COMM485

PROTOCOL SPECIFICATION V2.52

Date	Rev	Issue	ECN	Prepared by	Checked by
March 2003	2.52	Correction to Typo Error (pg25)	331	L Clark	S Duncan
November 2001	2.51	Revised Version	250	D Milne	S Duncan
March 2000	2.50	Revised Version		D Milne	S Duncan
March 1999	2.00	1st Draft for comment		D.Milne	R.Johnson

1 Introduction

1.1 What is the COMM485 Protocol?

The Micropack Engineering Ltd *Comm485* protocol is a master-slave network communication system intended for use by embedded devices in industrial settings; such systems are usually called *fieldbus protocols*. The main advantages of the protocol are that it is easy to understand, easy to implement, and provides good performance on moderate hardware :-

Easy to understand: *Comm485* is similar to a number of other packet based fieldbus, network and file transfer protocols. Any reader with previous exposure to such protocols will readily understand the workings of Comm485. For example, anyone with a knowledge of the Xmodem file transfer protocol (one of the simplest of such protocols) will probably find much that is familiar in the design of Comm485, since both are packet based, half duplex designs.

Easy to implement: although *Comm485* currently includes around 50 different message types, very few of these need to be understood by any single device. Most network devices are *passive* (or slave) devices, which is to say that they initiate no transactions on their own and instead transmit messages only in response to requests from one of a small number of *active* devices. Since slaves only ever transmit in reply to a message from a master there is never any possibility of a collision and thus no need for a complex media access layer in the protocol.

Good performance: *Comm485* does not mandate a particular packet size (except that they be 1032 bytes or smaller), so packet sizes can be adapted to the needs of individual transactions. A balance can therefore be struck between the desire to reduce per packet overheads and the desire for fast response times. Also, the fact that Comm485 lacks a media access layer means that no network bandwidth is consumed in waiting for a token to be passed or for some other form of bus arbitration to take place prior to transmission.

1.2 Who uses COMM485?

Comm485 was developed as a flexible communications mechanism for the specific range of devices which are designed and manufactured by MEL. Currently, MEL is the only company which uses the protocol in all its products, although MEL is currently in negotiations with other companies who feel that the protocol may be suitable for their needs; MEL welcomes similar approaches from others. Essentially, MEL considers the technology to be open and freely available to all, though we *would* like to keep control of the specification in order to guarantee inter-operability for the foreseeable future.

1.3 Why not use use an existing fieldbus protocol?

MEL originally designed the Comm485 protocol as a communications tool for our mark I flame detection cameras. While the camera electronics was being designed we looked around for a "standard" protocol which would be suitable for our needs, and could find none.

Many existing protocols are closed; one has to license the technology in order to obtain a detailed description of the underlying architecture, and only then is it possible to determine suitability; MEL was uninclined to thrash around in the dark like that. Other protocols make limiting assumptions, for example that field devices return individual analogue "readings" in each polling sweep - such a model is not easily extended to cope with large data transfers (of grayscale images and firmware updates) over the network, or would have carried high overheads had we tried. Other protocols assume the cheapest possible hardware, and thus might mandate a maximum speed of, say, 9600 bps up to 100 metres, whereas we wished for speeds around 100kbps over one or two kilometres. Finally, many existing fieldbus protocols require the use of specialised chipsets, which would have been an obstacle if we wished to integrate desktop personal computers and laptops into the network, and might also tie us to the manufacturers of these chipsets: we would prefer that communication be done using standard off-the-shelf UARTs, at speeds and with timing constraints easily achieved on standard personal computers without a need for specialised plug in cards (which might not be available for laptops) or device drivers.

Eventually MEL determined that the only sure way to obtain an open protocol which precisely suited our needs for the foreseeable future was to design one ourselves.

1.4 Where can Comm485 be used?

The underlying assumption in Comm485 is that a network consists of a small number of master stations communicating over half duplex (potentially multidropped) transmission lines with a larger number of slave devices, and where realtime operation (say, polling intervals and response times accurate to less than a second) is not required. If these assumptions are valid than *Comm485* is likely to be a suitable choice.

1.5 Changes to the protocol from version 2.5 to 2.51

- Slight modification to the discovery protocol (optionally reply with ACK(UR_FOUND,w) instead of ACK(UR_FOUND,0) as previously defined. The new field is intended to aid in differentiating dual redundant port connections from device address collisions.
- Some new CAMTHRESH variables are defined for several device types.
- Addition of a motion sensor device class (derived from camera). Becomes device class $8xxx

1.6 Changes to the protocol from version 2.0 to 2.5

Most changes were caused by the introduction of the mark II camera, which uses a larger frame (image) size than the mark I camera :-

- Frame grabs - raw and JPEG grayscale frames have changed size. Both were 256x256 in the old camera, and are now 352x289 (raw) or 352x288 (JPEG). See the description of the PREPARE_DATA message.
- The "thresholded video" feature. Not only did the frame size change, but the run-length encoding algorithm had to change as well when the new pixel coordinates could no longer fit into bytes. See PREPARE_DATA again, and also see appendix B.
- Bit masks are affected by the same issues as the thresholded video feature.
- Minor changes had to be made to the "master device list" (MDEVLIST) structure stored on the master hub, caused by oversights in the previously defined structure: the old structure did not align fields on their expected byte boundaries, which made them very difficult to handle with some C compilers (eg. the C compiler for the ARM RISC microcontroller used by the new MEL hardware).
- Changes had to be made to the S_HUB_ERRSTAT structure returned by hubs (see description of the ERRORSTATS message), due to the introduction of the mark II hub, which has sixteen ports instead of eight. There were also field alignment problems in the old structure.
- The DISPLAY_ALERT and OPERATOR_ACK messages have been changed to allow the message body to contain state change information for more than one device. So, if a lot of things happen at once the hub can tell the display about it using fewer messages than before. If the body only contains info for one device then the new format is backward compatible with that used in protocol version 2.0.
- A new-and-improved discovery protocol has been introduced, allowing a hub to detect all connected devices in about a second, compared to around 40 seconds previously. Support for the new discovery protocol is mandatory for all devices, but is very simple to implement (unless you are developing a hub from scratch). This involved the introduction of two new messages, and is described in full in chapter 5.
- A new SET_RELAY message has been introduced to control the new MEL relay card, as well as the alarm and fault relays in the mark II camera.
- The mark II camera now latches alarms. To reset an alarm output you need to send the camera a SET_RELAY(0) message.
- A number of modifications have been made to the S_CAMTHRESH structure used to make configuration changes to devices such as cameras. The changes are all backwards compatible however.
- The protocol now allows devices to notify displays (give a warning) when a self-diagnostic test fails. Previously the only failure mode recognised was total failure involving loss of comms. This feature

assigns a previously reserved packet flags bit (the warning flag), and adds two new messages to allow the display to ask for details or clear the warning flag.

- MEL has introduced its own video switcher, which supports four video outputs instead of the two supported by the previous (non-MEL) switcher. One consequence is that the protocol no longer assumes that video-outputs 1 and 2 go to display A and display B respectively. Which display(s) receive video, and which outputs they are connected to is now fully configurable via the master device list.

2 Comm485 Protocol Layers

It has become the industry norm to describe communication protocols in terms of functional layers with reference to the standard seven layer ISO/OSI network model. We will follow that convention in this chapter as best we can, even though *Comm485* is rather too simple to fit well into the generic OSI model.

2.1 Physical Layer (RS485)

The physical layer of the protocol describes the hardware required for communication.

Comm485 does not mandate a particular hardware layer (the protocol could easily be adapted to run on, say, a wireless link or optical fibre), however all current MEL devices use the RS485 standard over copper twisted pair as the basic medium, and use standard UARTs running at 57600 bps, 8 data bits, 2 stop bits and no parity to communicate across this medium.

The RS485 standard is essentially a modern decendant of the more familiar RS232; both are at heart *serial* mechanisms, ie. bytes are transmitted one bit at a time over the medium, each bit being represented by a specified voltage range. There are however some important differences :-

RS232 represents bits as voltages measured with respect to a common ground wire, the same ground being shared by all the other signals present on the connector. A positive voltage of 3V or more* represents a 0 bit, while a negative voltage of 3V or more represents a 1 bit. If the voltage which represents a signal falls inside the -3V..+3V band then the signal is said to be in an *indeterminate* state. RS232 connector pins are normally driven all the time, so the indeterminate state arises only if the cable is too long.

* The maximum voltage for an RS232 signal depends on which revision of the standard you read, the later versions specified a maximum of 25V. On the other hand, real world RS232 devices tend to have maximum voltages at the transmitter of +5V or +12V, as these are the voltages most readily available inside a computer.

RS485 does not use a common ground: it represents bits using a voltage differential across a pair of conductors, commonly called a *balanced pair*. A positive voltage of 200mV or more represents a one bit, while a negative voltage of 200mV or more represents a zero bit. If the absolute differential is less than 200mV then either there is no signal present (perhaps because all devices connected to the transmission line are currently in listen mode), or the signal has been attenuated so much that it can no longer be read reliably; another *indeterminate* state. You could implement a complete modem interface using RS485, including all the common signals such as Rx, Tx, RTS, CTS etc, but every such signal on the RS485 connection would require its own separate twisted pair. However, this is not a big issue, because practical RS485 links are nearly always half duplex, with no hardware flow control signals,

making it very unusual to have more than one pair. Note that RS485, unlike RS232, does not mandate the use of a particular type of connector, though most RS485 implementations will either use screw terminals or "phone socket" type connectors.

RS232 allows two devices to be connected together. RS485 is a so called *multidrop* standard, allowing up to 32 devices to be connected in parallel along a single twisted pair, provided that they don't all try to talk at once (typically one master device is talking, the remaining slave devices listen and transmit only after they have been specifically addressed). The limit of 32 devices assumes that each device presents a so called "*unit load*" impedence to the transmission line. The unit load is the *maximum* load that an RS485 compatible device may present, but in fact many modern RS485 line drivers (eg. from Maxim) present only a fraction of this load, which allows more devices to be connected to the transmission line - a maximum of 128 devices is often quoted.

RS232 is designed to handle data rates up to 19200 bits per second over a maximum distance of about 15 metres. In fact we can generally go a bit faster (say 115kbps) or further (say 25 metres) than that without too many problems. However, with increasing distance or speed it becomes much harder to maintain the relatively high voltages required by the standard. RS485 starts from a lower initial voltage (5V), but allows this to be attenuated to a much greater degree (a factor of 25 versus a factor of about four for RS232) before signal quality is affected. This change is sufficient to allow RS485 to communicate easily at data rates in excess of 115kbps over a few *kilometres* of good quality cable.

RS232 voltages are referenced to a ground wire, which can cause difficulty in the presence of induced currents (noise), because these currents will discharge more easily on the earth wire than they will on the signal wire. This means that the voltage potential between the signal and ground wires can be altered, corrupting the signal. In RS485 both wires in the twisted pair are signal wires, neither are connected to ground. The impedence of the two wires is identical, both are affected equally by induced noise, the potential between the two cores is unchanged thus the signal is unaffected. Relative immunity to so-called "*common mode*" interference makes RS485 more suitable for industrial environments.

RS485 does have its own dangers however :-

At high data rates signal pulses can "reflect" off the end of the transmission line and interfere destructively with the ongoing transmission. To solve this problem it is necessary to fit termination resistors at the two extreme ends of the transmission line. Ideally these resistors should match the characteristic impedence of the cable (as quoted by the cable manufacturer); in practice however a standard 120R termination load is fitted at both ends and it is the cable which is selected to match.

Termination tends not to be an important issue on good quality cable using only moderate data rates and cable lengths. You should be careful not to *over*-terminate a transmission line (ie. connect lots of devices all with their termination jumpers fitted), since the various line drivers present may be incapable of driving a large enough voltage through the increased load, and may even damage themselves in the attempt.

A more significant problem, in MELs experience, follows from the fact that RS485 recognises three states for a transmission line: 0-bit, 1-bit and indeterminate, while the UART receiver to which the transmission line is connected has TTL inputs and therefore only recognises two states: low and high. The indeterminate state does not cause problems in RS232, because the signals are always being driven. In RS485 however the indeterminate state occurs often, eg. if all connected devices are listening then no one is driving the line. For as long as the transmission line remains in the indeterminate state the line driver will be unable to indicate the true state of the line to the receiving UART, and while under those circumstances the driver *could* continue to signal the last known "good" state, with a little bit of noise it *may* instead thrash between the 0 and 1 states and confuse the UART: if your comms device driver is processing thousands of break, framing error or spurious character interrupts per second then this is almost certainly the cause of the problem. The solution is to make sure that the indeterminate state never occurs, which you can do by *biasing* the line. Biasing is achieved by connecting the A wire to a pull-up resistor (+Ve) and the B wire to a pulldown resistor (-Ve) with the net effect that the line is held at a little over +200mV when nothing else is driving the line, causing the signal from the line to look like it's in the 1-bit (or idle) state from the point of view of a listening UART*. Be careful not to enable biasing in more than one connected device: provided biasing is not excessive then transmitters should have no difficulty driving through the increased load. Manufacturers data books for RS485 line drivers will often give more detailed instructions on how to correctly bias a transmission line.

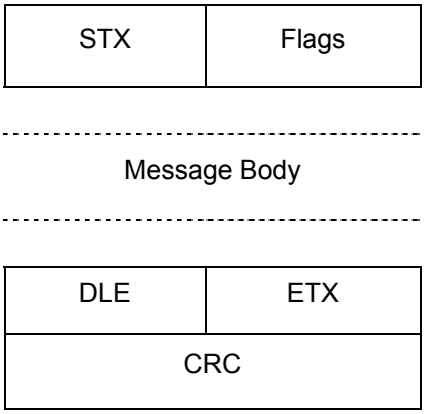
* The "1-bit" state is the appropriate idle / listen / marking state for an asynchronous serial connection, as any zero bit seen will then be interpreted as the start bit for the next incoming character.

It was mentioned above that the RS485 standard allows up to 32 devices to be connected to a single transmission line, and that some commercially available RS485 line drivers go further and allow up to 128 devices to be connected. However, there is a danger if you go overboard with the number of devices on a single multidrop, which becomes obvious if you consider what would happens when a fault such as a cable short occurs, since the result would be that you lose contact with all of those devices at once, ie. a multidrop transmission line is a *single point of failure* for all connected devices. A balance must be struck between cable costs and the number of devices you can afford to lose as a consequence of a single fault. The client will normally specify where the balance should lie. If you decide to break a large network into a number of smaller

(safer) subnetworks then you need some kind of repeater and/or router to bridge the various subnetworks - which is precisely why the MEL Hub exists.

2.2 Data Link Layer

The Data Link layer of the network specifies the format of network packets. The basic packet framing structure is as follows:-



Transmission order is from top left (STX) to bottom right (second byte of CRC). The STX, DLE and ETX fields refer to the eight-bit ASCII control codes with those names, their byte values being 0x02, 0x10 an 0x03 respectively. CRC is a 16-bit *Cyclic Redundancy Check* field, a description of which appears below. A table driven algorithm for calculating this type of CRC is given in appendix A.

Multi-byte fields in the frame or message body are transmitted least significant byte first, ie. Intel or "little-endian" format. STX, which serves as a "start of frame" attention character, may *only* occur at the start of a frame; any occurrence of STX in the body will be replaced by the two byte sequence DLE-B (0x10 0x42), ie. insert a preceding DLE character and then xor the control character with 0x40 to get a "safe" second character. Since this escaping mechanism makes the DLE character special, it too must be replaced if it occurs in the message body area, this time with the two byte sequence DLE-P (0x10 0x50). Although only these two bytes *must* be escaped, the receiver is required to be able to decode any control character which is escaped in the manner described.

- Note: although shown as a single byte, a transmitter may send several STX's prior to the flags field; this might be desirable if the environment exhibits a tendency to trash the first transmitted character in a packet.

DLE escaping is only necessary inside the region marked "message body". The protocol itself ensures that STX cannot appear anywhere else in the frame except at the start; for example an STX cannot appear in a legal "Flags" field because bit one of the flags is reserved and is always set to zero - ie. the

flags field can never equal 00000010b (STX). STX is prevented from appearing in the CRC by a method which is described later. The DLE character *can* occur in the flags or CRC fields, so you must be careful not to treat DLE as an escape character *unless* it appears in the data area of the packet.
Notice that there is no *length* field in the frame structure. Instead, the receiver should scan for the DLE-ETX couplet which marks the frame end sequence. In other words, you scan through the incoming bytes until DLE is detected, then check the following character. If that character is ETX then it marks the end of a frame, otherwise you have found an escape code.

