Ethernet Monitoring and Documenting System

This manual describes the architecture and current implementation of the EMU system

June 1995

Preface

preface

The manual is divided into two broad categories:

- Architectural overview
- Current implementation

Contents

Preface v

preface vii

Chapter 1 Introduction

1.0.1 Main Features 1–2

Chapter 2 System Overview

2.1 Core System 2–1
2.1.1 Listener 2–1
2.1.2 PSR 2–1
2.1.3 PSR interface 2–2
2.1.4 Network Views 2–2
2.1.5 Relater 2–2
2.1.6 Identity 2–3
2.1.7 EMU network 2–3
2.1.8 Database Overview 2–4
2.1.9 Record Structure 2–4
2.1.10 Record Header 2–4
2.1.11 Database Access 2–5
2.2 Counter Processing System 2–5

Chapter 3 Implementation

3.1 Introduction 3–1
3.2 Listener 3–1
3.2.1 Description 3–1
3.2.2 Processing abstract 3–1

3.2.3 Ethernet buffers . . . 3–2
3.2.4 Protocol type formats . . . 3–3
3.3 PSR . . . 3–3
3.3.1 Description . . . 3–3
3.3.2 Control Record . . . 3–4
3.3.2.1 PSR Records . . . 3–4
3.3.3 General processing . . . 3–6
3.3.3.1 Adding records . . . 3–6
3.3.3.2 Deleting Records . . . 3–6
3.3.3.3 Compression . . . 3–6
3.3.3.4 Locking . . . 3–7
3.3.3.5 Exit and restart . . . 3–7
3.3.3.6 Command Mailbox . . . 3–7
3.3.3.7 Other process interaction . . . 3–7
3.4 Relater . . . 3–7
3.4.1 Description . . . 3–7
3.4.2 Relater database . . . 3–7
3.4.3 Relater Frame . . . 3–8
3.4.4 BOXID Modification . . . 3–8
3.5 Alert Subsystem . . . 3–10
3.5.1 Description . . . 3–10
3.5.2 Alert generation . . . 3–12
3.5.3 Alert processing . . . 3–12
3.5.4 Alert Formatting and Transmission . . . 3–13
3.5.4.1 Data typing . . . 3–13
3.5.4.2 Parameters and translation . . . 3–13
3.5.4.3 Alert log files . . . 3–14
3.6 Counter Processing . . . 3–15
3.7 Counter Processing . . . 3–15
3.7.1 Description . . . 3–15
3.7.2 Counter Processing files . . . 3–15
3.7.3 CNTRPOLL.DAT . . . 3–15

Chapter 4 Programming Notes

4.1 General Notes . . . 4–1
4.2 Acknowledgements . . . 4–1
4.3 System Tour . . . 4–2
4.3.1 Startup and Shutdown . . . 4–2
4.3.2 LISTEN . . . 4–2

4.3.3 PSR 4–3
4.3.3.1 PSR Tour 4–3
4.3.3.2 PSR Database 4–4
4.3.3.2.1 PSR Mapping 4–6
4.3.4 Relater Tour 4–7
4.3.4.1 Relater Frame 4–8
4.3.5 Configuration Pollers 4–9
4.3.6 Main Database - EMUDB 4–9
4.3.7 User Interface 4–10
4.3.7.1 Query Overview 4–10
4.3.7.2 Report Overview 4–10
4.3.8 Alert Mechanism 4–10
4.4 Summary 4–11

Index

Tables

3–1 Ethernet Buffers 3–2
3–2 Protocol Types 3–3
3–3 PSR Control Record 3–4
3–4 PSR Record Header 3–4
3–5 System Control 3–6
3–6 Relater Database 3–7
3–7 Relater Frame 3–8
3–8 Alert Format 3–10
3–9 EMU_CNTPOLL.DAT Format 3–15
3–10 EMU_CNTPOLL.DAT Format 3–16
3–11 CNTRPRC Record 3–16
4–1 PSR First Record 4–4
4–2 PSR Common Record Header 4–5
4–3 Map Structure 4–6

Chapter 1
Introduction

Ethernet monitoring and documentation system is intended to achieve two major goals: Document and Monitor the network.

In it's simplest form, the system listens to the LAN, receives all packets and passes selected packets to protocol specific routines which extract information specific to that protocol. This is used to build an internal database which in turn drives the monitor and documenting functions. More specifically:

- Document:
  - For all addresses:
    - Protocols in use by address
    - Manufacturer
  - For selected protocols:
    - Node name(s)
    - Higher layer address(es)
    - Functions address will respond to
    - Protocol specific data useful for Monitoring and documenting
- Monitor:
  - Provide alert system
  - Monitor param and status changes
  - Collect and monitor counter data (as available)

In this version (5) it is intended to move the system towards a wide area system where the information gathered locally directs the system to collect information about devices not on the local LAN. As topologies evolve, it is clear that a system restricted to the local LAN will be less useful.

1.0.1 Main Features

The two main features of this system are it's ability to build and provide a database of current network devices, it's ability to listen to and understand the real time activity of individual protocols and report problems quickly and in context.
This system (working title is EMU) is a client/server arrangement with the server providing database and alert information to the client.

The system is plug and play: On the server side, the user simply starts the system and it starts listening and building the database. There is little interaction at this level although a programming interface may be provided. On the client side the system starts up and listens for server broadcasts. The information displayed is mainly raw alerts accompanied by addition information from other protocols running on the same address and current counter data. The actual presentation is not yet defined but it is thought that it will be a mixture of text and graphics under windows. See the Client Description for detail.

