is included in the cancel message. When the server node sees such a cancel request message, it forwards it to the other nodes participating in the request so that all partial replies cease and all appropriate resources are freed. There is no reply to a cancel request message. If the cancel is unsuccessful, replies will continue to be delivered to the host, each of which should prompt a cancel response from the host. Very soon all communications for that request should stop.

For a data request to be considered valid, its parameters should pass some tests of reasonableness. The number of listypes may range from 1–15, the number of idents may range from 1–1024, and the product of the two is also limited to 1024. The size of a reply is limited to the maximum datagram size supported, about 9000 bytes.

Data reply message

The format for a data reply message layout is as follows:

```
typedef struct { /* Data reply message structure */
      int16 mSize;  /* Message size in bytes including mSize word */
      int16 dNode;  /* Destination node# may be zero */
      int16 mType;  /* 0000(F000) + servF(0800) + msgId(07FF) */
      int16 rStat;  /* reply message status */
      int16 data[];  /* reply data in request-implied order */
      } DATA_RPY_MSG;
```

The low 12 bits of the `mType` word, including the server bit flag, are exactly the same as those of the request message `mType` word. The status word applies to the entire request and may have the following values:

Reply status	*Server*	*Meaning*
0	—	No error
1	—	Internal error—should not occur.
2	—	Internal error—should not occur.
3	—	Internal error—should not occur.
4	—	Bus error detected in building reply.
5	S	Internal error—should not occur.
6	S	Internal error—should not occur.

7	S	Partial reply tardy or missing
8	S	Partial reply non-existent.

A non-server request that is judged invalid when received is simply ignored, and no reply is sent. A server request that is invalid *may* be ignored, unless only the contributing node can detect its validity. Some error codes apply only for server-style requests. Bus error status may be returned for both request types.

Since only a single status value is included in a reply message, a priority system is used to determine which nonzero value to return, in case more than one applies. Bus error status is shown if building any part of the reply produced a bus error. Without a bus error, an 8 takes precedence over a 7. If a 7 is received, it means that all nodes participating in a server request have replied to the server node at least once since the request was initialized, but that at least one node has died or is tardy in delivering its partial reply by the deadline. An error 7 can occur when the participating nodes are not all running synchronously, such as may occur when the interrupt trigger (or clock signal) is removed from a node, so that it runs at a slightly slower internally-triggered (and asynchronous) cycle rate. An 8 error means that at least one node referenced in the ident array is not responding at all.

For a non-server request, a reply status that is nonzero is likely to mean only bus error. For a server request, one will likely only find nonzero status values of 4, 7, 8.

Data request/reply/cancel example

The following example shows a data request sent to a node `0562` for 15 Hz analog channel readings and settings for its own analog channels `0100`, `0102`, and `0107`. It includes an example reply message and a suitable cancel message. The message is shown in integer words. Internet byte order is such that the most significant (left-hand) byte of each word occurs first on the network.

Data request message sent from host to node 0562

```
001E     Message size = 30 bytes.
0000     Destination node may be zero.
2001     Request message, non-server, request id = 1.
0102     Request period = 1 cycle, #listypes = 2.
0003     #idents = 3.
0000     Listype 0 = analog reading
0002     2 bytes
0100     Listype 1 = analog setting
0002     2 bytes
0562     Ident #1
0100
0562     Ident #2
0102
0562     Ident #3
```

```
0107
```

Data reply message returned at 15 Hz from node 0562 to host

```
0014     Message size = 20 bytes
0000     Destination node may be zero
0001     Reply message type, request id = 1.
0000     Status = 0. No errors.
FFFE     Chan 0100 reading
0047     Chan 0102 reading
00A5     Chan 0107 reading
472D     Chan 0100 setting
0040     Chan 0102 setting
00B4     Chan 0107 setting
```

Cancel message sent from host to node 0562

```
000A     Message size = 10 bytes.
0000     Destination node may be zero
2001     Request message type, request id = 1.
0000     #listypes = 0.
0000     #idents = 0.
```

Data setting message

A data setting message uses the same listype and ident reference as used for a data request. Some listypes, however, are not settable, such as an analog reading. A setting command only includes one listype and one ident. One or more concatenated setting commands comprise a setting message. Of course, one can include more than one setting message, or any Classic protocol message, within the same datagram.