The CRC is accumulated on all bytes from the "flags" field to the end of the message body area, inclusive, ie. it protects all changeable parts of the frame. If any characters have been escaped as described above then it is the single byte *prior* to escapement which is included in the CRC, not the two byte escape sequence. The CRC polynomial is exactly that used by Xmodem-CRC, except that the accumulator is initialised to 0xFFFF rather than zero. However, if either byte in the final CRC word equals 0x2 (ie. STX), then that byte will be replaced by the value 0x42.

The "Flags" field in the packet frame provides general control and status information relating to the device which transmitted the packet. The flags field allows devices such as hubs to extract device status information from en-route packets without necessarily understanding the content of the packet. Bits currently assigned to the flags field are as follows :-

Bit	Meaning
0	1=Ready.
1	0=Reserved (must be zero).
2	1=Active.
3	1=Warning (for diagnostics).
4	1=Alarm (mostly from cameras).
5	1=Masked (camera has "bitmask" set).
6	1=Flaremon (camera is designated as a "flare monitor").
7	1=XFlags.

The *ready* indicator (bit 0) should be set by all devices.

Bit 1 must be set to zero by all devices, which ensures that the flags field can never equal STX.

The *active* flag (bit 2) is used by a hub to indicate that it has entered normal polling mode; most other device classes will set this bit to one. When a hub sets this bit to zero it means that it is still initialising the network, discovering which devices are attached, what the correct routing is for each device and so on. In this condition the hub is not yet ready for complex exchanges, in which case other devices should restrict themselves to simple replies to specifically addressed messages coming from the hub. Slave

hubs which receive packets from a parent hub with the *active* bit reset should **not** route the packet downstream. This is particularly important when the packet is a broadcast. The master hub will transmit a CONTROLLER_ACTIVE message with the *active* bit set to one when it goes fully operational, which notifies displays in particular that they may now use the network for general communications (also see the Discovery Protocol description, chapter 5), and also allows slave hubs to begin routing again.

The *warning* flag (bit 3) is used by a device to indicate that an internal self diagnostic test has detected some kind of non-fatal problem.

The *alarm* flag (bit 4) is mainly intended to allow a camera to indicate that it has detected a fire. However, any device can in fact signal an alarm event using this bit; the display software would have to be updated to interpret the alarm condition appropriately, eg. so that an alarm event reported by a video switcher is not handled as if it were a fire (note: this is given as an example only and should *not* be interpreted as saying that MEL video switchers *do* make use of the alarm bit).

The *hasmask* flag (bit 5) is used by cameras only (however it is possible that other device classes may in future use this bit to indicate some other kind of device class specific status). In the camera, *hasmask* (if set) indicates that the camera has been told to "mask off" some portion of its view, ie. not do fire detection inside that area.

The *flaremon* flag (bit 6) is again used by cameras only (however it is possible that other device classes may in future use this bit to indicate some other kind of device-specific status). In the camera, *flaremon* (if set) indicates that the camera has been designated as a "flare monitor". Hubs and displays do not trigger alarm or control action for fire detection signals raised by a flare monitor.

The *xflags* bit (extended flags, bit 7) indicates that the flags field has been extended by at least one further byte. No use is made at present of extended flag bits, but for future compatibility existing implementation should check this bit, and if set should discard the following byte. Note that this feature is cumulative - if the second byte also has bit 7 set then a third flags byte is present, and so on. Applications may assume that the flags field will never need to exceed 32 bits (four bytes).

Note: having just read a description of the flags you may at this point be assuming that a display device is expected to determine the status of the network by polling all devices and checking the flags field of the replies. This is only true if the network does not include MEL hubs. If hubs are present then it is the hubs responsibility to do that chore, and they in their turn will notify displays (by exception) when anything interesting happens. Hubs use DISPLAY_ALERT messages for display notification purposes: note also that the format of the device status field in a DISPLAY_ALERT message does *not* have the same format as the packet flags field.

The following state machine suggests how a receiver may correctly detect and parse an incoming frame: this is the essential core of the data link layer in Comm485 :-

State	Event	Action
Idle	STX arrives.	State = Wait_Flags.
	other ch arrives.	ignore (discard).
Wait_Flags	STX arrives.	State = Wait_Flags.
	other ch arrives.	Flags = ch.
		CRC = upd_crc(0xFFFF,ch)
		plen = 0
		State = Wait_Body
Wait_Body	STX arrives.	State = Wait_Flags.
	DLE arrives.	State = Wait_ETX
	other ch arrives.	data[plen++] = ch
		CRC = upd_crc(CRC,ch)
Wait_ETX	STX arrives.	State = Wait_Flags.
	ETX arrives.	crc_lo = CRC & 0xFF
		crc_hi = (CRC>>8) & 0xFF
		if (crc_lo .eq. STX) crc_lo = 0x42
		if (crc_hi .eq. STX) crc_hi = 0x42
		State = Wait_CRC_Lo.
	other ch arrives.	data[plen++] = (ch xor 0x40)
		CRC = upd_crc(CRC,ch xor 0x40)
		State = Wait_Body
Wait_CRC_Lo	STX arrives.	State = Wait_Flags.
	ch .eq. crc_lo arrives.	State = Wait_CRC_Hi.
	other ch arrives.	State = Idle (crc error).
Wait_CRC_Hi	STX arrives.	State = Wait_Flags.
	ch .eq. crc_hi arrives.	<signal successful frame receipt>
		State = Idle.
	other ch arrives.	State = Idle (crc error).

The procedure given places the destination address field in the first two bytes of the *data* buffer. A receiver may check the destination address as soon as it is parsed - if the new frame is not addressed to this receiver then the state may be immediately reset to idle, and further incoming bytes from that frame can be discarded; note that this is correct behaviour even if the frame is corrupted. This should also bring an early abort to pseudo-frames started when line noise creates the initial STX; validity checks on flags bits may also help in that regard. **Appendix A contains a definition of the *upd_crc()* function.**

There is currently a limit of 1032 bytes on the size of the message body (ie. not counting framing bytes), prior to DLE-escapement. Robust receivers may allow for larger frames if they wish. Note that the simple state machine given above made no attempt to check for frame buffer overflow: this would of course be required in a production implementation.

2.3 Network Layer

The Network layer of the protocol deals with device addressing, packet routing and packet delivery.

2.3.1 Device Addressing

Each device attached to the network must be assigned a unique sixteen bit address so that packets can be delivered to individual recipients. The network address is formed from a four bit *device class* and a twelve bit *device number,* ie. :-

15 14 13 12	11 10 09 08 07 06 05 04 03 02 01 00	← bit number
Device Class	Device Number	

Note: This document typically shows device addresses in four digit hexadecimal form, eg. $12AB. (the '$' prefix is used here to denote a hex address). In the example given, $1 is the device class, and $2AB is the device number.

The device class is simply a convenience, as it allows you to tell what *type* of device is being addressed by a network message (eg. camera, hub, display), while the device number selects an individual from within that class. A device class of zero means that *all* device types are being addressed. Likewise, if the device number is set to zero then all devices present which belong to the selected class are addressed. If both parts of the address are zero then all devices on the network receive a copy of the message.

Note that it is the full sixteen bit address (eg. $12AB) which must be unique across the network; device numbers (eg. $2AB) need only be unique within the relevant device class.

These are the device class numbers which are currently assigned :-

- 0 - All device classes (open broadcast).
- 1 - Cameras (flame detectors).
- 2 - Control PCs (obsolete).
- 3 - Display PCs.
- 4 - Servers / Virtual devices.
- 5 - Video switchers.
- 6 - Hubs.
- 7 - Relay cards.
- 8 - Cameras (motion sensors).
- 9 to F - Reserved for future expansion.

So, a message sent to address $0xxx will be picked up by all devices on the network, of all classes, whereas a message sent to device class $1xxx will be ignored by everything except cameras.

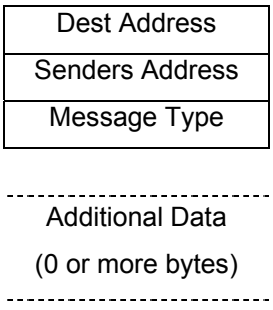
The protocol does not specify how a device determines its own network address. Typically however the device *number* is configured using either a non-volatile memory (a disk file if the device in question is implemented as a PC application), dip switch bank, jumper block or solder bridge header. Implementors should take steps to ensure that devices do not default to, and cannot be configured to use a device number of zero, as that is an illegal address which conflicts with the semantics of broadcast messages. One way to ensure that this is never a problem is for the device to add one to whatever device number is configured. Alternately, the device might default to an agreed "new device" address if the switches are all zero. Neither of these methods work well if more than one unconfigured device is allowed to be connected to the network at the same time; in an ideal world all devices would be preconfigured before they are connected to a live network.

The above description allows certain address combinations which are not necessarily supported, specifically a message sent to device class zero (indicating open broadcast), but with a specific (ie. non-zero) device number nnn. Do not use this address format.

2.3.2 Packet Address Header

Having agreed what a network address looks like, we now need to know where the addressing information appears in a data packet. The packet framing structure described earlier allows a receiving device to distinguish network messages from noise, but does not allow the receiving device to tell if it was the intended recipient of the message. To do that there needs to be an agreed location and format for message addressing information.

Address information is stored as a six byte header placed at the start of the message body, as shown in the following figure. In fact the header includes more than just address information, it also includes a message type and, optionally, may be supplemented by additional data depending on the nature of the message (*Comm485* does not split message content off into a separate layer).



The *destination address* field allows you to tell if you are an intended recipient of this message - this might be a specific device address or a broadcast address. The *senders address* tells you who sent the message (and also tells you where to send any reply), and is always the address of a single device, never a broadcast address.

The *message type* is the number of a supported command or notification message (or an acknowledgement of a command or notification), and if that command requires you to have additional information then the necessary information will immediately follow the basic header. A description of the individual protocol messages is given in chapter 4 and will include the necessary content and format of the "Additional Data" section, where it is required.

Note: Keep in mind that the message header is part of the overall message body as discussed in section 2.2. In particular you should remember that the entire data area of a packet is subject to DLE escapement during transmission, including the header shown above. Do *not* assume that you can copy the first few bytes out of a receive buffer and act on them (say, compare the destination address with your own) without first filtering out DLE sequences.

2.3.3 Message Routing

If the network consists of one long multidrop then no actual routing is required. When one device transmits all other devices on the multidrop receive the message, but only the device(s) which have been correctly addressed will do anything with it.

Note: You probably understand by now that RS485 will not tolerate multiple simultaneous transmissions on one multidrop. For this reason it is usually the case that all messages for which a reply is expected are directed to a single recipient. Broadcast messages *almost* never elicit a reply. See the description of the discovery protocol for the only major exception to this rule.

When greater reach or reliability is required the network will generally be broken up into a number of subnetworks which are connected together using hubs, and it is then that the question of how routing works becomes important. In theory the routing mechanism employed by MEL hubs is fully transparent and automatic, and is certainly not specified by the protocol. In practice you will be able to make better use of hubs and the protocol if you understand how hubs manage the routing task. Unfortunately, a short discussion about the inner workings of MEL hubs will be necessary before we can tackle the subject of message routing directly.

MEL hubs are designed with the assumption that the network is organised into a tree topology (a hierarchy), with a special hub known as the *master hub** at the root, so called slave hubs at intermediate layers, and slave devices such as cameras or video switchers at the leaf nodes.

* The master hub has a dedicated network address of $6001. *Any* hub which finds that it has been configured with this network address assumes that it must be expected to play the role of master hub. The distinction between master and slave hub is thus controlled by configuration - the firmware in each hub is identical.

Each hub in the network forms one node in the hierarchy: one or two ports on every slave hub are reserved as *upstream* ports; the hub uses these ports to receive and respond to polls from the parent hub, the remaining ports on the slave hub are *downstream* ports leading to "child" devices which the hub is responsible for polling . The master hub is only slightly different: it has no parent, so all ports are considered to be downstream ports. The distinction between upstream and downstream ports is not hardwired for specific ports, instead the hub determines for itself which ports are which during the discovery phase. It is also during the discovery phase that correct routing tables are established.

A hub is only responsible for polling its immediate neighbours on downstream ports. We emphasise that a hub does *not* poll every downstream device - only its immediate downstream neighbours. Some of those neighbours may be slave hubs, in which case it is they who are responsible for polling *their* downstream neighbours, and so on until we get down to the level of cameras and other passive devices. Since each hub rarely has more than a dozen or so "children" the polling overhead is both much reduced and constant: unlike the case of the long multidrop the polling overhead is not a function of the total number of devices connected to the network.

Aside from reduced polling overhead, another major benefit delivered by the hub is that each "spoke" which connects a hub port to a downstream device is, electrically speaking, a completely distinct network. Failure of that spoke will only cause loss of comms to devices which are downstream from the affected hub port, the rest of the network remains unaffected, bringing an improvement in both reliability and fault location compared to the "big multidrop" arrangement. Furthermore, the connections between hubs will most often be done inside a single cabinet and will be quite short, making them unlikely to fail. The spokes that are most likely to fail are those which must traverse the plant and are therefore at greater risk of accidental damage and weathering, however in that case only the single device on the end of the spoke would be affected, and localisation of the communications fault is a trivial task.

So far, only the hubs "distributed polling" (data concentration) role has been discussed. However, hubs also operate as routers for messages which need to cross the network, and as repeaters, so a network can be much larger than than the 32 or 128 devices possible on a single multidrop. Hubs are intended to operate transparently as far as routing and repeating is concerned: eg. a display can talk to a camera precisely as if that camera was directly connected to the display, with few clues that there may in fact be several layers of hubs in between.

Routing for all network devices is established during network initialisation. First, the master hub uses the discovery protocol (described in detail elsewhere) to find out what devices are *directly connected* to its downstream ports. Typically these will be all slave hubs plus a couple of displays, or there may be a mixture of hubs, displays and other (slave) devices. For every hub present on a downstream port the master hub sends it an INIT_REQUEST command. This command does two things: first it tells the slave

hub which of its ports is upstream (the port it receives the command on must be its upstream port, all the rest are downstream ports). Second, the INIT_REQUEST command tells the slave hub to begin its own discovery procedure. The master hub will send an INIT_REQUEST to all slave hubs at essentially the same time, causing them to all perform discovery in parallel, ie. the time taken for a complete network reset is a function of the number of levels in the hub hierarchy, not of the number of hubs present in the network. Having told all its neighbouring slave hubs to start discovery, the master hub then goes to sleep until the slave hubs indicate that they have finished.

During discovery a slave hub more or less repeats what the master hub did: it uses the discovery mechanism to find out what devices are immediately connected to downstream ports (a slave hub must *not* attempt discovery on an upstream port, as that could lead to circular routes!), and if any of those devices are hubs it sends INIT_REQUEST messages to them.

INIT_REQUEST thereby propagates downstream, each layer executing the discovery protocol in turn, until we finally reach the level of a "leaf" hub, ie. a hub which does not include other hubs as children; the children of a leaf hub (if any) are all slave devices such as cameras and video switchers. The leaf hub creates a table of all the child devices that it found, the table entry will include the network address of the child device plus the number of the hub port to which that child is attached. The slave hub stores this table in memory for its own use and also passes a copy of it up to the parent hub. The parent hub has no direct access to ports on the slave hub, and so must change the port number of devices in the passed-up table to that of the port which leads to the slave hub. The parent hub merges this amended table with similarly amended tables from any other child hub it knows about, stores the combined table and also passes a copy up to *its* parent hub. Discovery information propagates up the hierarchy in this way until it reaches the master hub, which now has a complete list of all devices present on the network, along with correct port number to use when routing packets to any device.

Having described how routing information is gathered, we can now turn to the actual routing mechanism in the hub.

The hub routing algorithm turns out to be remarkably simple: when a packet arrives on any port the hub examines the destination address. If the destination is a broadcast address then a copy of the packet is queued for transmission on the current upstream port plus all downstream ports which lead to devices with compatible addresses. If the destination is a unique (non-broadcast) address then the packet is routed to the single downstream port which leads to the target device. If the targeted device is not found in the hubs routing table then it is assumed that this is because the device in question is not downstream from the current hub - the packet is therefore routed upstream. The one golden rule, which may override the description given so far, is that a packet is *never* reflected back to the port from which it arrived, and similarly, a packet which arrives from upstream is never routed back upstream (a

separate clause is needed here because there may be more than one physical upstream port, but they needed to be treated collectively as one port).