The system is extensible. It is not possible for any single tool to provide necessary information on every known protocol. A standard interface is defined to allow any number of protocol routines to be present and tools are provided to aid in interface to the system. These tools will be published and made available to users. The current version processes MOP, LAT, LAVC, IP, DECnet,OSI, IP, IPX/SPX, Ethernet and STP (bridge)

The system is self maintaining. To the degree possible/practical, the system provides extensive feedback loops and is aware of bottlenecks in itself. Simple counter data is maintained by each major routine and is made available to a system monitor that can reset base parameters used to start the system. Think about AUTOGEN, what is does and what it could/should be.

EMU can operate primarily as a passive device in which there are no special requirements for any node on the network to be included for monitoring and documentation purposes. In future, node server processes may be available and when run on the client will add significant functionality. It is intended that this be optional.

EMU now requires both a DECnet and IP address and must integrate into the customer's addressing schemes. Also, with the addition of Counter processing, and SNMP and NML access, EMU adds data to the customer net. The amount of data is somewhat controllable and easily estimated should this be necessary.

In addition to VMS and DECnet software, EMU now requires UCX (AKA TCPIP Services) software to operate fully.

It is intended that this product address a critical lack in other products seen:

- It is common for any single network device to utilise multiple protocols in it's normal operations and this trend is expected to expand greatly in the near future. No product seen or heard to date has the ability to make the relationship between multiple protocols and single stations and use this information to enhance management functionality. This is a central objective of this system.
- All tools out there assume that the customer already understands, to varying degrees, both the content of the network and what constitutes 'proper' operation. This is an unsafe assumption that this system does not make. It finds all the devices and monitors them , building over time, a simple profile of what normal operation is for any device. It then reports changes and anomalies.

- In addition to the assumption above, many tools require that the customer manually enter this lack of knowledge into the system, defining the devices, possible errors, actions to take and so on. EMU is self-configuring both in finding stations and the protocols they they are using, and in determining normal operation of the attached network. There is no user configuration required to provide full function although interfaces are supplied if the user wants to name items, group items together or selectively monitor components. A comprehensive reporting package is inherit.
- Many tools, particularly SNMP based ones use polling almost exclusively to gather data and determine status. At this writing, more than 80% of the information known to the EMU system is gathered by passive listening. EMU does not contribute to the loaded network problem while attempting to solve it.

In the longer term, it is intended that the system become a sort of 'system service', providing a dynamically updated network information on request from any client. The client can also set up a window into the server to receive current alerts and initiate specific monitoring functions. This product is not intended to be 'yet another Ethernet monitor'.It does not count or record traffic volumes beyond those counts that are necessary to monitor a specific protocol. Neither does it display top talkers and listeners and it has only rudimentary ability to show the traffic relationships between protocols. Those types of monitors can be classed as Level 2 devices in that it is at that level that they read and analyse data. This is a layer 3 monitor in that it uses layer 3 (and above in some cases) data to determine protocol activity. It's primary purpose is to provide an up to date picture of the network that network and system managers can use to determine the presence and state of protocols on the net and use this information to monitor the net and automatically load any other monitoring tool with current information.

Chapter 2
System Overview

The system is currently described in 2 major divisions:

- Core system
- Applications

The major distinction between a process existing at either level is the internal database and write access to it: Core routines have direct access and applications do not. In implementation there may be difficulty in maintaining this distinction clearly but it requires persuasive reasoning to violate it.

2.1 Core System

The core system is made up of a number of processes and structures who's main objective is to document the network and supply an API to the application layer who's main objective is to monitor the network.

2.1.1 Listener

The listener is thoroughly documented elsewhere. Suffice to say here that it provides one of many inputs to the PSRs and acts as a source of data.

2.1.2 PSR

The array of PSRs provide the first level of 'understanding' in any particular network. A network is composed of multiple devices cooperating via a protocol. Most protocols provide an internal management facility in order to maintain orderliness and will exchange information on the content and events within that network among themselves. It is this 'conversation' that the Listener picks up and passes to the PSRs. Each PSR is responsible for a single protocol and extracts data useful for management purposes and stores it. Additionally as network changes ensue, the PSR detects and alerts those changes to higher layers. A fully implemented PSR will usually be implemented in a number of standalone but intercommunicating processes. Fully implemented includes the following:

- Ability to receive protocol data units (PDU) directly from the network and extract information useful to EMU purposes.

- Provide a poller function to inquire of the network further data to enhance EMU usefulness.
- Provide a process to receive and process any asyncronous events the network may generate.
- Provide an analysis function able to correlate the inputs from the other processes and make determinations as to protocol address and/or network integrity and functionality.

2.1.3 PSR interface

It is essential to the success of this architecture that new PSRs can be added, changed and deleted easily. It is a fundamental goal here to add new functionality at this level as dictated by events and to retire older, less useful functions. Further, as functions change, they must integrate seamlessly with the supporting structures. In practice, the Listener(s) and PSRs are the centre of the core.

2.1.4 Network Views

There are a number of underlying principles driving the database structures. The most fundamental is the required 'view' of a network. From one perspective, the network is a collection of devices all connected together and able to transport data from any point to any other point. From another perspective a network is a system offering a service wherein physical location of resources and the complexities of providing that service are hidden from consumers. That is to say the network is an extended computer system, providing a seamless end to end service. Either view is valid and it is a goal to support both views. An EMU user can view either a network and it's collection of devices or devices and it's attachments to various networks. As such the PSRs build network specific databases and pass on information about devices and other networks to another process that forms the relationship of any particular device to any particular network. A secondary design goal is to limit the amount of traffic EMU adds to any network to absolute minimum while effectively attaining it's more primary purposes. As such the following is to be observed:

- The main source of data for any PSR is that which the network uses to control itself. This data is listened to by the system and understood by the PSR. A more successful PSR understands this conversation and is able to extract more information from it than a less successful PSR.
- EMU does not attempt to monitor all available data on the network. Primary parameters are those contained in the control messages sent by the network. Secondary params are those deemed to be essential to network operation but must be polled for.
- The majority of useful parameter data can be gathered from the network on an as necessary basis - that is to provide a complete report or similar.