The following structure definitions show the format of a setting message for an analog setting following the generic Classic protocol message header:

```
typedef struct {   /* analog channel ident structure */
      int16 node;  /* node number */
      int16 chan;  /* analog channel number */
      } CHAN_IDENT;

typedef struct {  /* Data setting message structure */
      int16 mType;  /* 3000(F000) + servF(0800) + ident size in words(000F)*/
      LISTYPE_SPEC listype;  /* listype spec */
      CHAN_IDENT chan;  /* analog channel ident */
      int16 data;  /* setting data */
      } DATA_SET_CMD;

typedef struct {  /* Data setting message structure */
      int16 mSize;  /* Message size in bytes including mSize word */
      int16 dNode;  /* Destination node# may be zero */
      DATA_SET_CMD setCmds[];  /* array of setting commands */
      } DATA_SET_MSG;
```

In the `mType` word is the message type number, a server flag bit, and the ident size in units of 16-bit words, not bytes. The ident size must be nonzero and serves to locate the position of the setting data within the setting command. The format of a listype spec is the same as before in the description of the data request message format.

The server flag bit, used to identify a server-style setting message, is analogous to that used in the server-style request message. It allows sending a setting message to a node other than the node that must perform the setting. When a node receives a server-style setting message, it forwards it (with the setting flag bit *not* set) to the node whose node number it finds in the first word of the ident.

When a setting message is received for which the server flag bit is *not* set, the Classic protocol server invokes the appropriate setting handler that is indicated by the listype number specified. No setting acknowledgment message is returned to the client to indicate the success of the setting operation. (With a suitable definition of a new message type, this could probably be easily added.) When it is vital to determine whether the setting "worked," one may issue a data request to see whether the data was changed. In the case of an analog channel setting, for example, one can follow the setting message with a one-shot data request message to read the current analog setting value. If the setting value returned matches the value that was sent in the setting command, then the setting operation was successful. (When a setting handler processes analog control settings, it updates the setting value *only* if no errors were detected in performing the setting.) Note that in using this technique to verify the success of a setting operation, one may include the data request message inside the same datagram as the setting message, as long as the data request follows the setting message. All Classic messages in a datagram are processed in sequence.

Data setting message example

The following example illustrates an analog control setting message sent to node `0508` that sets its analog control channel `0007` to 5 volts (`4000`):

Data setting

```
0010     Message size = 16.
0000     (Dest node may be zero)
3002     Setting message, non-server, ident size = 2 words.
0100     Listype 1 = analog setting.
0002     2 bytes
0508     Ident
0007
4000     Data
```

The following data request message and reply message would serve to verify the success of the above setting in reading back the current value of the setting word.

Data request   *Data reply*

```
0012  Message size = 18.                    000A  Msg size = 10.
0000                                        0000
```

```
      2002  Request message,request id = 2.     0002  Reply msg, req id = 2.
      0001  One-shot request, one listype.      0000  Reply status = 0.
      0001  One ident                           4000  Reply data
      0100  Listype 1 = analog setting
      0002  2 bytes
      0508  Ident
      0007
```

Alarm message

The alarm message format is used when a node detects a change in alarm conditions for some analog channel or digital bit during its alarm scan that it performs every 10–15 Hz operating cycle. A separate message type number is used for each of the three variations of alarm messages, which include analog alarms, digital (bit) alarms, and comment alarms. Each message format is described in turn.

```
typedef char[6] NAME6;  /* 6-character text */
typedef char[4] UNIT4;  /* 4-character text */

typedef struct {  /* Time-of-day in BCD */
      byte yr;  /* 2-digit BCD year, range 00-99 */
      byte mo;  /* 2-digit BCD month, range 01-12 */
      byte da;  /* 2-digit BCD day, range 01-31 */
      byte hr;  /* 2-digit BCD hour, range 00-23 */
      byte mn;  /* 2-digit BCD minute, range 00-59 */
      byte sc;  /* 2-digit BCD second, range 00-59 */
      byte cy;  /* 2-digit BCD cycle, range 00-14 */
      byte fill;  /* filler byte */
      } TOD_BCD;

typedef struct {  /* analog alarm message structure */
      int16 mSize;  /* Message size in bytes including mSize word */
      int16 dNode;  /* Destination node# may be zero */
      int16 mType;  /* 4000 */
      int16 chan;  /* analog channel number */
      int16 aFlags;  /* alarm flags word */
      int16 reading;  /* analog reading prompting alarm */
      int16 setting;  /* analog setting */
      int16 nominal;  /* analog nominal used in alarm scan */
      int16 tolerance;  /* analog tolerance used in alarm scan */
      int16 spare;  /* spare word */
      NAME6 name;  /* 6-character ascii name text */
      TOD_BCD time;  /* 8-byte time-of-day in BCD */
      float fScale;  /* reading full scale conversion constant */
      float offset;  /* reading offset conversion constant */
      UNIT4 units;  /* 4-character ascii engineering units text */
      } ANALOG_ALARM_MSG;
```

A node emits an analog alarm message for every change in good/bad state of an analog channel that is enabled for alarm scanning. Analog alarm scanning includes a hysteresis logic to prevent alarm message chatter when a reading is near the nominal-tolerance window, defined as the range between (nom–tol) and (nom+tol). Once a channel enters the "bad" state by its reading being found outside this window, the reading must be found inside the half-size range of (nom-tol/2) to (nom+tol/2) for it to change to the "good" state.