*Note: hubs never route packets which have the *active* frame flag reset. If a packet arrives at a hub with the *active* flag reset, and that packet is not addressed to the hub, then the packet is ignored and discarded.

MEL hubs do not "store-and-forward" packets when routing them. In other words, the hub does not wait for a complete packet to arrive before switching it to the output port(s). If the hub *did* route in that way then the length of the transmission would be the time taken to transmit the packet across one hop (which depends on the packet length), *multiplied* by the number of hops needed to traverse the network, which could soon mount up to become a very substantial overhead indeed. Instead, MEL hubs perform what is known as "cut-through" routing: basically the packet is routed to the correct output port at the earliest possible opportunity, ie. as soon as the router sees the destination address, contained in the first two bytes of the message body. A per-hop delay is still inserted (because the routing hub has to wait for that first few bytes to arrive), but the overhead involved is relatively small - on the order of a millisecond or so per hop - and is not affected by the length of the packet. Essentially, the packet traverses the network, regardless of the number of hops required, in about the same time as the packet would take to traverse one hop under the store and forward scheme. The one downside of the cut-through strategy is that it is not possible for intermediate hubs to filter out packets which have bad CRCs; by the time the hub sees the CRC the transmission onwards to the next hub is already nearly done, so this style of routing *will* allow bad packets to cross the network.

2.4 Transport Layer

The Transport layer, as defined by the ISO/OSI network model, deals with error recovery, in other words in providing a guarantee that a message transmitted by one node will be delivered to another node. In the *Comm485* protocol there is no distinct error correcting layer. Instead it is specified on a message by message basis whether an acknowledgment should be required for that message (if an acknowledgement is required then a device which receives that message must send the specified acknowledgement message), and it is up to the sending device to take any course of action it wishes if it should fail to receive an acknowledgement. However, we can say that the typical response to a timeout (failure to receive the acknowledgement within a particular time interval) will be to resend the message a limited number of times, say three or five times.

There are few hard and fast rules regarding the length of timeouts. The timeout should generally begin when the initial transmission is completed (not when the transmission starts, otherwise you might time out before the packet has left your transmit buffer!), and should be long enough to allow the message to be delivered, consumed, and for the reply to return.

You should also take *turnaround delays* into account. A device X which has just received a transmission from device Y will generally wait around 5 to 8 ms before sending the reply. This delay is intended to provide ample time for device Y to turn off his RS485 transmit driver and switch into listen mode so that he can hear the reply.

We recommend that you allow about 50 ms for the reply when a minimum length reply is expected (about 12 to 14 bytes), and about 250 ms for maximum length packets. At 57600 bps each byte takes 174 microseconds to transmit, so if you know what size of packet to expect in reply you can easily calculate other timeouts - remembering of course to allow some time for unforeseen holdups.

One further important note about packet timing: MEL devices will not tolerate very long intra-packet delays while receiving a packet. Basically, once you start a transmission you should strive to transmit the entire packet as a continuous stream. If a mid-packet delay is introduced which is more than about 20 ms then receiving devices (and intermediate hubs) are likely to discard the packet.

2.5 Session Layer

The Session layer of a communication protocol deals with how one device establishes and breaks a network connection, how a "virtual circuit" is made to a remote device, and so on. In *Comm485* these issues are mostly inapplicable.

Devices which are attached to the network are discovered by the network, ie. by the hubs, as the hubs start up. Once a device is known to be attached to the network the connection is assumed to be permanent; there is no making and breaking of links. The hubs will recognise when a device leaves the network (the polling hub will lose contact with the device and flag it as having failed), but hubs will *not* automatically recognise the connection of a new device, nor can it handle an existing device being moved from one hub port to another: routing tables are static and distributed up and down the network hierarchy, so the only way to update a routing table is to reset the network (resetting the master hub will cause a full network reset).

The only exception to the above are displays (especially laptop PCs), for which CONNECT and DISCONNECT messages have been provided. Even here however the display must use the same master hub port every time; if the display moves to another port then a network reset is required.

Comm485 does not use "virtual circuits". In other protocols a "virtual circuit" is like a telephone call: first you dial the number to establish all the routing connections, then once the circuit is established you can pipe any number of messages down the connection, and when you're done you hang up the phone. Comm485 does not work like that, because it is not conversation based: each packet is treated as a

separate message, and since it contains complete addressing information the *potential* exists for any packet to take a completely different route than that used by earlier or later packets.

2.6 Presentation Layer

The Presentation layer of the protocol is where we establish standards for the representation of data objects which are to be consumed by more than one device connected to the network. *Comm485* does indeed specify such standards, but we will leave the description of these until we meet them in the context of the message types with which they are associated.

2.7 Application Layer

The final layer of the ISO/OSI network model, this is the layer which implements network applications, for example on the internet it is easy to see that protocols such as telnet and ftp are application layer protocols. In *Comm485* the applications are "flame detector", "hub" and so on, but the protocol does not specify the internal workings of these applications except insofar as it is necessary to exchange data with these applications. Data exchange is a presentation layer issue, and as mentioned earlier, we describe these standard data structures in the context of the message types used to exchange them.

3 The Typical Comm485 Network

A typical MEL Comm485 network* consists of a number of cameras (flame detectors), hubs, video switchers and relay cards, plus two or more display terminals (the latter implemented as applications running on standard WinTel PCs). All network devices are in some sense "smart", ie. they have a CPU - sometimes more than one - and firmware which allows them to perform their class specific role without close supervision from a network master, they also respond promptly to requests for status or other data. This chapter takes a closer look at some of the existing device classes which the Comm485 protocol supports.

* This list of device classes is likely to increase with time.

3.1 Cameras (Flame Detectors) : Device Class $1xxx

The main role of the camera is of course to detect fires. Indeed the basic purpose of the network is to allow cameras to report the presence or absence of fires to display terminals. The camera responds to network messages from its parent hub, or from a display (in fact it will respond to any device which asks it a question, but hubs and displays are the only devices that do). Despite its importance the camera is a passive device, ie. a slave. On powerup and thereafter the camera will transmit nothing on the network except when explicitly asked to do so, eg. when polled for status.

It would be possible for any passive device to "pretend" to be a camera, ie. by using a network address that falls within the camera device class. The device would however need to be able to spoof sensible responses to the specialised requests which might be sent to a camera, for example it should respond to requests for a frame grab. NAKing such requests should be a fairly harmless response. It will be vital of course that the spoof device NAK attempts to update the "camera" firmware, since that firmware will be intended for real cameras!

MEL flame detectors now "latch" alarms, ie. if the camera detects a fire it will set the "alarm bit" in its device flags, and that bit will remain set until either the camera is reset, or until it receives a SET_RELAY(0) message.

3.2 Control PCs : Device Class $2xxx

This device class is a hangover from version 1.x of the protocol, which since the introduction of the MEL hub has become obsolete. New devices should not "pretend" to belong to this class, since they will be ignored by other devices and may be barred from accessing the network. It is also likely that this network class will be reassigned in future.

3.3 Display Terminals : Device Class $3xxx

The "display terminal" device class is a very loose category. Just about any device could give itself a network address that causes it to be included in the "display terminal" device class, and the only consequence of doing so is that the display is given greater latitude when it comes to making transmissions (a display is not expected to remain passive); also the display will automatically receive notifications from the master hub when the status of any network device changes. The cost of this latitude is that displays *must* be directly connected to a master hub port, and cannot share this port with any other device.

The range of messages which a display terminal is required to implement is quite small. Like any other device it must of course support the discovery protocol, and it must respond to STATUS_REQUEST. The main additional requirement is that a display, once it has appeared "online", must transmit at least one packet (to any destination) roughly once every ten seconds or so, which assures the master hub that the display has not failed.

A display terminal normally provides visual indication of the current status of other devices connected to the network, including whether those devices have failed, or (in the case of cameras) have detected a fire. Display terminals are part of the *Comm485* network and have a unique network address stored in a configuration file on hard disk.

In order to provide a graphical indication of network status, the display terminals will of course have to be informed of that status. The display receives this information in two ways, a) by automatic notification from the master hub which occurs when the status of a device changes, and b) by explicitly asking the master hub to return the current status of all network devices.

Display terminals may also have an analogue video connection, fed by a video switcher which is is turn fed by the cameras, in which case the display will be capable of showing live video from a selected camera inside a window. The display can be configured so that it automatically displays the live video feed from a camera if that camera goes into alarm. The Display terminals can also log alarms and other important events.

3.4 Virtual Devices : Device Class $4xxx

Provision has been made in the Comm485 protocol for "virtual devices". These are devices which need to have an apparant presence on the network, and which are addressable like any other device, but which do not necessarily reside inside a dedicated piece of hardware.

For example, suppose we wished to implement a print server as a network function, but did not wish to dedicate a whole PC to this function. We might instead choose to implement the server as a module

within the display application, but only enable the feature in one of the available displays. The question now arises: how does some other device, which wishes to use the print server, know which network address to use? In a hub for example, should we hard code a knowledge of which display address also handles print server messages? Alternatively, do we reserve memory inside the hub to store the display address as a configurable item?

We do neither. Instead we would create a print server as a virtual device, whose location would be discovered by the network like any other device. The fact that this device is "hosted" by hardware whose main function is something else is not important.

Note: As mentioned above, provision has been made in the protocol for virtual devices. However, the only such device so far implemented by MEL has been a "display server" device (with address $4001). A display can use this address when it starts up, ie. the display sends a STATUS_REQUEST message addressed to device $4001. The hub to which the display is attached sees this message and intercepts it, but uses its real network address in the reply instead of the virtual address. Thus a display is able to find out the address of the hub to which it is connected. At present this feature is not needed because we currently require displays to be connected directly to a master hub port; the mechanism describes creates scope to relax this requirement in the future.

3.5 Video Switchers: Device Class $5xxx

Every camera has at least one data connection (RS485) and a video output. The data connection would go to a MEL hub, and the video connection goes to a video switcher. In other words, a video switcher plays a role which is analagous to that performed by the hubs, except that it is routing video instead of data.

The present MEL video switcher has sixteen standard video inputs plus eight further connectors which provide support for up to four shared "video transmission lines".

Conceptually, a single video transmission line (or video bus) is a bit like a multidropped RS485 transmission line. Several video sources may be connected to the line, but only one of these devices can be driving a video signal onto the line at any one time.

Unfortunately, and unlike RS485, it is not possible (at present) for one video line to carry more than one more video signal, so if you have, say, two display terminals then either they must both view the same video pictures, or you need two entirely separate video transmission paths in order to service the two displays. The MEL video switchers have four entirely separate video transmission paths and can consequently service up to four displays (rather, it can service up to four different video consumers).

A single video switcher can route any of its sixteen inputs to any combination of its video outputs. It is important to recognise that adding further video switchers increases the number of video inputs available, but does *not* increase the number of outputs, because the output paths are multidropped among the available switchers.

A final warning about video switchers: given the obvious analogy between hubs and video switchers you might easily imagine that the switchers will implement some sort of discovery mechanism so that video routes can be established automatically. Unfortunately, this is not possible and the video switchers do not work that way. The inputs on a video switcher are *only* inputs, so the switcher cannot use them to "probe" for the identity of the device which is attached to that input. Nor is there any way in which a switcher can identify what device is attached by looking at the video signal that device generates: in fact the video switchers have no access to the video signals: they can route the video, but they have no means to process it. In summary, video routes are not determined automatically, but instead have to be created manually and then stored somewhere convenient. Video routing information must be processed by a non-slave device which is able to communicate with all switchers on the network - in MEL networks it is the master hub which actually processes the video routing information and, on receiving a high level command from a display, sends the appropriate detailed switching instructions to affected video switchers.

3.6 Hubs : Device Class $6xxx

Hubs perform a very specialised role: no device should "pretend" to be a hub. The master hub has a fixed network address, $6001.

Version 1.00 of the COMM485 protocol was built upon a single, simple, RS485 network as described in the chapter 2. The status of connected devices, particularly cameras, was actively polled by a "network controller" implemented as an application running on a PC. However, this configuration proved to be unreliable.

In essence, all devices on an RS485 multidrop are electrically connected in parallel at various points along a twisted pair. The device which is transmitting "drives" a voltage differential onto the two cores of the twisted pair, then modulates this differential to produce a signal which is decoded by the RS485 transceivers of other devices. If a device on the network should fail while still driving the transmitter the result *can* be that all other devices are deafened; ie. a failure at a single point can kill off the entire network, which is clearly unacceptable. Unlikely as it sounds, in fact this fault is not at all uncommon, especially when the RS485 device in question is actually a PC using an RS232/RS485 converter, since in most such cases the transmitter driver must be turned on and off manually (using RTS) - if the PC misses the tx-empty interrupt then RTS can be left stuck high, causing the failure mode described.

Furthermore, once such a failure has occurred it is very difficult to pinpoint the cause, since the fault may be inside any of the devices connected to the network, or in any run of cable between pairs of devices. Essentially, one is required to dismantle the entire network, then put it back together again a section at a time until it breaks, thus pinpointing the location of the failure. Again, this is not ideal.

Finally, the idea of having a single PC poll the status of the entire network turned out to be too cumbersome, and not at all scaleable to larger networks without introducing very long polling delays.

Therefore, MEL eventually scrapped the "control PC" concept and introduced the MEL hub, the latest version of which is the iHUB48516. Each hub has an onboard RISC microcontroller (plus the usual complement of flash ROM and SRAM), and sixteen entirely independent, high speed RS485 compatible comm ports. The independant comm ports and onboard RISC processor allows the network to be rearranged in the tree topology described in chapter 3.

The introduction of the MEL Hub did not require substantial changes to the protocol - the packet format is unchanged and all previously existing message types continue to be supported. Passive devices require no changes whatever.

The only changes are due to the fact that the Hub design has created opportunities for alleviating some of the more onerous restrictions formerly placed on devices. For example, it was previously the case that since all devices shared a common transmission line it was absolutely necessary for a device, such as a display, to wait for transmit permission from the controller before sending requests to cameras, for example to grab a frame of video. This is no longer true: a display can (and now must) be given a dedicated port on the master hub. Since this port is an electrically distinct medium shared by no other devices, and since the hub is aware that a display is attached to that port, the display is in effect granted permanent transmit permission; it can send out packets whenever it likes, and no longer needs to implement complex queuing mechanisms for packets which are awaiting transmit permission. This immediacy gives the display a much nicer, "snappier" performance from the point of view of a human operator.

It was previously the case that the display was a network device like any other, and was regularly polled for status by the network controller, which needed to know that the display was still alive. Again, this is no longer true: the display is now the master of the hub port to which it is attached and it is the display which must poll the hub (if it cares to), to check that the hub is still alive. This reversal of roles is necessary if we expect the display to be able to transmit at any time with a low probability of conflicting with a message coming from the hub.

However, once the hub has become aware of the existance of the display it is necessary for the display to demonstrate at regular intervals that it is still alive, by transmitting a packet either to the hub or to any some other device. The hub implements a very long timeout (about 30 seconds) for display activity, but if that timeout should expire then the hub will in that circumstance transmit a polling message to the display to confirm that it has in fact died.

As part of its polling and data concentration duties, a hub maintains status information for every downstream device: hence the master hub - with respect to which every other device is downstream - has status information for the entire network. Information stored for each device includes the network address, a "friendly name" for that device, and a 16 bit status word. The low nibble of this word contains the base status as follows :-

INACTIVE(0) this device is known (listed), but not included in polls. The most common reason for an inactive status is that a device has been added to the device list, but has not yet been powered up and/or physically attached to the network.

NORMAL(1) this device is active and responded OK to recent polls.

FAILURE(2) this device failed to respond to N (implementation dependant) consecutive polling messages, the device is thus flagged as failed, the human operator has not yet acknowleged the failure.

ALARM-NOTACK(3) this device reported an alarm (eg. fire) condition in the most recent poll, which the operator has not yet acknowledged.

ALARM-ACK(4) this device reported an alarm condition in the most recent poll, which the operator has acknowledged.

If the controller registers a FAILURE state for any device, then it will maintain that state until the operator acknowledges the failure, after which the device will be set to an INACTIVE state. However, the hub will continue to poll the failed device at a reduced frequency and will automatically change the device status from INACTIVE to NORMAL if the device should come back to life. Note that this behaviour may be undesirable in a device which is exhibiting an intermittent fault: the device would fail (producing an audible alarm at the display), the failure is acknowledged by the operator, the device goes offline and comes back again, only to fail again, requiring a repeated acknowledgment from the operator. This irritation can be circumvented by inhibiting the device, which will prevent subsequent failures from generating alerts.