2.1.5 Relater

Information gathered on any single network often has implications for (and about) other networks, particularly when multiple networks interconnect as is common practice. It is a responsibility of the PSRs to make this information available and it is the relater that acts upon the results. The relater essentially builds the 'device' level database and ensures the relationships the PSRs detect are propagated correctly. As such the relater is part of the 'centre of the core'.

Essentially, the relater database is device specific. Each record specifies EMU's view of a single device, the protocols that it implements (that is the networks it attaches to), and other device specific information such as device type, class and services. Additionally it specifies which EMU server contains the network level information for this device. It does this with the BOXID.

2.1.6 Identity

For both internal purposes and external representation it is necessary to uniquely identify any single device. Internally EMU defines a BOXID - simply a number that is used to relate disparate pieces of data together. In EMU, the basic unit of information is an address/protocol combination. That is a known address and the protocol it runs is considered an atomic structure and they are indivisible because knowledge of each part is useless without knowledge of the other. This unit is termed a protocol address. As information becomes available, various protocol addresses are found to exist on single devices and via various mechanisms, further device and/or network specific information is gathered. The relater table is the place where this information is correlated and the key between this table and the network databases is the BOXID. This identifier is quite fluid - as any protocol address is detected it is assigned a unique BOXID and if, as and when it is found to relate to another BOXID the number is changed so that the related data is keyed with the same BOXID. The BOXID is made up of 2 distinct parts:

- EMU part. Each EMU server is assigned a number. In implementation it simplifies a number of processes if this ID directly relates to it's network address in that it is guaranteed to be unique in the network and where it is located is immediately known without requiring further translation.
- Local part. This is simply a number unique in the local system.

2.1.7 EMU network

EMU servers cooperate in their own private network. The goal here is to have each server contain the complete list of protocol addresses on the network. Each instance of a protocol address is assigned 'responsibility' to a single server. That is only one server will undertake to monitor any particular protocol address. This is to prevent multiple servers from polling a single device.

Each EMU server periodically sends changes to it's network databases to all other EMU servers on the network. The EMU server to server update exchanges information on protocol addresses. The system must have sufficient information to decide which EMU server any protocol address is local to. Essentially this is supplied by the combination of the protocol address and the management capability it offers to a particular server. The majority of protocol addresses seen from multiple servers will provide identical facilities to all but there are some circumstances where this is not true. The system then compares the management capabilities as seen from each server and sets the location based upon that server with the greatest capabilities. In (the usual) case of equality, a scheme ensuring only one server has 'responsibility' needs to be present to ensure a single device is not being polled from multiple servers. Additionally, the following rules must be applied in an update message:

- Send only those addresses detected locally. Do not send any updates containing addresses that were created or changed as a result of a received update.

- Each update will contain the BOXID and the protocol address it refers to followed by an action. the actions may be:
  - New (or updated) protocol address showing a management level. This could be simply a number with a higher value indicating more management capability.
  - Delete. The sender has deleted this from it's database. the Receiver must also delete it.

2.1.8 Database Overview

The database is divided into protocol specific sections with each section independent of all others. Thus there is a LAT database, an IP database and so on. Above this is the relater database which is in effect the device database. That is the relater collates information from the PSR databases and correlates the addresses that coexist on any single device. In each case the database itself is divided into 3 sections:

- The first record in any database describes the size of the following structure. That is it supplies the size of the records and number of each type of record to follow (at minimum). This record is used to initialise the database at startup. It is written at shutdown.
- The next section is termed the summary section and it's contents and use are directed by the protocol. It is effectively a summary of network contents and activity on this protocol.
- The remainder of the database is made up of the individual records of addresses appearing on this network.

2.1.9 Record Structure

While the contents and size of any record in any database is dictated by the needed and available information from the associated network, there are a number of common features that each database must adhere to:

- The database is created and maintained by a single process. This means that only that process may add and (physically) delete records and further it is at that process's discretion if as and when the database is available.
- Each record any any database begins with a standard header.
- Each parameter stored is typed with a standard descriptor.

2.1.10 Record Header

The record header supplies information to accessing processes in standard format so that these processes can access any database using common algorithms. The header, it's size and contents are strictly defined and it is required that any participating database implement it. An extract will make the purpose clear:

- BOXID. Assigned and written by Relater.
- Other protocols present.Assigned and written by Relater.
- System control bits
- Status. A field showing current status of this protocol address.
- Time last packet this addr.

- Time last alert this addr.
- Time this node 1st heard from.
- and so on...

The data record itself can only be written by the PSR (and possibly some associated pollers and such) but the header allows specific process to write specific areas in the header. Note that any area allows only a single writer - any specific field can only be written by a single process. Thus the BOXID is written by the Relater and can only be written by that process. Immediately following the header is the data record for a particular protocol address.

2.1.11 Database Access

There are 2 standard keys that any record in the database always has: The BOXID and the protocol address. Beyond this, a lookup routine must be able to specify a series of specified params used to find records. Wildcarding of any param or address is supported. The BOXID is internal and not displayable. The return is always a pointer to the record satisfying search criteria and a context variable used internally to find the next record in a recursive search. For security purposes any field not accessible by the caller is not displayed and any search specifying a field not accessible is rejected. Given the structures above any parameter stored by EMU is directly addressable by specifying the database, a record key (any field) and a symbol specifying the offset to the parameter . In this context a parameter is the combination of it's Datatype and value. For most purposes this structure is indivisible.