The alarm flags word modifies how the alarm scan logic proceeds for each analog channel. Here is a list of the bits in the flags word and their meanings when set:

Bit mask	*Meaning*
`8000`	Active bit. Channel enabled for alarm scanning
`4000`	Pattern bit. Digital logic used rather than range logic.
`2000`	Inhibit bit. Beam inhibit control line asserted when in bad state.
`1000`	spare.
`0800`	Beam bit. Alarm scan enabled for beam cycles only.
`0400`	Bypass bit. Turn off Active bit, emit good message if bad.
`0200`	Alarm clear bit. Used obscurely internally.
`0100`	Bad bit. 0=good, 1=bad.
`0080`	Silent bit. Inhibit sending alarm message, but keep counts.
`0040`	Invalid bit. Inhibits scan this time. Set if data acquisition error.
`0020`	spare
`0010`	spare
`000F`	Tries needed count. #times needed to switch good/bad state.

The pattern flag bit modifies the analog alarm scan to perform a digital check on an entire digital status word, as opposed to a single bit as the digital alarm scan does. When this pattern flag bit is set, the nominal value is taken as a nominal bit pattern, and the tolerance value is taken as a mask. The logic then exclusive-ORs the present reading with the nominal bit pattern and ANDs with the mask. If the result is zero, the channel is in the "good" state; otherwise, the channel is in the bad state. This logic allows declaring an alarm condition if any selected set of bits in a digital status word deviates from a nominal pattern. Such digital status words are built by specifications in a system table that prescribe how to combine digital status bits into composite status words to update a set of "analog" channel readings.

The reading scale factors are included in the analog alarm message structure so that all information required to convert the reading value into engineering units is available within the alarm message itself.

To determine the node number of the node reporting the alarm message, one must obtain it from the socket that sent it. The node's name as registered with the Domain Name Server includes the node number, as in `node0562`, for example. Also, one could send a data request to the node and ask for the node number from its global system variables—and presumably cache such results.
An example analog alarm message is as follows:

Analog alarm

```
      002E     Message size = 46.
      0000     (Dest node may be zero)
      4000     Analog alarm message
      0107     Analog channel#
      8109     Alarm flags (bad state)
      438E     Reading
      0000     Setting
      6146     Nominal
      1999     Tolerance
      0000     spare
      4356     Device name 'CV01W '
      3031
      5720
      9803     03/02/98 1529:47, cycle 11
      0215
      2947
      1100
      41C8     fScale = 25.0
      0000
      0000     offset = 0.0
      0000
      4750     eng units text 'GPM '
      4D20
```

The alarm message shows that device `CV01W` became 'bad' as a result of its reading being found outside its tolerance window. The time-of-day is included, down to the operating cycle on which the alarm scan detected this condition. The scale factors and engineering units text permit the alarm handler to announce the reading value in engineering units by performing the linear scaling using the standard formula

```
      eng = raw/32768*fScale + offset;
```

In this formula, `raw` is the reading taken as an integer value, and `eng` is the result in engineering units, in this case in GPM.

Digital alarm message

The digital alarm message is reported by a node when a digital status bit changes state. Each digital status bit has an associated set of alarm flags that have the same meanings as those described in the analog alarm message section, but with one difference. The Pattern bit is replaced by a Nominal bit that indicates the nominal bit state, which is the state corresponding to the "good" alarm state. The opposite state is the "bad" state. The digital alarm message format is as follows:

```
typedef char[16] TEXT16;

typedef struct {  /* Digital alarm message structure */
      int16 mSize;  /* Message size in bytes including mSize word */
      int16 dNode;  /* Destination node# may be zero */
      int16 mType;  /* 5000(F000) */
      int16 bit;  /* digital bit number */
      int16 bFlags;  /* alarm flags word */
      TEXT16 bitText;  /* 16-character ascii bit text */
      TOD_BCD time;  /* 8-byte time-of-day in BCD */
      } DIGITAL_ALARM_MSG;
```