Bits 4..7 of the status word may be ORed into the base status to indicate status modifiers. Currently assigned status bits are :-

INHIBITED (bit 4) set if the device failure and alarm alerts are inhibited on this device.

HASMASK (bit 5) this bit applies to cameras only, and indicates (if set) that a portion of the scene viewed by the camera has been "masked off", ie. the camera has been told to ignore part of its field of view: the problem of a partial view of the flare could have been solved using this feature.

FLAREMON (bit 6) this bit applies to cameras only, and indicates (if set) that this camera has been designated as a "flare monitor". A flare monitor is a flame detector that intentionally watches the flare, and while it does generate alerts when it sees the flare in operation, the system does not treat such alerts as candidates for alarm or control actions.

Bit 7 is reserved for future use.

Bits 8 and 9 of the status word may be ORed into the base status to indicate the current warning (internal diagnostic) status of the device. Warnings are a new feature in version 2.5 of the *Comm485* protocol. Bits 8 and 9 should be interpreted as follows :-

b9=0, b8=0	no warning is currently being reported.
b9=0, b8=1	warning reported, not yet acknowledged by operator.
b9=1, b8=0	warning has been reported and acknowledged.
b9=1, b8=1	reserved for future use.

See a description of the new WARNING_RESET and WARNING_REQUEST messages for a description of how to get further details about the diagnostic condition being reported. Note for implementors: if your display software was developed for earlier versions of the protocol and cannot easily be updated to support them then warnings can easily be ignored.

Note: the "per device" status word maintained by the hub is *not* the same as the device flags included in the frame of every packet transmitted by the given device. While the hub does admittedly derive the device status mostly from the device flags, the hub also incorporates other tidbits which the originating device knows nothing about, eg. a camera does not know, or care, that an alarm has been acknowledged by a human operator, or that alarms may have been inhibited, and of course, a device can hardly indicate for itself that its comms link has failed.

3.7 Relay Cards

The MEL Relay card is a slave device having a somewhat underutilised RISC microcontroller, two RS485 network ports, and sixteen relay outputs. The relay card is configured using CAMTHRESH messages, the description of that message will describe the configuration options available on the relay card.

3.8 Motion Sensors

A motion sensor is a recently introduced device class. It consists of a camera (the hardware is identical to the flame detection camera, device class $1xxx), but with a different ROM which enables the hardware to do motion sensing instead of flame detection.

4 Protocol Messages

4.1 General Information

Chapter 2 described the overall format of message packets, including the six byte address information header which is common to all messages and appears at the beginning of the message body in every transmitted packet. In this chapter it is assumed that you already know all about that, so the need for the address header will *not* be reiterated in the individual message descriptions of this part of the specification. When a message description says that this or another data parameter is required, we mean that it is required *in addition* to the fixed header. If the description says that no parameters are required then we actually mean *nothing other than the fixed header*.

The purpose of this chapter is to list the symbolic names, "message type" number and structure for all protocol messages supported in version 2.5 of the MEL *Comm485* protocol. Each description begins with a C-like function definition which includes the message symbolic name, the associated "message type" ID number in square brackets immediately following the name, and function parameters shown in parenthesis; empty parenthesis implies that no additional information is required in the message body.

Parameter types precede the parameter name in the function definition, just like an ordinary function prototype from ANSI C. The following parameter types are used :-

char	-	A 7 bit ASCII character value passed in an eight bit byte.
byte	-	An unsigned 8 bit integer.
word	-	An unsigned 16 bit integer.
dword	-	An unsigned 32 bit integer.
short	-	A signed 16 bit integer.
int16	-	A signed 16 bit integer.
int	-	A signed 32 bit integer.
int32	-	A signed 32 bit integer.
long	-	A signed 32 bit integer.

Array parameters may also be used, eg. a declaration of "char s[80]" defines an 80 character buffer usually containing a nul terminated string. Larger buffers may be used for raw data of various sorts, such as bitmaps, bit masks, and software updates. An empty array parameter such as "int data[]" implies an array containing a variable number of elements; sometimes a preceding integer parameter provides the actual length of the array, in others cases the array will be terminated by the end of the message body, the latter being possible only if the array is the last parameter.

C like structure definitions may also be used to describe the format of a message body. Bear in mind however that if you choose to directly paste this structure definition into a high level language you must remember to consider the alignment conventions used by your compiler: most modern C compilers use the convention that structure fields should be aligned on a boundary which has the same granularity as the field itself, ie. byte fields will be aligned on byte boundaries, 16bit values on 16bit boundaries and so on. Where possible the structure defined in this document will use explicit padding between fields of different sizes so that varying structure field alignment conventions are not a problem.

Message parameters should be copied to the message body in declaration order from left to right, and should immediately follow the standard six byte address header. Integer values more than one byte in size should be transferred least significant byte first (ie. "Intel format").

There is *no* separation of the protocol into layers, such as "core" and "optional". In theory at least, all devices implement all messages. However, bear in mind that most network devices are passive - they initiate no conversations on their own, responding only to polls. Other devices which do send out polling messages need only be able to handle the reply messages which they themselves have requested. Hubs do initiate messages, however if a new device simply implements the discovery protocol described in chapter 5, and sends a STATUS message in response to a STATUS_REQUEST from the hub then that will be sufficient to provide the device with a basic network connection. We do recommend that all devices respond to the PING_REQUEST message as well; although optional this message is very useful when diagnosing network problems, as it allows us to conduct more rigorous comms tests with the device.

4.2 Supported Messages

4.2.1 CONTROLLER_ACTIVE[0]( )

The master hub broadcasts this message when it first goes online, ie. after it has finished initialising the network using the discovery protocol described in chapter 5. One typical response is that of the standard display software, which responds to this broadcast by requesting a copy of the master hubs "discovery list" (the list of devices it detected during discovery). The display uses the difference between the discovery list and its own stored device list to detect whether any device failed to come online during the reset.

See also: CONTROLLER_RESET.

4.2.2 STATUS_REQUEST[1]( )

The STATUS_REQUEST message requests that the addressed device return a STATUS message to the sender, with the "flags" field in the reply frame correctly reflecting the current status of the addressed device. STATUS_REQUEST is primarily intended to be used by parent hubs when polling their downstream children, however it may also be used by any device when it wishes to check that some other device is "alive and well" before beginning a more complex conversation.

All network devices are required to be able to correctly respond to STATUS_REQUEST, as failure to do so would cause the parent hub to mark the errant device as faulty.

See also: STATUS.

4.2.3 RESERVED_MSG1[2]( )

The RESERVED_MSG1 is not a real message; it will never be transmitted by any device. The purpose of this section is to openly document the fact that message number two is reserved for efficiency reasons, ie. the value two equals STX, and would need to be DLE escaped every time it was used.

4.2.4 STATUS[3]( )

The STATUS message is sent in reply to a STATUS_REQUEST message. There is no additional data to send, the important status information is carried by the frame flags.

See also: STATUS_REQUEST.

4.2.5 PREPARE_DATA[4](word iDataSubtype)

Most data structures which you would want to retrieve from another device are obtained using an explicit request message which elicits an immediate reply packet whose body contains the requested data. However, that simple form of transaction cannot be used if the data structure in question (say, a grayscale image) is likely to be larger than then maximum allowed packet size. In that case the data in question will have to be cached by the source device for you to read out in chunks. The PREPARE_DATA message is used as the first step in retrieving such large data structures from the target device - it orders the target device to place a particular data structure in its input/output cache. Note that there is only one cache in each target device, ie. each new PREPARE_DATA call overwrites anything that was previously stored there, as does *sending* data to the device using the BEGIN_TRANSFER mechanism.

A PREPARE_DATA message is sent, with the iDataSubtype parameter indicating exactly what data is requested. The target responds with NAK(PREPARE_DATA) if the requested data is not available or not supported, otherwise it responds with ACK(PREPARE_DATA, nBytes) where nBytes is a word value giving the size in bytes of the cached data. In some cases the data is larger than 64Kb and so cannot be represented by the 16bit nBytes value - in that case some other method will have been

provided to enable you to determine the data size - the description of each supported iDataSubtype parameter will indicate how the object size is obtained, if it is not via the nBytes method.

If ACK is received then the originator may then proceed to extract the data from the target device using the appropriate number of DATA_REQUEST messages, each of which causes a 1k block of data to be returned inside a DATA message wrapper.

The following are the values of iDataSubtype currently supported. Any value not mentioned is either reserved for future use, is obsolete and withdrawn, or is used inside MEL for device diagnostics, testing, debugging etc (contact MEL if you wish to reserve a new data subtype which will not conflict with existing types) :-

0x00 (PREPDATA_CAPTURE) - no longer supported (replaced by PREPDATA_RAWCAPTURE for frame grabs from mark II camera).

0x02 (PREPDATA_BITMASK) - no longer supported (replaced by PREPDATA_BITMASKEX).

0x03 (PREPDATA_RAWCAPTURE) - this data type is supported by cameras only. The camera grabs the next video frame and stores it in the cache for collection. The camera returns the video frame as a 352x289x8bpp grayscale image, which is immediately preceded by a 3-dwords header structure as follows :-

```
typedef struct {
    dword Signature;            /* 0x004D454C              */
    dword image_width;/* = 352 in mark II camera.        */
    dword image_height;         /* = 289 in mark II camera.        */
} IMAGE_HEADER;
```

Note that you have to fetch the first page of the data in order to see this header. For compatibility with future MEL cameras you should not assume the image size given here, but should instead check the size given by the header. The nBytes parameter should be ignored when retrieving this data structure.
0x0B (PREPDATA_JPEG) - once again, this data type is supported by cameras only. The camera grabs the next video frame and stores it in the cache as a compressed 352x288 grayscale image. The compression algorithm is baseline JPEG: the data retrieved can be written directly to a .JPG file on disk and decoded by any JPEG compatible image viewer (all required JPEG header information is included). Note once again that for compatibility with future MEL cameras you ought not to assume the image size given here: instead you should extract the image size from the JPEG-standard data stream.

0x0C (PREPDATA_THRESH) - (cameras only). The camera grabs the next video frame and stores it in the cache as an RLE coded thresholded bitmap (352x289x1bpp), along with some camera diagnostic information. The general format of the data is :-

```
typedef struct {
   <diagnostic_data>;    /* variable length */
   <rle_data>;           /* variable length */
} thresh_data;
```

The <diagnostic_data> information is not useful outside MEL, you do however need to know how to skip over it to get to the <rle_data> - here is a rough guide to the layout of the diagnostic info :-

```
typedef struct {
   int32 nObjects;
   /* following fields not present if nObjects <= 0 */
   int32 lenObject;
   byte obj_data[nObjects*lenObject];
} diagnostic_data;
```

The <rle_data> structure is as described in appendix B.

Uniquely, the camera responds to a PREPARE_DATA(PREPDATA_THRESH) instruction with an immediate DATA message containing the thresholded frame, instead of following the usual PREPARE_DATA - ACK - DATA_REQUEST - DATA sequence used by other PREPARE_DATA subtypes. This exception was made for performance reasons, as it allows a "pseudo live video window" to update a PC representation of the thresholded view in close to the real frame rate, bypassing the additional turnaround delays of the standard sequence.
0x1nn (PREPDATA_DATAOBJECT). Data subtypes 0x100 to 0x1FF are reserved to indicate data objects which are stored in the non-volatile memory (flash or EEPROM) of the target device. The last two hex digits (nn) of the data subtype number indentifies which particular data object is sought, acting rather like a filename for the object.

Every data object stored in flash has a version information header structure associated with it. This structure is also stored in flash, immediately before the data object itself. The version info header serves a similar function to that of the directory entry data associated with disk files on your PCs disk drive. This header is returned along with the data object when you retrieve an object, and must be prepended by you to any data object you wish to store. All data objects stored in non-volatile memory share the following generic format :-

```
typedef struct {
   <S_VERINFO header>;
   <object_specific_data>;
} generic_data_object;

typedef struct {
   word  reserved;
   word  len;
   word  objId;
   word  prjId;
   word  fmtVer
   word  timeStamp[2];
   word  CRC;
   byte  additional_data[len];
} S_VERINFO;
```

len is the number of additional data bytes which follow the header, for example if the data object referred to a disk file then the additional data bytes could contain the file name. *objId* identifies the specific data object and is the same as the "nn" code used to retrieve it. *prjID* is a "project code" which indicates which specific MEL client the data relates to; this is used as part of a version checking scheme to ensure that (eg.) device lists for one client are not accidentally propagated around the network of a different client. *fmtVer* is a version number for the data format used in the <object_specific_data> portion. *timeStamp* is date/time stamp (the first word contains the time, the second word the date), the representation of which is compatible with that used to timestamp files in MS-DOS* (Y2K note: the year field of this format will not overflow until 2108). The timeStamp allows further version checking - in the case where two devices hold copies of the same data object, the timeStamp allows us to determine which copy is more recent. *CRC* should allow even more version checking, but is not currently used - it is always zero.

* For those who don't have access to an MS-DOS programmers reference, here is the format of the time and date words. Time word: bits 0..4 is the number of seconds divided by 2, bits 5..10 gives the minutes, bits 11..15 gives the hour (24hr clock). Date word: bits 0..4 gives the day of the month (1 = 1st day), bits 5..8 gives the month (1=January), bits 9..15 give the calendar year minus 1980.

0x100 (PREPDATA_MDEVLIST) - (currently hubs only). Note that this is a subtype of the PREPDATA_DATAOBJECT subtype. This data subtype is used to get or set the master device list structure on the master hub. The "master device list" is a manually constructed list of all the device which the master hub *should* find connected to the network, along with some associated information for each device. Note that this device list is not the same the actual device list which the hub determines for itself during discovery. The master hub actually makes very little use of the master device list (the information is more useful to a display than to a hub), and is stored on the hub only because the hub

makes a convenient central location from which all displays can load the same (latest) device list. The data format is as follows :-

```
typedef struct {
   <S_VERINFO header>;
   word nDevices;
   word spare;           /* pad to dword boundary */
   DEVICE dev[nDevices];
   char StringTable[];   /* variable length */
} MDEVLIST;

typedef struct {  /* an entry in the master devlist */
   word  DevAddr;        /* device network address */
   word  sref_DevName;   /* index of device name string */
   word  sref_MimicName; /* index of mimic name string */
   word  Subnet;         /* top nibble used for hardware rev.*/
   dword VideoRoute;     /* video routing information */
   dword RelayRoute;     /* associated relay output */
} DEVICE;
```

Note that *sref_DevName* and *sref_MimicName* are indices into the *StringTable* array, respectively giving the device "friendly name" and the name of the default mimic file for the device. The length of *StringTable* is not given explicitly, but *is* given implicitly; ie. it is the difference between the size of the above structure and the data structure length indicated by the ACK(PREPARE_DATA, nBytes) message. *Subnet* should be set to zero.

The *VideoRoute* field tells you where the video output (from the device whose devlist entry this is) goes to. The coding of VideoRoute is described in appendix C. *RelayRoute* associates the device list entry with a relay output. If RelayRoute is zero then there is no associated relay, otherwise LOWORD(RelayRoute) is the network address of a relay card, HIWORD(RelayRoute) is the relay output number on that card (1..16 on current relay cards).

0x102 (PREPDATA_BITMASKEX) - (currently cameras only). Note that this is a subtype of the PREPDATA_DATAOBJECT subtype. This data subtype is used to get or set a *bitmask* from or to a camera. A bitmask is a run length encoded bitmap used to mask off portions of the field of view of the camera, causing the camera to enclude the masked portions from flame detection coverage. For example, if a flare was directly visible in a corner of the field of view of a camera then a bitmask would be the ideal way to prevent that from causing unwanted alarms. The bitmask data object is stored in flash using the following format :-

```
typedef struct {
   <S_VERINFO header>;
   <rle_data>;
} BITMASK;
```

The S_VERINFO structure is as described above. The <rle_data> section is encoded as described in appendix B.

See also: ACK, NAK, DATA_REQUEST, DATA.

4.2.6 DATA_REQUEST[5](word iPage)

The DATA_REQUEST message should only be used to retrieve data prepared by an immediately preceding PREPARE_DATA message. DATA_REQUEST(iPage) asks the target device to return the contents of page 'iPage' of the input/output cache. Each 'page' is a 1k block of data, the first page being number 0. The camera responds to this request with a DATA message.

See also: PREPARE_DATA, DATA.