2.2 Counter Processing System

This subsystem is intended to be a universally useful system used to process and store a related stream of numbers showing their relationship over time. It has the following basic features:

- Stores a limited number of previous samples received and the time each was received.
- Processes each sample as it is received into the tables and determines:
  - Long term average - The average of all samples received.
  - Short term average - The average of the previous 48 samples.
  - Min/Max - Stores the highest and lowest values for this stream.
- Calculates the movement range in both long and short terms. That is, the system determines what the normal range that this stream of samples is in and expresses this as a percentage of the current average.
- Adjusts the range of each stream based on received values.
- Calculates thresholds based on current averages and movement ranges.
- Returns a status block to the caller showing which (if any) thresholds were exceeded with an error value relative to the amount and number of thresholds exceeded.

In general, the following sequence is used to identify addresses with useful data, and include them in the counter system.

- On each cycle EMU_CNTPOLL determines which counter analyser to call for the set of counters and calls it with the address param. If the counters do not have CNTID set, they are registered with EMU_CNTRPRC and that routine returns a CNTID - a unique

ID to be associated with this counter. Each received counter is passed to CNTRPRC with it's CNTID and if the return specifies any error status, an alert is generated.

- On each cycle CNTRPOLL determines if it is any use to continue polling this address for data. If the address answers less that 10% of attempts with good data, the record is removed from the database and this is flagged in the DB1 MOP record. This flag prevents the address from being inadvertently reintroduced to the database.

The counter system, is intended to be universal. That is any stream of related numbers can be processed through it with only 2 prerequisites:

- The samples are integer only and must be 64 bits wide.
- The samples must not be negative.

However, to provide usefulness the following should also be observed:

- The samples should be passed to the subsystem on a regular, cyclical basis. The resulting outputs are less meaningful if the time between samples is highly erratic.
- Each counter uses significant disk space processing time. For a single MOP node about 6300 bytes of disk space is used and again, some processing time. The point is that the system is a heavy user of resources and should be called upon only when the returns are useful and worth the effort. To that end, one of the calculations always performed is the number of times a node was polled for data and the number of times it responded with good data. Should this fall below 10% the entry is deleted from the system.

Chapter 3
Implementation

3.1 Introduction

This is the 5th major revision of EMU and it's 3rd complete re-write. While many of the essential ideas and approaches are preserved, virtually all of the data structures and many algorithms are changed. This section specifies an implementation of the architecture described previously.

Core

The core routines are the basic building blocks upon which the rest of the system depends. The individual routines are described here.

3.2 Listener

3.2.1 Description

The listener assigns a channel to the first Ethernet device found on the local system and enables it in promiscuous mode enabling it to receive all frames on the cable. Through logic described below it distributes selected frames to the PSRs for further processing.

3.2.2 Processing abstract

- Acquire control lock to ensure exit at system shutdown.
- Open error log. See Error processing for details
- Map to:
  - Monitor section. See system monitor section for detail.
  - Control Section. See Control section for detail.
  - Ethernet buffers. Details below.
- Create command mailbox and que read with AST.
- Create Ethernet buffer address. Details below.
- Associate with CEF 63-127. (64 flags)
- Call ASSETHCHN to locate and assign channel to Ethernet device

- Que all buffers to Ethernet.
- **Main Routine:**
- On any flag set:
- Locate buffer using flag set
- Validate source addr and Dest addr
- Determine and set PTYTYP. See Detail below.
- Apply all defined filters:
- Lookup PSR table for processor
- If found set corresponding bit in frame.
- If at end any bits are set, set the corresponding CEF(s). If not reque to Ethernet
- Scan outstanding table and reques any returned buffers.

3.2.3 Ethernet buffers

There is an architectural limit of 64 buffers set by the maximum number of Common Event Flags available in VMS. This implementation uses 32 for Ethernet buffers reserving the remaining 32 flags for other uses. Each buffer is associated with a single event flag through the Ebuff_addr table set up at init time. The table is a list of addresses of the buffer and is 32x4 bytes long. After mapping Ebuffs, the start address of this section is placed in the first location of the table and this +1552 is placed at the second location and so on for all 32 locations. Thus when a buffer is read off the Ethernet, locating it is a simple task. The buffer itself is used as follows:

Table 3–1: Ethernet Buffers

Offset	Len	Description
0	4	Target process flags
4	4	Return Flags
8	4	Buffer Number
12	4	Spare
16	8	IOSB
24	20	Ethernet header
44	4	Buffer number
48	1500	Data
1548	4	Spare

Notes:

- The Ethernet header is as returned by VMS and NOT as on the network. See VMS doc for detail.

- IOSB is the input output status block as returned by VMS. The listener must check status completion and all other processes use the len field in processing.
- Data can be up to 1500 bytes long (Ethernet specification).
- Target Flags, Return Flags. When LISTEN determines a buffer should be passed to a process, it sets the processes corresponding flag in Target. This collection is then used as a mask to set the corresponding CEFs, waking up the process(es). Once the process has completed, it sets it's corresponding flag in the return field. When all Target Flags are matched by Return flags, the buffer may be requed.

3.2.4 Protocol type formats

There are 2 major Ethernet types and within 1 type there are 2 subtypes - effectively 3 types in total.

- Ethernet type 2 - also known as DIX, specifies a 2 byte protocol type and in the VMS return is found at offset 12. Byte 14 - 2- are not written. We will know this as type 1
- 802.3 also known as IEEE specifies Service Access Points (SAPs) with the source SAP at offset 12 and the destination SAP at offset 13. Byte 14 is written with a control field. A SAP is often meaningless to us though some are universally defined as protocol types. We know this as type 2.
- SNAP. An IEEE frame with both saps set to %XAA that is byte 12-13 are %XAAAA. This defines the SNAP SAP and means that a 5 byte protocol field follows at offset 15. The is EMU type 3.