During the alarm scan each cycle, every binary status bit is checked for a change in its alarm state, if it is enabled by the active bit in its alarm flags word. Each change in alarm state provokes a response. Here is an example of a digital alarm message:

Digital alarm

```
      0022     Message size = 34.
      0000     (Dest node may be zero)
      5000     Digital alarm message
      010C     Digital bit#
      9180     Alarm flags (bad state, silent)
      5246     Bit text = 'RF3 DRIVER PA OL'
      3320
      4452
      4956
      4552
      2050
      4120
      4F4C
      9803     03/02/98 1611:32, cycle 5
      0216
      1132
      0500
```

Comment alarm message

The comment alarm message is reported by a node for the purpose of announcing that some remarkable event has occurred. There is no associated good/bad state of a comment alarm. Only two comment alarms are traditionally supported. One is emitted when a system reset occurs. The other is emitted when an alarms reset is performed. An alarms reset means that the good/bad alarm flag bits for all analog channels and all digital bits are reset. On the next alarm scan, then, new bad messages will be issued for all channels and bits found to be in the bad state. The comment alarm message layout is as follows:

```
typedef struct {  /* Comment alarm message structure */
      int16 mSize;  /* Message size in bytes including mSize word */
      int16 dNode;  /* Destination node# may be zero */
      int16 mType;  /* 6000(F000) */
      int16 comment;  /* comment number */
      int16 cFlags;  /* alarm flags word */
      TEXT16 commentText;  /* 16-character ascii comment text */
      TOD_BCD time;  /* 8-byte time-of-day in BCD */
      } COMMENT_ALARM_MSG;
```

During the alarm scan, the good/bad bit is checked in the alarm flags word for each comment alarm that is enabled for such checking. If it is set, a message is emitted. Comments do not really have good/bad status. That flag bit is merely used internally by logic that wants to cause a comment message to be emitted, in order to get the alarm scanning task to build the comment message for queuing to the network. Here is an example comment message that occurs following a system reset:

Comment alarm

```
0022     Message size = 34.
0000     (Dest node may be zero)
6000     Comment alarm message
0000     Comment index#
8000     Alarm flags (active)
564D     Comment text = 'VME SYSTEM RESET'
4520
5359
5354
454D
2052
4553
4554
9803     03/02/98 1627:02, cycle 1
0216
2702
0100
```

Summary

A host implementation for Classic protocol data request support will need to manage request ids so that multiple requests made by clients do not conflict. One way to do this is to allow each client to claim a separate client socket from the underlying system. Then each client only needs to choose request ids that are different across its own requests. This approach is simple, but it means that the same host may have multiple client sockets issuing requests to the same target nodes, which means that replies must be built into separate datagrams for delivery, even though they are being sent to the same host.

A second approach is to implement host server support that uses a common source socket to communicate with the server front ends on behalf of any number of host-based clients. A client issues its request for data to the host server who constructs a data request message, including allocating a request id to use for the duration of that request's activity. When replies are received by the host server, it uses the request id in the reply header to determine to which client it should deliver the reply data. This approach may be more complex, but it means that the front ends send replies to only one target socket at the host end, which in turn means that the network can be used more efficiently, as the front end can return multiple replies in the same datagram, even when they are destined for different clients on the same host node.

No matter what method is chosen, if a reply message is received that, according to the request id found in the reply message, cannot be delivered to any client, then the support should return a cancel message. Here is an example of a reply message and the associated cancel message to be used to get the front end to cease replying.

Reply message
```
000C  mSize
0000  dNode
0855  reply, servF, req id = 85
0000  status = 0
1234  data[1]
5678  data[2]
```

Cancel message
```
000A  mSize
0000  dNode
2855  request, servF, req id = 85
0000  #listypes = 0
0000  #idents = 0
```

Another reason for implementing this cancel support is to allow the front end to monitor its own clients, so that it will time out delivering replies to clients that no longer exist, perhaps because someone else in the house picked up the extension phone, or for any other reason. Periodically, the front end may deliver a reply message to a host socket with a request id that doesn't exist, such as `0000` or `07FF`, either of which is an invalid request id. If the front end receives a response (the cancel message), it knows that the host socket connection still exists. If it does not, after repeated attempts to provoke one, it may cancel the host's requests.

IRM front ends are typically configured to send alarm messages to a multicast destination, so that multiple hosts that desire to receive alarm messages may do so without requiring additional network activity. Alarm message host support may be implemented by a separate program that listens to the same multicast port (#6800).

IRMs include alarm message support that permits any IRM to generate encoded alarm messages for printing via its serial port. If the serial port is connected to a computer's serial input, an easy alarm log can be constructed by passing the serial input data to a capture file.