4.2.7 ACK[6](word idMsg, word AdditionalInfo)

This is a generalised acknowledgement message, and is sent in reply to any of several messages for which an acknowledgment is expected, usually indicating successful receipt and/or processing of the original message. The *idMsg* parameter identifies the message which is being acknowledged. The interpretation of *AdditionalInfo* depends on the message being acknowledged, and in fact is not used at all for many messages - if the description of a message says that the response is ACK(msg) then the additional info word is not used. See the description of PREPARE_DATA for an example of what happens when it *is* used.

See also: NAK.

4.2.8 BEGIN_TRANSFER[7](word nPages, word idPurpose, dword timeStamp, word fileCRC)

We have seen earlier how PREPARE_DATA is the starting point for fetching large data structures from a device. BEGIN_TRANSFER serves much the same purpose, but in the opposite direction, ie. this message informs a specific target device that the sender wishes to transmit one or more pages of data into its input/output cache.

nPages is the total number of 1k pages which will be sent.

idPurpose is a code indicating the purpose of the transmission; assigned values are :-

0	-The data contains a flash ROM (firmware) update.
1	-The data contains a new bitmask (cameras only).
0x100..0x1FF	-Data object update, for storing objects in the flash memory of the target device (see PREPARE_DATA for discussion of PREPDATA_OBJECT, the other half of this feature).

If the transfer is successfully completed then the receiving device will automatically process the data in the manner made appropriate by the value of *idPurpose*.

timeStamp is currently ignored and should be passed as zero.

fileCRC is an additional safety feature added in version 2.5 of the Comm485 protocol. It consists of a 16-bit CRC of the entire data object which is to be transferred, as it will appear when stored in the cache (this CRC is calculated in the same way as a message packet CRC, see appendix A). When the data transfer is complete the target device will compare the CRC passed to BEGIN_TRANSFER with one calculated from the received data. If the CRCs fail to match then the target device will refuse to write the offending data to non-volatile memory, and will set a diagnostic (warning) flag to say so. This is especially important if the data in question is a firmware update, as this additional check ensures that the target device will never trash its own bootstrap ROM in the field, even if corrupted data does somehow manage to get through the packet-level CRC checking.

The received data will be used to update the target firmware, bitmask memory, or stored data object as appropriate, once (and only if) the transfer is successfully completed. On receiving the BEGIN_TRANSFER message the target device should initialise any internal variables which it may need during the transfer, eg. the variables which keep track of whether any pages of data get lost. Having done this the target device sends ACK(BEGIN_TRANSFER) to signal that it is ready to receive.

See also: DATA, TRANSFER_COMPLETE, DATA_ERROR.

4.2.9 DATA[8](word iPage, char buff[])

The DATA message is used to transfer data to or from the input/output cache of the target device. For the purpose of data transmission the cache is partitioned into 1024 byte "pages" numbered from zero up to whatever is needed to cache the data.

To read data from the device: first send a DATA_REQUEST message, passing the number of the cache page you wish to read. The target devices returns the requested data as a DATA message, in which the *iPage* field can be used as confirmation that the page returned is the one requested. The buffer *buff* contains the data itself, and is always exactly 1024 bytes long except for the last page, which may be smaller.

To write data to the device: You should have sent a prior BEGIN_TRANSFER message giving the purpose of the transmission, and this will have been acknowledged by each individual device which is expected to process the data. *iPage* is the index of the cache page in which the device should store this

page of data. The data block *buff* in this message is exactly 1024 bytes long for all pages except the last, which may be any length to a maximum of 1024 bytes. Note that in the case of firmware updates it is normal to broadcast this DATA message to all participating devices, allowing all to be updated simultaneously, hence no acknowledgment of the DATA message should be expected from the receiving device (a device knows that it is participating in the transfer because it previously acknowledged the BEGIN_TRANSFER message). However, the receiving device should keep track of whether any pages were lost (by checking the *iPage* sequence number), so that it can respond to a future TRANSFER_COMPLETE message.

Implementors should note that error recovery (perhaps requested by another device) involves retransmission of pages, which may cause some of the receiving devices on the network to receive duplicate pages, and it is furthermore possible that this could happen even *after* the target device believed the last transmission to have been successfully concluded. Any duplicate pages received during a transmission, and any page at all received when there is no expected transmission, should be discarded and ignored.

See also: DATA_REQUEST, BEGIN_TRANSFER, TRANSFER_COMPLETE, DATA_ERROR.

4.2.10 TRANSFER_COMPLETE[9]( )

This message is sent to a specific device which has previously received and acknowledged a BEGIN_TRANSFER message, which should also have received a stream of broadcast DATA messages, and informs the target device that all pages have now been transmitted. The target device responds with ACK(TRANSFER_COMPLETE) if it did receive all pages promised by the BEGIN_TRANSFER message, otherwise it responds with a DATA_ERROR message which requests the retransmission of selected pages. In the latter case the transmitter will send another TRANSFER_COMPLETE message once it has finished the retransmission. If the target device agrees that it has received all expected pages then it may carry out the NVR reprogramming or bitmask storage, as requested in the original BEGIN_TRANSFER message, before sending back the ACK reply.

See also: DATA, BEGIN_TRANSFER, DATA_ERROR.

4.2.11 DATA_ERROR[10](word PageMap[])

See description of TRANSFER_COMPLETE. If a device has received the latter message, but did not receive all the pages promised by the original BEGIN_TRANSFER message, then it informs the source device of that fact using a DATA_ERROR transmission. The parameter 'PageMap' is a variable length array of words in which each bit of each word represents a single page. The bit is 1 if the corresponding

page was received, 0 otherwise. Bit 0 of word 0 represents the first page of the transmission (page 0), and so on. The source device will then use DATA messages to retransmit the lost pages.

Note that the source device may *broadcast* transmitted and retransmitted pages. This allows other receiving devices which missed the same pages to repair the situation without requesting further retransmissions of the same data.

See also: DATA, BEGIN_TRANSFER, END_TRANSFER.

4.2.12 DEVLIST_REQUEST[11]( )

This message has been withdrawn (new implementations should use PREPARE_DATA with a data subtype PREPDATA_MDEVLIST).

4.2.13 CONNECT[12](word dev_address)

This message is sent by a display terminal to the master hub, asking it to change the status of a device from INACTIVE to NORMAL. The master hub does so, if it can establish comms with the selected device. No explicit acknowledgment is transmitted for this message, instead the hub sends DISPLAY_ALERT messages to all connected displays (including the display which originated the CONNECT message), when the status of the new device changes from inactive to active.

NOTE: This message is not implemented in the mark II (16-way) hub. It didn't work well in the mark I hub either. The best way to "connect" a new device is to do a full network reset: the new fast discovery mechanism means that this isn't nearly as cumbersome as it used to be.

See also: DISPLAY_ALERT.

4.2.14 DISCONNECT[13]( )

This message is sent to the master hub by a device (in practice - a display) which wishes to perform an "orderly disconnection" of itself from the network; the message informs the master hub that the sending device should be removed from the polling list. This "friendly" disconnection avoids the need for the parent hub to make fruitless attempts to reestablish communication with the device or post a misleading "device failure" alert to remaining display terminals. The master hub responds to this message by sending ACK(DISCONNECT) to the requesting device, and DISPLAY_ALERT messages to remaining displays.

See also: ACK, DISPLAY_ALERT.

4.2.15 GET_NAME[14]( )

This message has been withdrawn. It was intended for use with the obsolete PC controller device class, and has been superceded by the ability to obtain the complete master device list from the master hub.

4.2.16 SET_NAME[15]( )

This message has been withdrawn. It was intended for use with the obsolete PC controller device class, and has been superceded by the ability to store a complete master device list on the master hub.

4.2.17 RESERVED_MSG2[16]( )

This is not a real message; it will never be transmitted by any device. The purpose of this section is to openly document the fact that message number sixteen is reserved for efficiency reasons, ie. the value sixteen equals DLE, and would need to be escaped every time it was used.

4.2.18 DISPLAY_ALERT[17](S_DISP_ALERT devstat[])

```
typedef struct {
   word dev_address;
   word status;
} S_DISP_ALERT;
```

This message is sent by the master hub to all display terminals (one at a time, ie. not a broadcast), informing them of a change in the status of one or more devices. The message body contains a variable number of S_DISP_ALERT structures, each of which contains two fields; the number of elements in the array can be calculated from the size of the message body. The *dev_address* field contains the network address of a device whose status has changed. The *status* field is one of the values described in section 3.6. It is up to the display terminal to decide how to represent a change in status to the user. The display should send an ACK(DISPLAY_ALERT) in acknowledgement that it received the DISPLAY_ALERT message. Note that ACK(DISPLAY_ALERT) is not the same as an OPERATOR_ACK.

4.2.19 OPERATOR_ACK[18](S_DISP_ALERT ackstat[])

A display terminal sends OPERATOR_ACK to indicate that a human operator has acknowledged one or more device failures or device alarms as indicated by previous DISPLAY_ALERT messages. The message body consists of a variable length array of S_DISP_ALERT structures. The fields in each structure indicate which device and device status is being acknowledged by that structure. Note that this message is not the same as ACK(DISPLAY_ALERT) - the latter simply indicates that the display terminal application received the DISPLAY_ALERT message, whilst OPERATOR_ACK implies that the new device status has been seen and accepted by a human operator.

See also: DISPLAY_ALERT.

4.2.20 DEVLIST[19]( )

This message has been withdrawn. It was intended for use with the obsolete PC controller device class, and has been superceded by the ability to obtain the complete master device list from the master hub, using PREPARE_DATA with the PREPDATA_MDEVLIST data subtype.

4.2.21 SELECT_VIDEO[20](word idSource, word idDest)

This message number can be processed by three types of device - hubs, video switchers and cameras. However, the interpretation of the arguments is different for each.

Hubs : The master hub receives SELECT_VIDEO messages from displays. In this case the *idSource* argument is the network address of the camera whose live video feed the display wishes to view. The *idDest* argument is ignored by the hub, because the destination for the video switch is assumed to be the display itself. For this feature to work the master hub needs a copy of the static "master device list" stored in its non-volatile memory. The master device list provides the information required to calculate video routes (see appendix C). If the master hub fails to switch the video correctly then likely causes are that *idSource* does not identify a camera which the hub knows about (the camera was not discovered, or is not present in the master device list), or the requesting display has not been assigned a video output in the master device list, or the hub doesn't have a master device list at all.

The master hub replies to this message with an ACK_SELECT_VIDEO message (if all goes well), which reports to the display both the camera address requested, *and* the camera address actually selected. As mentioned, the latter may be different if the requested camera is unknown, or not connected to the video switcher bank. Deliberately requesting an illegal camera address (eg. 0) allows the display to determine the address of the camera whose video is already being routed to it, without causing a video switch.

Video switchers : when the master hub has worked out a video path between camera and display it then sends individual SELECT_VIDEO messages to the video switchers. The video switcher interpretation of the SELECT_VIDEO message is more primitive: *idSource* is a video input number (1-16 for a normal video input), and *idDest* is the video output number (0-3). To select one of the passthrough inputs on the switcher you set *idSource* to 0 and *idDest* to the output number (passthrough-x can only be routed to output-x). The video switcher responds with ACK(SELECT_VIDEO) - note *not* the same as ACK_SELECT_VIDEO!

Cameras : interpret the SELECT_VIDEO message as an instruction to enable or disable their video output. If the *idSource* argument is non-zero then the video output is enabled, else it is disabled. The *idDest* argument is ignored. There is a CAMTHRESH option to control whether the cameras video defaults to enabled or disabled at powerup. The camera responds with ACK(SELECT_VIDEO).

See also: ACK, ACK_SELECT_VIDEO.

4.2.22 SUPPRESS[21](S_DISP_ALERT sda)

This purpose of this command is to allow display PCs to set or reset the INHIBITED attribute of the status of a particular device. However, this command in fact allows the display PC complete power to override the status of a device - at least until the parent hub next polls that device and rediscovers the true status for itself. The master hub responds by sending DISPLAY_ALERT messages to all displays, include the display which sent the SUPPRESS command.

4.2.23 RESET_DEVICE[22]( )

This message is sent directly to a camera, hub, video switcher or other network device, causing it to perform a full firmware reset. The subject device replies with an ACK(RESET_DEVICE) message. Note that this is a full firmware reset, not an alarm reset. You should try to avoid sending firmware reset commands unnecessarily, as the device in question loses anything it has learned as part of the reset (eg. a flame detector will forget previous classifications for suspicious objects, and a hub will lose error statistics and video routing records).

4.2.24 CAM_INFO_REQUEST[23]( )

Originally intended only for cameras (hence the message name), this command now works for all embedded devices (hubs, cameras, video switchers, relay cards), and returns basic basic information about the current settings for that device. The camera replies with a CAM_INFO message.

4.2.25 CAM_INFO[24](S_CAM_INFO sci)

Originally intended only for cameras (hence the message name, and the names of several of the structure fields), this feature now applies to all embedded devices, ie. hubs, cameras, video switchers and relay cards, and is used to return basic information about the device in response to a CAM_INFO_REQUEST message. The message body has the following structure :-

```
typedef struct {
   word  dev_address;
   word  av_intensity;
   dword cam_status1;
   dword cam_status2;
   word  ROMcrc;
   word  ROMversion;
   char  device_type[20];
   short Temperature;
   short TempSetPoint;
   word  HeaterSetting;
   short spare;
} S_CAM_INFO;
```

However, the interpretation of the fields in this structure varies with the device type :-

Visual Flame Detectors:

dev_address is the cameras network address.

av_intensity is the average pixel intensity (0..255) of the most recent video frame.

cam_status_1 is a bit vector which is used to return current settings of the exposure control feature in the sensor (the odd layout of bits is a feature of the VM5426 sensor in the mark I camera): b0..b1 contains the two lsbs of the coarse exposure value, b2..b6 is the gain value (minus one, ie. 0 means a gain of 1), b7 is unused, b8 is the heater control enabled flag (mark II camera only), b9 if set indicates that the camera is using an ADV7176 video encoder, otherwise it is using an ADV7171 video encoder. b10..b15 is unused, b16..b24 is the fine exposure value, b25..b31 has the seven msbs of the coarse exposure value.

cam_status2 is not used with the current sensor.

ROMcrc is a CRC of the cameras onboard flash ROM, calculated as elsewhere.

ROMversion is the version number of the onboard firmware.

sensor_type contains a NUL-terminated string describing the device. In the mark II camera this string is "MEL-Intcam Mark II".

Temperature is the current temperature (degC) as read by the temperature sensor in the mark II camera.

TempSetPoint is the "set point" for temperature control. If heater control is enabled then the mark II camera will use to heaters to keep the temperature inside the camera enclosure close to this setpoint.

HeaterSetting says how hard the heaters are currently being driven, expressed as a percentage ie. 0=0%=heaters off, 100=100%=heaters full on.

Hub:

dev_address is the hub's network address.

av_intensity is used to return the hubs configuration flags: b1..b0 contains the "hub role" (00=slave, 01=primary, 10=master, 11=reserved). b3..b2 contains the "dual hub mode" setting (00=not dualled, 01=default standby, 10=default active, 11=reserved).

cam_status_1 is used to return the rom CRC of the "other" hub (ie. the hub currently on standby) in a dual hub arrangement - this is zero if the hub is not dualled.

cam_status2 is not used.

ROMcrc is a CRC of the hubs onboard flash ROM, calculated as elsewhere. In the case of a dualled hub this is the CRC of the currently active member of the duo.

ROMversion is the version number of the onboard firmware.

sensor_type contains a NUL-terminated string describing the device. In the mark II hub this string is "MEL-iHUB48516".

Remaining fields are unused.

Video switcher:

dev_address is the video switcher's network address.

av_intensity is not used.

cam_status1 is used to return video switching information: b0..b7 contains the video input number (0 to indicate the loop-through input, otherwise 1..16) which is currently being routed to output 0. b8..b15, b16..b23 and b24..b31 do likewise for video outputs 1,2 and 3 respectively.

cam_status2 is not used.

ROMcrc is a CRC of the switchers onboard flash ROM, calculated as elsewhere.

ROMversion is the version number of the onboard firmware.

sensor_type contains a NUL-terminated string describing the device. In the video switcher this string is "MEL VSwitch Mark II".

Remaining fields are unused.

Relay card:

dev_address is the relay card's network address.

av_intensity is not used.

cam_status1 is used to return the current state of the relays. b0 is 1 if relay1 is energised. b1..b15 do likewise for relays 2..16.

cam_status2 is not used.

ROMcrc is a CRC of the relay card onboard flash ROM.