In order to speed processing. EMU converts these to an internal standard format as follows:

Table 3–2: Protocol Types

Offset	Len	Description
0	1	EMU defined type - as above
1	5	SNAP protocol type
4	2	Ethernet protocol type
5	1	IEEE source SAP

Note that any part of the above not defined for a particular type must be zero. That is bytes 1 - 4 are 0 in type 2 and bytes 1-3 are zero for type 1.

3.3 PSR

3.3.1 Description

The Protocol Specific Routines are exactly that - they process all frames received for a particular protocol, extract the useful data (formatting as necessary), build and maintain the database and perform basic analysis as possible with the data available.

Each PSR is responsible for it's own separate database. It creates, saves and restores it largely under it's own control and discretion. The database itself is made of fixed length records in memory and is in 2 parts:

3.3.2 Control Record

The 1st record of all PSR databases is a control record that the PSR uses to map to the section. The size of this record is set by the record size for that PSR but cannot be less than 64 bytes (standard header) The 1st (undefined number) of fields in the control record are standard and the PSR may use the remaining space for it's own purposes:

Table 3–3: PSR Control Record

Symbol	Offset	Description
COM_DBHDR_L_ENTRIES	0	Number of physical entries
COM_DBHDR_L_RECSIZE	4	Size of each rec
COM_DBHDR_L_MAXENTRIES	8	Max number of physical entries
COM_DBHDR_L_FLAG	12	EMUPID of the DB owner

This record is read by the PSR on startup and is used to size the section initially. It is written at exit or any time a re-size is executed. This record is the same length as all records in the associated DB so will be various lengths in different DBs. Some part of the rec should be reserved for PSR specific needs.

3.3.2.1 PSR Records

This is the familiar DB1 structure with a new header. All recs have this header:

Table 3–4: PSR Record Header

Symbol	Offs	Len	Description
COM_HDR_L_FLAGS	0	4	Flags below
COM_HDR_L_BOXID	4	4	Unique device id
COM_HDR_L_PTYBITS	8	4	Other protocols present
COM_HDR_L_SYSCTL	12	4	System control flags
COM_HDR_Q_LSTHRD	16	8	Last time heard
COM_HDR_Q_LSTALT	24	8	Time last alert sent
COM_HDR_Q_FSTHRD	32	8	Time 1st heard
COM_HDR_L_STATUS	40	4	Current status
COM_HDR_L_ACNT	44	4	Count of access this rec
COM_HDR_L_LEN	48	4	Len of KEY
COM_HDR_L_HOWSET	52	4	How this addr found
COM_HDR_Q_LOCKFIELD	56	8	Count of current accesses
COM_HDR_L_READLOCK	56	43	Count of read accessors

Table 3–4 (Cont.): PSR Record Header

Symbol	Offs	Len	Description
COM_HDR_L_WRITELOCK	60	4	Count of write accessors
Configuration Monitor	64	4	

1. FLGS. The EMU defined PID for this process.
2. ID is unique to that box an may be duplicated in many PSRDBs when the box runs multiple protocols on multiple devices. It is the method EMU uses to 'tie' addresses to boxes. Made of 2 parts: bytes 0-2: locally generated ID, byte 3: EMU station ID that target is local to. Results in a unique ID over space.
3. PTYBITS is a bit pattern wherein a bit set indicates that this box runs that (EMU supported) protocol. Thus if DECnet, MOP and IP all appear on a specific Ethernet device the appropriate bits are set and propagated to this field in each DB. Use of this field is at 1st two-fold: Indicates to the PSR that the corresponding protocol is known and exists in the corresponding DB. Indicates to a searching process where to find additional data for this box. The PSR will use this field when it discovers another protocol DB should contain this address. If the bit is not set it sends the info to the relater and continues to do so until the bit is set.
4. SYSCTL. System Control bits. Defined below
5. STATUS. Status bits. defined below.
6. LSTHRD. Time the last frame received this address.
7. LSTALT. Time the last alert sent this address.
8. FSTHRD. Time the first frame received this address.
9. LEN. The len of data after the header. Constant in any single database.
10. HOWSET. or more accurately how found. Each process in EMU is assigned an ID for various purposes. The process that found this data and caused this rec to be created is set here. Additionally the field could indicate 'set by management' if we allow user input to create recs. Essential field in troubleshooting.
11. ACNT is a count incremented whenever this rec is accessed. Since all access is by sequential search, arranging the recs so that the most often accessed are nearer the top is an advantage.
12. Spare. Exactly that - spare space for unanticipated needs.
13. LOCKFIELD. Should a process which to access the record, if call the locking routine with the type (read or write) of access required. That routine allows shared read or shared write but not shared read/write. If it grants access, the appropriate field is incremented so the count of current readers and writers is kept in the record. Unlocking reverses this - the appropriate field is decremented.

14. Configuration Monitor area. Area used by the config monitor. See that routine for details.

Table 3–5: System Control

Symbol	Bit	When Set
SYS_COM_V_SCDEL	0	This rec is deleted
SYS_COM_V_SC_UPDATE	1	Update this rec next cycle
SYS_COM_V_SC_NOPOLL	2	Do not poll this address
SYS_COM_V_SC_PSEUD	3	Pseudo Node - alias
SYS_COM_V_SC_CNTDB	4	Has been added to Counter DB
SYS_COM_V_SC_DISCNT	6	Counts disabled
SYS_COM_V_SC_PROPOG	7	Addr has been propagated to EMUDB

3.3.3 General processing

At startup acquire the exclusive access lock and open the file. If not found generate one using defaults either embedded in the PSR code or system params. The 1st rec gives the layout and info needed to create the section. Create it and read the records in using the ACNT field as the key (descending). This starts the DB with the most accessed recs at the top. Write the size of the section to the LOCK VALUE BLOCK to signal to other processes the current size and release lock allowing general access to section.