ROMversion is the version number of the onboard firmware.

sensor_type contains a NUL-terminated string describing the device. In the relay card this string is "MEL IRelay Mark I".

Remaining fields are unused.

Visual Motion Sensor:

Identical to the Visual Flame Detector except that field *sensor_type* contains the null terminated string "MEL iMotion".

When implementing a new device the important fields to get right are the *dev_address*, *ROMcrc*, an *ROMversion* fields. It is also useful to put something reasonably descriptive into the *sensor_type* string. The remaining fields can be used as desired, bearing in mind that your display application may need to be updated to interpret them correctly.

4.2.26 ACK_SELECT_VIDEO[25](word cam_requested, word cam_selected)

This message is sent by the master hub in reply to a SELECT_VIDEO message from a display. The parameters report, respectively, the address of the camera whose video feed was requested, and which camera was actually selected.

See also: SELECT_VIDEO.

4.2.27 NAK[26](word iMsg, word AdditionalInfo)

This message is a generalised negative acknowledgement message, the counterpart of ACK; see the description of the latter for further information.

4.2.28 ISESSION_CONTROL[27]( )

4.2.29 ISESSION_DATA[28]( )

Do not use these messages. They are used within MEL during development of new camera firmware revisions to continue "interactive sessions" with a firmware debugger running in the camera. Non-debug versions of the camera are unlikely to handle these messages properly.

4.2.30 TESTSIG_REQUEST[29]( )

This message is obsolete. The original purpose was to allow an engineer to debug comms problems using a scope, by having the camera generate a long transmission consisting of repetitions of the same character. The "ping test" functions are more useful for ordinary comms testing.

See also: PINGTEST_REQUEST, PING_REQUEST.

4.2.31 VERINFO_REQUEST[30](word objId)

A display sends this message to an embedded device when it wants to check version information for a data object stored in flash ROM on that device. The *objId* parameter identifies the relevant data object (see the description of generic data objects in the description of PREPARE_DATA). The devicereplies with a VERINFO message, or NAK(VERINFO_REQUEST) if the requested data object was not found.

See also: VERINFO, PREPARE_DATA.

4.2.32 VERINFO[31](S_VERINFO ver)

An embedded device sends this message in reply to a VERINFO_REQUEST message. The format of the S_VERINFO structure was given previously, in the description of the PREPARE_DATA message.

See also: VERINFO_REQUEST, PREPARE_DATA.

4.2.33 CAMTHRESH_REQUEST[32]( )

Originally intended for cameras only, hence the message name, this feature now applies to all embedded devices, ie. hubs, cameras, video switchers and relay cards.

Some aspects of the operation of these embedded devices can be altered by changing the value of certain variables, known generically as "threshold variables". The display sends the CAMTHRESH_REQUEST message to the device in order to retrieve the current values of all threshold variables, which the device returns in a CAMTHRESH message. See the description of CAMTHRESH for a further details.

4.2.34 CAMTHRESH[33](S_CAMTHRESH ct)

In embedded device sends this message in reply to a CAMTHRESH_REQUEST. A display (or other device) can also *send* this message to the device in order to set new values for the threshold variables. In the latter case the display should first retrieve the current settings from the device, change the variable it wishes to change, then send the whole structure back to the same device. The format of S_CAMTHRESH is as follows :-

```
typedef struct {
   word  dev_address;
   word  thresh_ver;
   dword val[20];
} S_CAMTHRESH;
```

The *dev_address* is merely used as a confirmation that you are talking to the correct address, it is not possible to change a devices network address by changing this field.

thresh_ver should be set to 0x0100 for all device types.

The *val* array is the important part of the structure, and contains the actual threshold settings. Not all entries in this array are used, in fact most are reserved for future expansion. The interpretation of *val* elements furthermore depends on the class of device being addressed :-

Visual Flame Detectors:

val[0] - CAMTHRESH format version number (currently 0x0100).

val[1] - Pixel intensity for image thresholding (default = 250). Be wary of changing this number, as a nonsensical value can prevent the fire detection function from working, and may even lock up the camera.

val[2] - contains several bit fields. The low byte is a camera "target sensitivity" index (0=5kW at 10m, 1=10kW, 2=15kW, 3=20kW, 4=30kW, 5=50kW). Bit 8 is set to one to designate this camera as a "flare monitor". Bits 9..31 are reserved for future use.

val[3] - contains several bit fields. Bit 0 should be set to 1 to allow the camera firmware to manage exposure control - if this bit is zero then a fixed exposure setting is used. If b1 is set then the camera's heater control mechanism is enabled. If b2 is set then the cameras video output is disabled after powerup or reset - a SELECT_VIDEO message must be sent (normally by a hub) to turn the video on at appropriate times. Set b3 to enable the "**dark video**" feature in the camera, whereby the average intensity of the video output is kept as dark as possible, consistant with retaining the ability to senses flames: this minimises the probability of encountering several types of false alarm; the video is automatically brightened when a display requests to see the live video. b4 controls the "**overlays**" feature whereby the bounding rectangles of candidate objects (blobs) is painted and superimposed onto the live video (color coding of blob rectangles is as follows: mid-gray = unclassified candidate, black = classified as non-fire, bright white = classified as fire or test torch). Bits 5..31 are reserved for future use and should not be changed.

val[4] is the **device number** for the camera, eg. setting this field to 0x234 changes the camera address to $1234. Note that you must reset the camera in order for the new address to become active. The MEL display software currently assumes that all mark II cameras have network addresses greater than $1200 - please continue this convention pending a better way to distinguish hardware revisions.

val[5] is the setpoint for the heater control feature, in degrees C.

val[6] allows you to set a limit on the maximum time which the camera can expose an individual video frame for, in coarse exposure units (video line times). Note that 310 is the true (sensor determined) cap on exposure time, longer exposure times are approximated by increasing the gain. Eg. setting a limit of 1240 means that the camera can expose the frame for 310 line times, plus an amplifier gain of 4. Gains higher than this lead to noisy images, which may increase false alarms. Overly bright images may also increase false alarms.

val[7] allows you to set the "desired average intensity" for an image. This is essentially the target image brightness for exposure control purposes.

val[8] contains an intensity (0-100%) for the IR leds.

val[9] is reserved for future use.

val[10] contains a minimum exposure time value, in fine exposure units.

val[11] contains a time in seconds to automatic alarm latch reset (0=infinite).

val[12] contains a time in seconds for automatic return to dark video mode (0 = default to 10 minutes).

val[13] to *val*[19] are reserved for future use and should not be changed.

Hub:
val[0] is the **device number** for the hub, eg. setting this field to 0x123 changes the hub network address to $6123. Note that you must reset the hub in order for the new address to become active.

val[1] contains several bit fields which configure the operation of the hub. The two lsbs (b0..b1) determine the role of the hub in the network hierarchy: 00=slave hub, 01=primary hub, 10=master hub, 11=reserved (most network devices assume that the master hub has a network address of $6001 - please continue this convention). The next two bits (b2..b3) configure the hub for dual hub operation: 00=not dualled, 01=dualled (default to standby), 10=dualled (default to active), 11=reserved. A "primary hub" is a hub whose network responsibilities lie intermediate between those of a slave hub and those of the master hub. A primary hub might act as master inside a remote equipment room (taking control action etc), but remain subordinate to a global master hub as far as data gathering is concerned. Primary hub implementations are complex and beyond the scope of this document: please contact MEL if you require further details. b4 is set if you do NOT want the hub to activate the
system VCR on an alarm. b5..b31 are reserved for future use.
val[2] to *val*[19] are reserved for future use and should not be changed.

Video switcher:
val[0] is the **device number** for the switcher, eg. setting this field to 0x123 changes the video switcher network address to $5123. Note that you must reset the switcher in order for the new address to become active.

val[1] to *val*[10] contain coefficients for the video switchers brightness and contrast compensation feature. No further information will be provided here, since at the time of writing the feature alluded to is essentially non-functional, and the video switcher design is therefore likely to be revised. For the moment you should avoid changing the value of these threshold variables.
val[11] to *val*[19] are reserved for future use and should not be changed.

Relay card:

The relay card is somewhat odd in its handling of the device number. The device number is not stored in the *val* array at all, but due to space shortage is instead stored in the *thresh_ver* field of the CAMTHRESH structure. Other than that the handling is standard, ie. setting that field to 0x123 changes the relay card network address to $7123. Note that you must reset the relay card in order for the new address to become active.

val[0] configures the "relay type" for the first four relay outputs. The least significant byte controls relay 1, the next byte controls relay 2, and so on. Bit assignments inside the four bytes are identical: b0..b6 sets the basic type of the relay, ie. 0=switch (discrete commands turn the relay on and off), 1=pulse (a command turns the relay on, but it goes off by itself after a programmed "on duration"), 2=square wave (discrete commands enable and disable the relay, but when enabled the relay outputs a square wave using programmed "on duration" and "off duration"). Other relay types reserved. Setting the most significant bit of the byte (ie. b7) inverts the meaning of "on" and "off", making the relay "normally on" - this bit can be used with switch, pulse and square wave type relays.

val[1] does the same as val[0], but applies to relays 5,6,7 and 8.

val[2] does the same as val[0], but applies to relays 9,10,11 and 12.

val[3] does the same as val[0], but applies to relays 13,14,15 and 16.

val[4] sets the "voting threshold" for relays 1 to 4, ie. the least significant byte controls relay 1, the next byte controls relay 2, and so on. Each byte contains the number of votes required to turn the relay on (or off), ie. if the voting threshold is two then two different devices have to request the relay to be energised before it is actually turned on.

val[5] does the same as val[4], but applies to relays 5,6,7 and 8.

val[6] does the same as val[4], but applies to relays 9,10,11 and 12.

val[7] does the same as val[4], but applies to relays 13,14,15 and 16.

val[8] sets the "on duration" for relays 1 to 4, ie. the least significant byte controls relay 1, the next byte controls relay 2, and so on. Each byte contains a number of milliseconds the relay should be turned on for (coded a special way so it will fit into a byte), and applies to pulse and square wave type relays. The byte is coded is a six bit value, plus a scaling factor in the two most significant bits: if the scale code is 00 then the scale factor is 1, if the scale code is 01 then the factor is 10, a scale code of 10 means a

factor of 100, and a code of 11 gives a scale factor of 1000. Thus it is possible to represent time periods from 0 to 63000 milliseconds in a single byte, even though not every number in that range can be represented.

val[9] does the same as val[8], but applies to relays 5,6,7 and 8.

val[10] does the same as val[8], but applies to relays 9,10,11 and 12.

val[11] does the same as val[8], but applies to relays 13,14,15 and 16.

val[12] sets the "off duration" for relays 1 to 4, ie. the least significant byte controls relay 1, the next byte controls relay 2, and so on. Each byte contains a number of milliseconds the relay should be turned off for (coded as per the "on duration" fields), and applies to square wave type relays only.

val[13] does the same as val[12], but applies to relays 5,6,7 and 8.

val[14] does the same as val[12], but applies to relays 9,10,11 and 12.

val[15] does the same as val[12], but applies to relays 13,14,15 and 16.

val[16] to *val*[19] are reserved for future use and should not be changed.

Motion Sensors:
val[0] - CAMTHRESH format version number (currently 0x0100).

val[1] - Pixel intensity for image thresholding (unused).

val[2] - The low byte is a camera "movement sensitivity" index (0=5 pixels, 1=10 pixels, 2=15 pixels, 3=20 pixels, 4=25 pixels, 5=30 pixels). Bits 8..31 are reserved for future use.

val[3] - contains several bit fields. Bit 0 should be set to 1 to allow the camera firmware to manage exposure control - if this bit is zero then a fixed exposure setting is used. If b1 is set then the camera's heater control mechanism is enabled. If b2 is set then the cameras video output is disabled after powerup or reset - a SELECT_VIDEO message must be sent (normally by a hub) to turn the video on at appropriate times. Bits 3..31 are reserved for future use and should not be changed.
val[4] is the **device number** for the motion sensor, eg. setting this field to 0x234 changes the camera address to $8234. Note that you must reset the camera in order for the new address to become active.

val[5] is the setpoint for the heater control feature, in degrees C.

val[6] is reserved for future use and should not be changed.

val[7] allows you to set the "desired average intensity" for an image. This is essentially the target image brightness for exposure control purposes. Note that for motion sensors (and unlike flame detectors) there is no incentive to keep the image dark.

val[8] contains an intensity (0-100%) for the IR leds.

val[9] is reserved for future use.

val[10] is reserved for future use and should not be changed.

val[11] contains a time in seconds to automatic alarm latch reset (0=infinite).

val[12] to *val*[19] are reserved for future use and should not be changed.

4.2.35 DSP_DEBUG[34](word cmd)

4.2.36 DSP_CONTEXT[35](dword data[])

Do not use these messages. They are used within MEL during development of new camera firmware revisions to run sessions with a debugger in conjunction with the DSP chip inside the camera. Non-debug versions of the camera will not properly handle these messages, and in any case the message data is highly firmware and hardware version specific.

4.2.37 FLARE_ALERT[36]( )

This message is no longer used.

4.2.38 FLARE_ALLCLEAR[37]( )

This message is no longer used.

4.2.39 INIT_REQUEST[38]( )

When the network is powered up the master hub automatically begins an initialisation and device discovery phase (see chapter 5 for a description of the latter). Slave hubs are different only in that they wait until a parent hub, ultimately the master hub, sends them an INIT_REQUEST message they do the same. The port that the INIT_REQUEST message is received on is labelled as the "upstream port", the slave then performs discovery on all remaining, ie. downstream, ports. The slave hub responds with ACK(INIT_REQUEST) and begins initialising itself. During discovery the parent hub continues to send infrequent STATUS_REQUEST messages to the slave hub, the slave (as always) sends a STATUS message in reply. The parent waits until the *active* bit is set in this reply (indicating that the slave has completed discovery), and then uses a HUB_DEVLIST_REQUEST message to query the slave hub for the list of the downstream devices it discovered. Non-hub devices should not send this message.

4.2.40 HUB_DEVLIST_REQUEST[39](word iStart)

A device can send this message to a hub to obtain a list of the devices which the hub detected during discovery. This is *not* the same as the master device list. The latter is a manually constructed and maintained list of those devices which *should* be connected (and in any case that list is only available on the master hub), whereas the "discovered device list" (henceforth simply "the discovery list") is the list of those devices which are *actually* connected, or are more properly those devices which responded to the polling tests during discovery.

This message and its reply are mainly intended to be used between hubs (see the description of INIT_REQUEST), however a display can also send this message, and then use the difference between the master and discovery device lists to determine whether all expected devices were detected. The hub replies to this message with a HUB_DEVLIST message.

See the HUB_DEVLIST description for the meaning of the *iStart* parameter.

Note that the "discovery list" obtained from a hub includes all devices which are downstream from that hub, it does *not* include only those devices which are directly connected to the hub in question. So, to obtain the discovery list for the entire network you simply send a HUB_DEVLIST_REQUEST message to the master hub (after ensuring, of course, that the master hub has completed its discovery phase).

4.2.41 HUB_DEVLIST[40](S_HUB_DEVLIST shd)

This message is sent by a hub in reply to a HUB_DEVLIST_REQUEST message. The format of S_HUB_DEVLIST is as follows :-

```
typedef struct {
   word iStart; /* index of first device in page */
   word iCnt;   /* total number of devices */
   S_HUB_DLENTRY list[169];
} S_HUB_DEVLIST;
typedef struct {
   word addr;     /* device address        */
   word aParent;  /* address of parent hub */
   word bFlags;   /* initial device flags  */
} S_HUB_DLENTRY;
```

Note that the discovery list is likely to be too large to fit into a single packet. So, the sending hub breaks the list into chunks, and it is up to the requesting device to request all the chunks. The requesting device starts by sending HUB_DEVLIST_REQUEST(0), which requests a list of all devices, starting from list index 0. The hub responds with a HUB_DEVLIST message, in which the S_HUB_DEVLIST.iStart field should match the requested start index, and the S_HUB_DEVLIST.iCnt field gives the total number of device list entries available. If iCnt is less than or equal to 169 then the list did in fact fit into one packet and the conversation is complete. Otherwise, the requesting device sends a second HUB_DEVLIST_REQUEST, though this time the iStart parameter is 169 - and so until the last chunk of the list is retrieved.