3.3.3.1 Adding records

In all cases, the protocol address is unique in the database. Search down for this address (ignore any deleted recs) and if not found add it. During search note 1st deleted rec. If deleted rec found, overwrite with new else add it at end. Update Control rec. Zero ACNT field in header Note that ONLY ONE process can create/delete the records while individual fields may be written by multiple processes.

Note that any protocol address is integrated into the system via BOXID only - it is the key by which all relationships are known. When a record is added it sends the alert to the Relater with the BOXID = 0 and no parameters. A common routine generates the BOXID and guarantees uniqueness.

3.3.3.2 Deleting Records

Same as add - list is keyed by address. Recs can be deleted by address (single delete) or by ID (multiple delete). Set the delete bit in any rec deleted.

3.3.3.3 Compression

If no deleted recs and no space invoke a compress routine that may ageout or otherwise identify recs to delete. If none are found then invoke end processing (writes new params and exits - Control process will restart with new params)

3.3.3.4 Locking

Normal mode has readers holding a concurrent read and writers holding a concurrent write lock respectively, allowing unlimited readers and writers. With the exception of the PSR, locks are queued with a blocking AST. Should the PSR need to compress or re-size the DB, it requests an exclusive lock, causing delivery of the BLAST to all other accessors. In this case the BLAST routine deaccsses the section and deletes VM and waits for the section to recreate and the exclusive lock to release. The new size is in the Lock Value block.

3.3.3.5 Exit and restart

An exit handler is included in all PSRs to guarantee (within reason) that the last action is to write the DB. First scan the DB to determine how many undeleted recs will be written. Calculate from this what the size of the section should be on next re-start (some room for expansion) and write this to the control record. Write the control record followed by all (undeleted) DB recs.

3.3.3.6 Command Mailbox

Each PSR has a command mailbox defined. Periodically the PSR reads this mailbox to determine if any messages are waiting. There are no specified uses for this facility but it seemed a good idea at the time and might even prove useful at some stage. It stays.

3.3.3.7 Other process interaction

When mapping to any DB section, the subroutine will acquire the appropriate lock (share_ read, Share_write) and set up the blocking AST. The AST routine deletes the virtual memory and releases the lock. This implies that ASTs must be disabled during all (write anyway - not sure about read) so that partial records are not written.

3.4 Relater

3.4.1 Description

The Relater accepts inputs from many sources and essentially builds and maintains the device database. A device in this context is a single physical processor that at minimum is attached to a network and cooperates in 1 protocol. While the PSRs build and maintain the Network databases, the Relater is device specific.

3.4.2 Relater database

The relater database contains the BOXID and all EMU supported protocols along with some device specific information:

Table 3–6: Relater Database

Field	offs	Description
BOXID	4	Key to other databases
PTYBITS	4	Bit pattern showing protocols present
MGMTBITS	4	Management capabilities
Device type	4	See below
Facilities	4	See Below

Notes:

- Class. Number indicating the relative importance of this device. It is (privileged) user writeable.
- Services. A bit pattern showing the OSI level(s) this device operates at. A bit set indicates the device supports the corresponding OSI layer.
- BOXID Table Table of BOXIDs that exist on this device. Within each DB a single BOXID associates the addresses together such that a device with 3 IP addresses will have 3 entries in IP database with the same BOXID. This BOXID is placed in the relater table at the IP offset. The Offset is dictated by SYS_PID_C_xxxx in sysdef (that is the IP offset is SYS_PID_C_PSRIP (7). The current table size accommodates 24 entries.

3.4.3 Relater Frame

Any process wishing to send info that the relater acts upon sends the following frame to it:

Table 3–7: Relater Frame

Field	Len	Description
BOXID	4	This device ID
SENDER	4	Sender's PID
Message Type	4	Add, delete or start
Targets	4	Count of targets
Data	Var	1 target structure per target

Notes:

- Message type defines the action the relater will perform and the format and content of the data field. Each message type specifies strict format and this is used to verify the received message.
- Sender is the identity of the sending PSR. It is used to locate the correct database and defines the contents of the record.
- Data is specifically formatted according to message type.

3.4.4 BOXID Modification

If as and when a PSR or other process finds an association between this address and another protocol, and the bit associated with this protocol is not set in the header, it sends the following frame to a process called the RELATER:

- BOXID = Value written in this rec header
- Message type = 1
- Message len = Var

- Data :
  - Number of targets
    - Target PSRID
    - Target protocol address

All fields must be present. Each PSR has an ID which may be used to control many things but here it denotes the format and length of the following address. The Relater puts the ID on a subject list and executes the following algorithm:

Locate each target on list and check the ID. If .eq. 0 copy subject ID to it and continue. This is an unusual case.

- If set and .eq. to input, continue.
- If set and .ne. to subject add to subject list, overwrite and restart.
- At end (all possible locations scanned and made equal) delete all references to ID's overwritten in relater table.

Other messages will be defined to allow PSRs to send data that belongs at device level to the relater.

3.5 Alert Subsystem

3.5.1 Description

The alert system exists to allow output of important events to the human interface. It's goal is to provide a convenient interface for both sides. For processing and architectural considerations, the alert subsystem is broken out into 3 distinct stages:

- Generator. Any process wishing to notify the operator through the alert mechanism for any purpose is called the generator.
- Processing. The 1st level where the alert is compared with other alerts, further information is added and passed to next stage.
- Format/Transmit. The alert is formatted for screen output and sent to all currently connected clients.