4.2.42 HUB_CHANGES_REQUEST[41](word iStart)

4.2.43 HUB_CHANGES[42](S_HUB_DEVLIST shd)

These messages are intended to be used between hubs, to propagate device status changes up through the network hierarchy, and should not be used by non-hub devices. However, for the curious, these messages work in exactly the same manner, and use exactly the same data formats as the HUB_DEVLIST_REQUEST and HUB_DEVLIST messages described earlier. The only difference is that unlike the discovery list, the HUB_CHANGES list consists solely of those devices whose device flags have changed since the last time they were included in a HUB_CHANGES or HUB_DEVLIST list.

4.2.44 FIND_DEVICE[43](word dev_addr)

This message is intended for use between hubs, and should not be used by non-hub devices. When the master hub receives a CONNECT(dev_addr) message asking it to activate a previously unknown device *dev_addr*, it first checks whether device dev_addr is directly connected to one of its own ports. If not, it sends a FIND_DEVICE(dev_addr) message to any slave hubs, asking them to check *their* ports. The message propagates downstream until the device is hopefully discovered.

4.2.45 ERRORSTATS_REQUEST[44]( )

Every hub maintains error rate and other diagnostic statistics relating to every network port, and also keeps seperate stats for every individual device which is connected to one of those ports. This message allows an interested party, ie. a display, to retrieve this information for subsequent presentation. The hub replies with an ERRORSTATS message. See the description of the latter for further information.

4.2.46 ERRORSTATS[45](S_HUB_ERRSTAT errstat)

Warning: the format of this structure changed in version 2.5 of the protocol, mainly to accomodate the increased number of ports in the mark II hub. Note in particular the addition of an *nPorts* field in S_HUB_ERRSTAT, and that number of elements in the *portstat* field is now controlled by the value of *nPorts*. The S_CHILDSTAT structure also changed size, to ease some alignment issues.

Every hub maintains error rate and other diagnostic statistics relating to every hub port, plus every individual device which is connected to one of its ports. This information can be requested using an ERRORSTATS_REQUEST message, and is returned to the requestor in an ERRORSTATS reply. The format of S_HUB_ERRSTAT is as follows :-

```
typedef struct { /* Hub error statistics */
   dword sig;
   word  nPorts;
   word  nChildren;
   S_PORTSTATS portstat[nPorts];
   S_CHILDSTAT childstat[nChildren];
} S_HUB_ERRSTAT, *PERRSTAT;
typedef struct {
   byte  nSent,nErrors; /* packet error rate */
   word  porttest;      /* top 11 bits==impedence */
                        /* b0=24V fuse OK. */
                        /* b1=Loopback test OK. */
} S_PORTSTATS;
typedef struct {
   word  DevAddr;
   byte  iPort;
   byte  nSent;
   byte  nErrors;
   byte  spare1;
   byte  spare2;
   byte  spare3;
} S_CHILDSTAT;
```

Contrary to the precedent set by most other network messages, a hub only maintains diagnostic information for those devices which are directly attached to it, *not* for all downstream devices. So, you cannot get diagnostics for the entire network by querying the master hub, you must query each individual hub, ideally displaying the results in a "tree view" hierarchical listing.

The *portstat[]* array gives diagnostic information for each of the hub ports, the *childstat[]* array gives largely the same information, but in relation to individual child devices. In this way an engineer can see, should he encounter a high error rate, whether the problem seems to be associated with all devices on a given port (implying that the port itself or the cable is faulty), or only with a single device - which would imply that it is the device which is at fault. *iPort* is the port (numbered from zero) to which a device is attached. The *nSent* field in either array gives the total number of packets sent to that device or on that port. The *nErrors* field is the number of transmissions which did not elicit a reply. Thus, the error rate can be expressed as a percentage using the formula: *error_rate = (nErrors*100)/nSent*. Note that *nSent* and *nErrors* are single byte values - when *nSent* is about to overflow the hub divides both it and *nErrors* by two, which keeps the ratio between the two values correct whilst also preventing the overflow.

4.2.47 PINGTEST_REQUEST[46](S_PTINFO ptinf)

This message is sent by device A (typically a display) in order to request that device B (typically a hub) should perform a series of comms reliability tests with a third device, C (typically a camera or other passive device). In theory the PINGTEST_REQUEST message can be sent to any network device, in practice only a hub understands the request. The result is that device B conducts a number of PING_REQUEST/PING trials on its comms link to device C, collecting error statistics as it goes, and finishes up by sending the processed results back to device A in a PINGTEST_RESULT message.

The ptinf structure gives details of the test which device A want device B to perform. The format of S_PTINFO is as follows :-

```
typedef struct {
   word DevAddr;
   byte nPort;
   byte fTimeout;
   word nPings;
   word lenPing;
} S_PTINFO;
```

DevAddr is the network address of device C, ie. the device to be tested. *nPort* is the hub port (0..7) to use for the test (note that the testing hub B is not assumed to already know the correct port to use, since the reason for the doing the comms test in the first place will often be because of a failure by the hub to detect device C during discovery); the *nPort* value is ignored if the testing device (B) is not a hub, ie. if it only has one port. *fTimeout* is the length of the timeout value which the testing device should allow for each PING reply message - if *fTimeout* is zero a short timeout is used, otherwise a long

timeout is used. *nPings* is the actual number of ping tests which should be conducted, and should be in the range 0 to 128 (0 is treated the same as 1). *lenPing* is the "ping length multiplier" - see the PING_REQUEST message for a description of this value.

- Note that the requested device (ie. the hub, device B) first sends an ACK(PINGTEST_REQUEST), then conducts the test, and finally sends the PINGTEST_RESULT message.

Do not confuse the PINGTEST_REQUEST and PING_REQUEST messages. The former asks a device to conduct a series of ping tests and send back statistics, the latter *is* a ping test.

4.2.48 PINGTEST_RESULT[47](S_PTRSLT ptrslt)

This message is used to return the data resulting from an earlier PINGTEST_REQUEST. The message should be acknowledged with an ACK(PINGTEST_RESULT). The format of *ptrslt* is as follows :-

```
typedef struct {
   S_PTINFO ptInf;
   word nOk;
   word nNoReplies;
   word nCrcErrors;
   word nUnderflows;
   word nOverflows;
   word nRestarts;
   word cSpurious;
   word cUartErrs;
   word spare[7];
   word nHistItems;
   word HistItems[2];
} S_PTRSLT;
```

ptInf is a copy of the structure passed in the original PINGTEST_REQUEST message. *nOk* is the number of PING messages which were received with no errors. *nNoReplies* is the number if times device C failed to reply at all. *nCrcErrors* is the number of times a reply came, but contained a CRC error. *nUnderflows* is the number of times a reply underflowed, ie. the reply packet was truncated. *nOverflows* is the number of times the reply overflowed the expected reply size. *cSpurious* is a count of the total number (over all the tests) of spurious or garbage characters, ie. those which did not seem to be part of a reply packet (this could also indicate that STX is being corrupted). *cUartErrs* is the number of character framing or other errors detected by the UART.

The pingtest histogram consists of *nHistItems* PAIRS of word values, the first word giving a position (length), the second word containing a count of the number of times the first corrupted byte in a packet occurred at that position. Since the ping test is carried out a maximum of 128 times, that means that there can only be a maximum of 128 entries in the histogram array, ie. the maximum reply packet length is 536 bytes - not counting header, framing or DLE-escaping bytes.

4.2.49 PING_REQUEST[48](word lenPing)

4.2.50 PING[49](byte testdata[])

PING_REQUEST is used to request that the receiver send back a single PING message in reply. *lenPing* is a requested reply length, in multiples of the 16-byte signature string, and should be in the range 0 to 64 (0 is treated as being the same as 1). The body the PING reply should be the signature string "01_PINGMESSAGE_3" repeated *lenPing* times; also the flags byte of the PING reply must be 0x05, regardless of the actual state of the device flags on the pinged device.

4.2.51 CONTROLLER_RESET[50]( )

The master hub broadcasts this message to all devices when it is about to reset the network. Displays and other devices should cancel any ongoing conversations, and not attempt any further network conversations until they see a CONTROLLER_ACTIVE or other hub message in which the *active* bit is set in the frame flags. The display should however respond to a STATUS_REQUEST specifically addressed to it. This message is also important to the discovery protocol (see chapter 5), since it announces to all devices that they need to consider themselves "undiscovered" again.

4.2.52 RU_THERE[51]()

4.2.53 UR_FOUND[52]()

These messages form part of the network discovery mechanism. The RU_THERE message is used by a hub to probe for the presence of undiscovered devices, while the UR_FOUND message informs a device that it has been discovered. See chapter 5 for a more detailed description.

The UR_FOUND message should be acknowledged with ACK(UR_FOUND, *w*), where the signature word *w* is a non-zero, unsigned 16-bit integer randomly selected by the device at or before the beginning of the discovery process. A device should not change its signature word once it has selected one, as the signature is used rather like a backup device id in the case where a parent hub detects an address conflict during discovery.

4.2.54 SET_RELAY[53](word nRelay, word addrVoter)

The SET_RELAY message is used to turn a relay on or off in any embedded device which includes a physical or logical relay output. In particular this message is used to control an MEL relay card, but is also used to unlatch an alarm output in a camera.

In a relay card the least significant byte of *nRelay* argument selects a relay number (0..15), while the most significant byte controls what is done with the relay: 0x00 means turn the relay off, 0x80 means turn the relay on, and 0xFF designates the selected relay as a watchdog relay (more on this later).

Other values for the msbyte are reserved. Remember that the precise meaning of on and off depends on whether the relay is configured to be normally on, or normally off.

Relay cards are configured to require a certain number of valid votes before the relay actually turns on or off. The *addrVoter* parameter tells the relay card which device is voting to turn the relay on (or off); note that *addrVoter* is not necessarily contain the same network address as the device which sent the SET_RELAY message. For example, hubs send SET_RELAY messages in response to camera alarms, identifying the camera as the voter. The relay card records which devices have voted, so that a single device cannot satisfy the voting requirements by casting multiple votes.

It is possible to nominate one relay on a relay card as a "watchdog" relay. This is most often used to implement a "panel fault" output in an equipment room. The basic idea is that once a watchdog relay output has been nominated the relay card *will expect to be regularly polled by the hub which sent the SET_RELAY message*. If that does not happen then the relay card will turn the watchdog relay on (or off, if the selected relay output is configured as normally-on). The *addrVoter* argument is ignored when the SET_RELAY message is used to designate a watchdog relay.

Mark II cameras also process the SET_RELAY message. Currently (firmware v1.18) cameras ignore both arguments to this message and instead unconditionally reset their alarm latches (and associated alarm relay if required). However, you should pass appropriate values in the *nRelay* and *addrVoter* fields nonetheless, for compatibility with future camera firmware releases. "Appropriate values" would mean passing a relay number of 0 for the alarm relay (1 for the fault relay), with 0x80 in the msbyte of nRelay to turn the relay on, 0x00 to turn it off. Also supply an *addrVoter*, which in this case *will* normally be the same network address as the sender of the SET_RELAY message.

4.2.55 WARNING_RESET[54]()

4.2.56 WARNING_REQUEST[55]()

Section 3.6 mentioned that a new mechanism had been provided whereby devices can signal that some internal self-diagnostic feature has turned up a problem. They signal this fact by turning on bit 3 in the device flags. The hub picks up this change and sends DISPLAY_ALERT messages to available displays. The device status included in the alert includes a new two bit field which indicates whether the device is currently signalling a warning, and whether that warning has been acknowledged. However, it is not possible for a display to diagnose the problem using only the device status: for that information it must communicate directly with the device which originally raised the signal.

To diagnose a warning condition the display can send a WARNING_REQUEST to the device. The device replies with ACK(WARNING_REQUEST, warn_code), where *warn_code* is a numeric code

which indicates the problem. The display can convert this code into a friendly text message and inform the human operator. Currently assigned warning codes are as follows :-

1 (from a dual hub). Indicates that this is a dual hub arrangement, and that the default active member of the duo has failed. By convention the default-active hub resides in the left hand slot of the dual hub baseboard. The default-standby component will have taken over automatically, if it was not already active.

2 (also from a dual hub). Indicates that this is a dual hub arrangement, and that the default standby component of the duo has failed. By convention the default-standby hub resides in the right hand slot of the dual hub baseboard. The default-active component of the dual hub has already taken over, if it was not already active.

3 (from any device). Indicates that a firmware update recently sent to this device has not been written to flash, because the reconstructed file failed a file CRC check (see also BEGIN_TRANSFER). You should try sending the update to this device again.

0x100 through 0x1FF (from any device). Indicates that the device has carried out an impedence test on its network port (code - 0x100), and has detected a sudden jump in impedence. This could indicate a growing cable short, or a core working itself loose etc.

0x200 through 0x200 (from any device). Indicates that the device has encountered an unacceptably high packet error rate while attempting to use its network port (code - 0x200). Where possible, the device in question will have switched to using another port.

Remaining code values are reserved for future use.

When the human operator has seen the diagnostic information the display may send the signalling device a WARNING_RESET message, which causing it to turn off its warning flag, *unless* the device has a another diagnostic problem to report. Ie. devices may maintain a queue of diagnostic results, and signal each in turn as the previous signal is reset.

5 The Discovery Mechanism

5.1 Overview

The discovery mechanism is the means by which the MEL hubs detect which devices are immediately connected to their network ports. It is therefore mandatory that all Comm485 compatible devices implement the slave side of the discovery mechanism, if they expect to operate in a network which includes MEL hubs.

All current and envisaged MEL embedded designs, ie. hubs, cameras, video switchers and relay cards, are implemented in two parts: a cpu board (which itself may include more than one part), plus a "baseboard" or "termination board". The latter contains field termination of some sort, normally screw terminals, and in the past has also included an n-way dipswitch or solder bridge for setting the device number, though later devices have included a serial EEPROM for this purpose. The logic part of the device concatenates the device number which is programmed on the baseboard with its own device class to form a full network address.

Hubs must, during their "discovery" phase, search for connected devices, which they originally did by (potentially) sampling every possible device address within the total address space of the network. As you might imagine, it is possible for the discovery process to take a very long time to complete, since a sixteen bit addressing scheme allows anything up to 65536 connected devices: if you allow three tries per possible address and assume a 50ms timeout on each poll then that adds up to 3 hours (approximately) to sample the entire address space. In reality the problem was less acute because the hub did not in fact sample the entire address space: not all device classes were assigned (4096*3 polls are saved for every unused class), and the hub also placed arbitrary restrictions on the number of devices allowed in particular classes, eg. 256 cameras, 32 hubs, 8 displays and so on, and also on the address ranges that these devices could use, so that discovery actually took only about 40 seconds or so, that time being dominated by the number of allowed cameras.

We needed a new discovery mechanism which would work faster *and* remove these arbitrary limitations. The simplest solution would have been to limit the range of addresses allowed on a single hub port, or introduce a new hierarchical addressing scheme which would have required all devices on the network to be aware of the position in the hierarchy of every other device on the network. Such schemes would obviously limit the flexibility of the topology and introduce additional housekeeping headaches - it is doubtful that such a cumbersome maintenance procedures would have met with approval.

In an effort to maintain network flexibility, while removing the arbitrary address range limitions mentioned above and incidentally speeding up the discovery procedure, the following new discovery mechanism was developed.

5.2 Assumptions

First, we assume that up to 32 devices (including the hub itself) can be connected to a single hub port. This assumption forms the basis for certain probability and timeout calculations, however the algorithm described in this chapter does not actually limit the number of potential devices to 32. We furthermore assume that every device present has a unique full network address assigned either by MEL or by the client.

We assume that every network device maintains a flag variable called *BeenDiscovered*. If a device has more than one port and is connected to different parents on the extra ports then there will need to be a separate *BeenDiscovered* variable for every parent. The discussion below assumes a single parent, but is easily extended to handle several.

All of the message passing described below takes place during discovery, but in any case it is assumed that every one of the messages described below is transmitted by the hub inside a packet whose flags field contains an *active* bit which is set to zero. Therefore, following the rules given earlier in this document (see sections 2.2 and 2.3.3), none of the messages described below should be routed downstream by a receiving hub. An immediate downstream hub should respond to messages it receives, provided the addressing is correct, but should not route messages to devices further downstream, even if the message is a broadcast, as some of the discovery phase messages are.

The description below gives absolute numbers for certain timeouts, and also gives predicted times to completion for certain tasks. These numbers are derived from the assumption that the current sixteen bit addressing scheme is used (which affects the size of packets), and that the data rate is 57600 bps. The numbers will need to be adjusted if these parameters change.

The description below describes the algorithm as it is used by a hub on "the downstream port". In fact the hub has several downstream ports and the algorithm is carried out separately on each one, most likely in parallel.