Table 3–8: Alert Format

Field	Len in Bytes	Description
Time	8	System time
Expirey	4	Number of Seconds
Sender	4	EMU process ID
BLINK	4	Location of address sending
Device	4	Device Type
Priority	1	Priority
Class	1	Perf,config etc
Recent	1	Number of recent events
related	1	Number of related events
Param Count	1	No. of Params in data field
Display Name	49	.ascic name displayed
Spare	12	Future
Data	var	Params

A binary alert minimum length is the standard header (88 bytes) and can be up to 512 bytes.

Notes:

- Time is the system time the alert was logged at. It is set by the generator and not changed thereafter.
- Expirey. If this field .gt. 0 then the message is placed in the timed que. If an identical message is in the que (and not expired) then the alert is generated. See Timed Alert que below.

- Sender. The EMU defined identity of the sending process. It is used to locate the correct database of the sending process.
- Blink. Backward link. Offset to the record for which this alert was sent. In combination with the Sender field, and record in any database can be located quickly.
- Device. The device type this alert is for. Device is an EMU Datatype (which see). This is a placeholder that layer 2 fills in.
- Priority is dynamic through the system. The generator sets the initial value according to some scheme (needs consistency) and the processor can adjust it in either direction depending on other related conditions (other alerts, other protocols down etc).
- Class is the general area the alert falls into. This is OSI standard:
  - Accounting
  - Performance
  - Configuration
  - Fault
  - Security
  - system * This is a bit field so that any alert can be classified in multiple types - Often a config change has security implications. The system class is not part of the standard but we will use it to alert problems with EMU (disk full etc) to the user(s).
- Recent. Layer 1 will determine if recent events are present, adjust the priority if indicated and pass a count of these with the alert. The user interface uses this to advise the user of their presence. Recent events are defined as any event of any class for this address in the previous 24 hours.
- Related. Again. Layer 1 works out the number of related events and passes this as a count. Related events are defined as any event of the same class on any node logically on the the same network as the address the event is for. Some examples to make this clear:
  - If the event is DECnet protocol down, any other node currently down in the same DECnet area is a related event.
  - If the event is IP down, any other IP node currently down on the same subnet is related.
  - So, DECnet is divided into areas, IP into subnets and IPX into subnets. This works for routed protocols but others such as LAT, MOP and bridging do not fit well here. It is for this reason mainly that future versions of EMU will begin the efforts of determining the physical topology of the network.
- Param count. The number of parameters found in the data field.
- Display name. This is a placeholder filled in by layer 2. Acceptable values here are:
  - Valid BOXID. That is a BOXID that exists in the relater table.
  - -1 The alert class must be system and there is no BOXID. The system is sending a message.

- Any other value results in an error and the message is rejected.

3.5.2 Alert generation

Any process can generate an alert and pass it to the system for processing. All fields must be present with the following exceptions:

- Device type. Layer 2 fills this field in (if known)
- Recent,related and param count may all be 0.
- Display name. If 0, this layer assigns and writes a unique ID.

The alert may include no parameters or any number of params up to the max len of 512 bytes. Parameter formats are given below.

The goal here is to simplify the processing the alert generator will have to do in order to get an alert out, and allow any params to be present or missing.

3.5.3 Alert processing

Here is where we accept a single alert and piece it together with other related events and objects in an attempt to show the alert as it relates to other problems on either the node or network. Basically, the alert generator (Usually PSR) sends an initial alert to the alert processor where the following happens:

- If the BOXID is 0 (not present) this layer assigns a new BOXID, creates a record in the relater table and writes it to the location given by the following parameters. Skip all remaining processing and pass 'new protocol' to next layer.
- If the class is 'fault' get status by calling EMU_NODE_STATUS. This routine will perform all currently available tests on the node and return a complete status report in specific format. If all protocols respond then ignore (positive check). Set priority by adding fixed value for each protocol down
- All alerts: Look up BOXID for last alert this address, If < 24 hrs old then: Search log file for other alerts this address. Log file is an indexed file. If any are found, insert symbol at end of current param list showing start of recent event list and record the key of each one found. Put the count in the header. The number of recent alerts will be limited by the first occurrence of MAX_RECENT_ALERTS or total message len (512). Raise priority according to number (and priority) of recent events.
- Fill in device type field. PSR does not usually know this. Adjust priority according to device class.
- FUTURE: Allow for 'related' events. Again, sear log file and supply count and record number of each related event. See below for description of related events.
- Set displayed name for this address. First occurrence of:
  - Name if known - protocol specific if appropriate
  - Protocol Address

- Because each class will have to have a separate routine, it becomes possible to institute some sophisticated alert processing in future. Some examples: If alert is a performance class and the alert is 'excessive collisions' the routine could look in the Ethernet DB for other nodes sharing this segment and determine if they are also experiencing 'excessive collision'. If so report the segment in trouble, not the node. See segmentation below.
- If alert is configuration class and alert is 'parameter change' then check param value against user provided template for legality. Adjust priority accordingly. See below for description of templates.
- Note that an alert can exist in multiple classes. Thus some config class alerts, particularly param changes may have security implications and will also exist as security class alerts. This process will make that determination via templates and/or hardcoded routines.
- This is also the point where we will provide the hook so that the user can supply external procedures to run when specific events occur.
- Log alert. Set time field to current time and log the error using the time as the key. The file is indexed and set to noduplicates. If the record does not store because of duplicate key, simply repeat the get time and store until it does. This guarantees a unique key.
- Look up the address in the index and copy the time to the last alert field. (new index field)
- If there are no clients currently connected then end, else pass to Layer2 for formatting and transmission.

3.5.4 Alert Formatting and Transmission

3.5.4.1 Data typing

The alert is only partially processed and displayed in brief format.