5.3 Outline of Algorithm

The (extremely simple) algorithm proceeds as follows :-

1. The hub zeroes its discovery list, then broadcasts a CONTROLLER_RESET message several times. Any device which receives CONTROLLER_RESET sets their *BeenDiscovered* flag to FALSE. The hub then enters its "discovery loop" as described below.

2. At the start of each discovery loop iteration the hub **broadcasts** an RU_THERE message on the downstream port. If a downstream neighbour receives this message, and if the *BeenDiscovered* flag inside that device is false, then it should wait for ***T***+RND(32)*****N*** milliseconds before sending a STATUS message back to the hub on the same port. Note that the RU_THERE message is a broadcast, but the STATUS reply header should contain unambiguous destination and sender address fields. ***T*** is the normal turnaround delay (5ms), RND(32) means a random integer between 0 and 31 inclusive, ***N*** is the length of a unit time slot, which should be roughly twice the time required to transmit a STATUS packet, given the size of the packet and the current network baud rate. Assuming 16bit address fields and a baud rate of 57600, a suitable value for N would be 5ms.

3. The hub listens for a full ***T***+32*****N*** ms (ie. 165ms at 57600bps) for replies, noting all uncorrupted replies which it receives. Go to step 4 if replies were received, else go to step 5.

4. This step is reached if one or more uncorrupted replies are received; each reply contains an address header which provides the address of a new device. For each reply the hub appends an entry for the source device to its discovery list (unless the device is already listed), then sends the device an unambiguously addressed UR_FOUND message. A device which receives a correctly (and specifically) addressed UR_FOUND message should set its own *BeenDiscovered* flag to true, then send back an ACK(UR_FOUND,*w*) message to the hub. At this point the device is deemed to have been discovered and ignores any further RU_THERE messages which it sees, unless it first sees a new CONTROLLER_RESET. The hub returns to step 2 once it has processed all replies in the manner described.

The *w* field in the child device's acknowledgement of UR_FOUND is called the *device signature*, and helps the parent hub to recognise address conflicts. The signature is just a non-zero 16-bit unsigned integer, selected at random by the child device when it first receives the CONTROLLER_RESET message. From the point of view of the parent hub: if two devices on different ports have the same network address then the signature word helps it determine whether this implies a dual redundant path to a single device, or whether it indicates an address conflict.

5. This step is reached if no (uncorrupted) reply is received for an RU_THERE message. If this is the *third consecutive* failure to receive a reply then go to step 6, else return to step 2.

6. When this step is reached, discovery is completed on the given port.

5.4 Reliability & Timing

We need to consider how likely it is that a device will fail to be discovered when the above technique is used. Potential causes of failure which should be considered are :-

- The child device fails to receive an uncorrupted RU_THERE message - this should not be a problem, since unless its comms are completely defective it should see the message in a future iteration. If its comms *are* defective then no discovery algorithm will work.

- Somewhat unlikely, but the device may fail to receive the UR_FOUND message. Again, this is not a problem since the device will not consider itself to have been discovered and hence will have further chances to complete discovery. It is to cope with this scenario that the hub must check, before adding a device to the discovery list, whether or not that device is already listed.

- Two or more devices may select the same RND(n) value, and hence reply at the same time, mutually corrupting each others replies so that the hub receives neither. However, not only is this not a problem, in fact it is an intended feature of the algorithm. We assume that in any pass of the algorithm a certain number of packets will be corrupted, and the remainder will get through. The fraction that get through are acknowledged and the source devices told to be quiet, which reduces the probability of conflicts in future passes.

Assuming all of the above, how long should discovery take? Assuming that the maximum number of undiscovered devices is present (31), and that the timeout values based on 57600 bps are used, then tests have shown that almost always, all 31 devices are discovered in only three passes of the discovery loop. We make a further three passes just to be sure, so the whole process takes 165*6 = 990ms, ie. less than a second. This is forty times faster than the old discovery method, yet has none of the arbitrary address range or "numbers per device class" limitations of the latter.

5.5 The RND(n) Function

The algorithm described requires each slave device, no matter how simple, to implement a reasonably good random integer generator function. MEL has found the following implementation to be suitable - note that this C implementation assumes 32bit ints.

You must call the *Randomize*() function during a device reset in order to seed the random number generator - we suggest that each device use a derivation of its network address as the seed. While other potential seeds might be available (eg. a timer counter value), one must bear in mind that discovery can be expected to take place within seconds of powerup, and device clocks may not have drifted sufficiently to guarantee a different seed on each device.

```
#define BUFFSIZE 55

static dword head;
static dword tail;
static dword History[BUFFSIZE];

/*.............................................................*/

void Randomize(dword x)
{
   dword i;
   for (i=0; i<BUFFSIZE; i++) {
      x = x*314159621+17;
      History[i] = x;
   }
   head = BUFFSIZE-1;
   tail = 23;
}

/*.............................................................*/

dword RND(dword range)
{
   dword r = (History[tail++] + History[head++]);
   if (head==BUFFSIZE) head = 0;
   if (tail==BUFFSIZE) tail = 0;
   History[head] = r;
   return LongUMul(r, range);
}

/*.............................................................*/
```

In function *RND()* the random number *r* is between 0 and (2**32)-1, which we need to scale into the interval 0..range-1. We do NOT just calculate (r mod range) since that returns the least significant digits of r, which are also the least random. Instead we do a proper scaling, ie. (r*range + (2**31)) / (2**32) : the divide corresponds to simply taking the high 32bits of the 64bit result produced by the 32x32bit multiplication. We assume the availability of a assembler function LongUMul() which can carry out this calculation.

5.6 Summary

The new discovery mechanism is much faster than the old one even in the worst case, and furthermore does not impose arbitrary limits on the address range of devices connected to the port, which the old method did. The new method could also be extended to handle future >16bits addressing schemes without significantly affecting performance. However, there is a remote possibility that the new method will fail to detect connected devices if they succeed in blocking each others replies for three passes. Finally, the old method preferentially detects slave hubs (ie. it searches for hubs first), in order to have them begin their own discovery in parallel with that of the parent hub. The new algorithm does not lend itself to preferential discovery, however this fact is not truly significant: even if each level of the hub hierarchy performs discovery separately, the summed discover time of 3-4 seconds is still only a fraction of the time required by the old method.

6 APPENDIX A - CRC16 Calculation

This section provides sample C code which implements the updcrc() function as used in section 3.5. The sample shows a table-driven CRC calculation, and hence it is first necessary to generate the auxiliary table, as in the first function :-

```
/*.............................................*/
#define POLY   0x1021
static word CRCtable[256];
static void near
MakeCRCTable(void)
{
   unsigned i,j,CRC;
   for (i=0; i<256; i++) {
      CRC = (i<<8);
      for (j=0; j<8; j++) {
         if (CRC & 0x8000) {
            CRC = ((CRC<<1) ^ POLY);
         } else {
            CRC <<= 1;
         }
      }
      CRCtable[i] = (word)(CRC & 0xFFFF);
   }
}
/*.............................................*/
```

The *MakeCRCTable*() function is called once, during software initialization. Alternatively the table itself may be stored permanently in ROM or NVR. Once the table is available , the *updcrc*() function is simply:-

```
/*.............................................*/

static word near
updcrc(word CRC, word c)
{
   return (CRCtable[CRC>>8] ^ (CRC<<8) ^ c);
}

/*.............................................*/
```

7 APPENDIX B - RLE Encoding of Bitmaps and Bitmasks

7.1 Overview

The MEL Flame Detection camera uses bitmaps extensively (the word bitmap is often abused: it specifically means a two-colour image wherein every pixel is mapped to a single bit in the represention, ie. a "bit map"). Bitmaps can be efficiently compressed and transported across our network, providing a useful alternative to the analogue "live video" feed; transporting image data over a digital connection enables telemetry data to be included as well. Also, the camera accepts bitmaps from display PCs which it uses as "masks" (hence "bitmasks") in order to exclude areas of the field of view from flame detection. Bitmaps are important to the MEL flame detection system, and so, to improve the efficiency with which bitmaps can be transported and stored, the MEL system works extensively with RLE (Run Length Encoded) bitmaps.

7.2 Problem and Solution

The mark II camera and firmware uses a bitmap frame size of 352x288 pixels, which is significantly larger than the 256x256 bitmaps used by the v1.00 camera. The most significant consequence of this is that x and y pixel coordinates no longer fit into bytes, and since the RLE format documented in previous protocol versions assumed that they did, the RLE format has had to change. In this revision we have endevoured to make sure that the new format does not embed similar hardware assumptions.

Absolute coordinates could now fit into 9-bit values, but this is an inconvenient size to work with - we would prefer if values fitted neatly into byte boundaries. However, the next byte boundary would be 16-bits, which would obviously be wasteful.

Instead, we abandon the idea of coding runs using absolute coordinates. Instead we code them as run lengths or "skip distances", ie. deltas rather than absolute values; using deltas also avoids building assumptions about frame size into the format. To further improve frame size independence we now include an explicit header in front of the RLE-data which gives the actual frame size used.

The header has the following structure :-

```
/*.............................................*/

typedef struct {
   word  bmWidth;    /* bitmap width, in pixels   */
   word  bmHeight;   /* bitmap height, in pixels  */
   dword nRuns;      /* number of encoded runs    */
} RLE_HEADER;

typedef struct {
   RLE_HEADER hdr;
   byte data[];      /* variable length rle data  */
} RLE_DATA;

/*.............................................*/
```

All fields in this structure are stored least significant byte first, ie. "intel byte order".

Note that while the new RLE format is frame size independant, consumers of this format are not! For example, if you supply a bitmask to a mark II camera then the RLE-coded mask *must* unpack into a 352x288 bitmap. The point of building frame size independence into the exchange format is to enable *Comm485* to handle past and future devices without further changes to the protocol specification.

Immediately following RLE_HEADER is the RLE data itself. As in the past the data is coded as a number of "runs", with the exact number of runs being provided by the "nRuns" header field.

A decoder would start by creating a bitmap of the appropriate size, filled with black pixels. It would also start by initialising a "current position" variable, which is initially set to zero. This "position" is a direct pixel index into the bitmap, assuming a standard raster ordering (ie. left to right, top to bottom). The decoder would then start unpacking the run data, unless of course nRuns is zero.

Each run is coded in two fields: a "skip distance" which gives the number of black pixels to skip past (starting from the current position), and a "run length", which is the number of white pixels to set starting from the target position. Both fields are coded in a variable number of bytes, and each uses a slightly different coding format because of the different expectations as to their typical magnitude.

To unpack the "skip distance" you start by reading the first byte. If this byte value is less then 255 then no further work is required - proceed to unpack the run length. If the first byte was 255 then you read the next two bytes and form them into an unsigned sixteen bit number (intel ordering again). If this number is less than 65535 then you have the skip distance. If the number equals 65535 then you need to read a further two bytes, form these into another 16-bit number which you add to the first number (ie. to the 65535). You keep doing this until the last sixteen bit number read is less then 65535, and the

accumulated sum of all the numbers equals the skip distance. Once the skip distance is decoded you advance the "current position" by the appropriate amount. The following is pseudocode for the algorithm just described :-

```
/*.............................................*/

/* decoding the "skip distance" */
skip = get_next_byte(input);
if (skip == 255) {
   delta = get_next_byte(input);
   delta = delta + ( get_next_byte(input) << 8 ) ;
   skip  = delta;
   while (delta == 65535) {
      delta = get_next_byte(input);
      delta = delta + ( get_next_byte(input) << 8 ) ;
      skip  = skip + delta;
   }
}
current_position = current_position + skip;

/*.............................................*/
```

Decoding the run length is quite similar, but the expectation is that run lengths will rarely require more than one byte. So, you read the first byte. If that byte is less than 255 then it equals the run length. If not then you read the next byte and add its value to the run length. You keep doing this until the last byte read is less than 255. Here is the pseudocode :-

```
/*.............................................*/

/* decoding the "run length" */
delta = get_next_byte(input);
run_length = delta;
while (delta == 255) {
   delta = get_next_byte(input);
   run_length = run_length + delta ;
}

/*.............................................*/
```

Once the run length is decoded you can then use it to update the bitmap :-

```
/*.............................................*/

i = current_position;
current_position = current_position + run_length;
while (i < current_position) {
   Set_Pixel(i);
   i++;
}

/*.............................................*/
```

Notice that the *current_position* variable is advanced by the value of the run length as well as by the skip distance.

A final note: be aware that the potential for encountering skip distances or run lengths of zero is a deliberate feature. You should never assume that *nRuns*, *skip_distance* or *run_length* is always greater than zero. They are however always zero or positive (unsigned) integers.

8 APPENDIX C - Encoding of VideoRoute Fields

This appendix describes how the *VideoRoute* field from a master devlist entry is interpreted. A VideoRoute is a 32bit value, a bit vector, and is present for every network device except video switchers (those need four VideoRoute fields, which are therefore stored outside the devlist entry). Each VideoRoute field is intended to map a video output port on the selected device to a video input port on another. When a hub receives a SELECT_VIDEO command requesting video from a particular camera the hub attempts to "walk the path" from that camera, through intermediate video switchers, to the display (or VCR). If successful in finding a path the hub then sends appropriate switching commands to the video switchers which were found on the path.

Devices which don't have a video output have their VideoRoute field set to zero. Otherwise, the assignment of bits in the VideoRoute field is as follows :-

b31..b30 : This 2 bit field is always 11 (indicating new format VideoRoute field), unless the entire VideoRoute dword is zero.

b29..b27 : Connection type code :-

- 000 Video out is not connected.
- 001 Video out is connected to a "loop-in" input on an MEL mkI 16-input video switcher. This option will generally only be used to chain two video switchers together; the input number on the second switcher is expected to be the same as the output number on the first.
- 010 Video out is connected to a normal input on a video switcher. LOWORD(VideoRoute) gives the network address of the video switcher, b16..b21 gives the input number on that switcher - the mkI MEL switcher allows input numbers in the range 1-16, while the mkII allows inputs in the range 1-20 (the mkII has no loop-ins).
- 011 Video out is connected to a networked display device. LOWORD(VideoRoute) gives the address of the display.VideoRoutes using this code are generally video switcher outputs.
- 100 Video out is connected to a non-network device, LOWORD(VideoRoute) identifies the type of non-network device. Currently 0001 is the only such code defined, and means that the device is a VCR.
- 101 Old (non-MEL) video switcher bank.
- 110/111 Reserved for future use.

b26..b24 : Reserved for future use (set to zero).

b23 : Borrowed by the master hub for non-volatile storage of the INHIBIT flag for the device.

b22 : Reserved for future use (set to zero).

b21..b16 : Video switcher input number.

b15..b00 : Network address of "other" device.

VideoRoute should be 0 (meaning not applicable) for all devices other than cameras and video switchers, as no other devices currently have video outputs.

As briefly mentioned above, video switchers have four video outputs, so there is no room in the standard devlist entry to store complete video output routing information for these devices. Enlarging the devlist structure for all devices would of course be wasteful. Instead, the VideoRoute field in the master devlist entry for a video switcher is actually a string reference (like the *sref_DevName* and *sref_MimicName* fields), and the string it references is a coded version of the four required VideoRoute fields. In order to store VideoRoute (ie. binary) data in the string table it was first necessary to ensure that no byte in the dword was a NUL, and to do that we set the msb of each byte to one. However, that meant that we have to store the original msbs separately, in a byte which precedes the four coded bytes of the dword. So, coding four VideoRoute fields requires 4*5 bytes, plus we add a terminating NUL on the end of the string. The *GetSwitcherOutputRoute()* function below shows how to unpack the coded switcher VideoRoute fields, returning a standard VideoRoute coded as above :-

```
/*................................................*/
static dword
GetSwitcherOutputRoute(word addrSwitcher, word nOutput)
/* Get VideoRoute for selected output on switcher */
{
   dword i,v;
   /* find the video switcher in the master devlist */
   for (i=0; i<nDevices; i++) {
      if (mdevlist[i].DevAddr == addrSwitcher) break;
   }
   if (i>=nDevices) v = 0; /* switcher not found */
   else {
      char *psz = pStringTable
                + (mdevlist[i].VideoRoute & 0xFFFF)
                +  nOutput*5);
      dword b, msbs = (((dword)(*psz++)) << 7);
      for (i=v=0; i<32; i+=8) {
         b = ((*psz++) & 0x7F) | (msbs & 0x80);
         v |= (b<<i);
         msbs >>= 1;
      }
   }
   return v;  /* return normal VideoRoute */
}
/*................................................*/
```