Brief format is the translation of the header (all fields always present) + the first param (if present). Because each type produces both the translation of the param and an associated string, a single param is adequate to produce a brief but useful message to alert the operator. The operator can then selectively display the full format which is the header and all params formatted and displayed.

3.5.4.2 Parameters and translation

The display process will translate the header in standard format:

- Lookup class and put address of string on list
- Put address of time on list
- Put address of display name on list
- Convert BOXID to DISPLAY_NAME to name and put address of string on list
- If any params present, translate param 1 and put address of resulting string on list.
- Set colour of display string according to priority. Very simple: low priority alerts are green and hi are red. A very simple calculation allows the priority to set the 'redness' by adjusting the green down and the red up. Thus we can show priority clearly in 255 equal steps.

- Display alert on all clients currently connected and having alert screen present.

In implementation the translation is in two parts: Part 1 translates the individual params according to the directives and modifiers producing both the FAO directive string and the faolist and then part 2 assembles the final message.

Note that for initial display the message is only part translated and displayed - that is in brief format. Only when the operator selects the displayed alert for expansion does the entire message become translated and displayed. Thus the programmer need not be concerned with message length and such but just include any and all relevant data.

A symbol file will be supplied.

3.5.4.3 Alert log files

Above implies a lot of log file access and disk is too slow, so... The alert log file can be made as a RAM disk and limited to storing only alerts for the previous 24 hours. After 24 hours the brief version is written to disk and removed from the RAM disk. At 512 bytes (max) per alert we can store 2,000 alerts/ MB of RAM space so a 5MB RAM disk should be more than adequate. If the disk file is limited to storing only the header (+- 100 bytes) we can store something like 10,000 rec/MB while keeping enough info for history lookups. This will relieve the current problem of the alert file growing uncontrollably. The message file could also be copied to the ram disk for faster processing.

3.6 Counter Processing

3.7 Counter Processing

3.7.1 Description

The counter system supplies a standard method of determining normal ranges any counter moves in. It can then be used to determine abnormal operation of the monitored device.

3.7.2 Counter Processing files

There are 2 data files , one standalone routine and one callable object associated with the counter system:

- CNTRPOLL.DAT. Contains the list of addresses and counters that will be polled for data on the next cycle. Any process detecting that counter data is available (and desirable) can add to this file.
- CNTRPRC.DAT. Contains the sample data and so on for each stream.
- EMU_CNTPOLL.EXE. Is the main standalone process that initiates polls to the devices and submits the raw results to CNTRPRC.
- EMU_CNTRPC is a callable routine that processes the streams as described here. This is the main counter processing routine.

3.7.3 CNTRPOLL.DAT

This file contains entries that the CNTRPOLL routine reads on each cycle. Each entry specifies the routine that requests and formats the counter data and address to request from. Additionally, it control information and indexes to the data in CNTRPRC.DAT.

Table 3–9: EMU_CNTPOLL.DAT Format

Field	Len	Description
BOXID	4	Identifies device
Routine	4	Counter processing routine
Last Seen	8	Time last successful poll
Polls	4	Number of polls this addr
Success	4	Number of answered polls with good data
Address	8	Protocol address of device
Counter Tbl	488	Counters - See table below

The file is indexed and the key is the 1st 8 bytes of the record (BOXID/Routine). Each 512 byte record will accommodate 40 individual counters. Each counter is stored as follows:

Table 3–10: EMU_CNTPOLL.DAT Format

Field	Len	Description
Name	8	Record header
CNTID	4	CNTID assigned by EMU_CNTRPRC

The name field is the standard PARAMTBL entry given to all params:

- Table 4 bytes
- Param no. 4 bytes

The CNTID is an index to the CNTRPRC database. In general, the following sequence is used to identify addresses with useful data, and include them in the counter system.

- Among it's many functions, SETFUNC determines if and under which protocols any address responds to defined counter requests. When this is identified, SETFUNC adds a record with the appropriate fields filled in to this database.
- On each cycle EMU_CNTPOLL determines which counter analyser to call for the set of counters and calls it with the address param. If the counters do not have CNTID set, they are registered with EMU_CNTRPRC and that routine returns a CNTID - a unique ID to be associated with this counter. Each received counter is passed to CNTRPRC with it's CNTID and if the return specifies any error status, an alert is generated.
- On each cycle CNTRPOLL determines if it is any use to continue polling this address for data. If the address answers less that 10% of attempts with good data, the record is removed from the database and this is flagged in the PSR record. This flag prevents the address from being inadvertently reintroduced to the database.

Each count is processed into the following record:

Table 3–11: CNTRPRC Record

Field	Ofs	Len	Description
CTP_W_CNTID	0	4	This record number
CTP_L_SPARE	4	4	Align/spare
CTP_L_LTCNT	8	8	Long term count
CTP_L_LTTOT	16	8	Long term total
CTP_L_LTRNG	24	8	Long term range % movement
CTP_L_STRNG	32	8	Short term range % movement
CTP_L_STCNT	40	8	Short term Counter
CTP_L_MAX	48	8	Max Value seen
CTP_L_MIN	56	8	Min Value seen
CTP_L_LASTSN	64	8	System time last Sample

Table 3–11 (Cont.): CNTRPRC Record

Field	Ofs	Len	Description
CTP_L_TBLPNT	72	8	Index Pointer to current sample
CTP_TL_TIMTBL	80	384	Time in Sec since previous Sample
CTP_TL_SAMTBL	464	384	Last 48 samples

Notes:

- CNTID is the key for the indexed record and is the unique ID stored by the caller to process further samples. It is generated at create time by this process.
- LTCNT. Count of samples added to long term total.
- LTTOT. Sum of all samples. Routine to reset on overflow is included.
- LTRNG. Long term range. The %(+ or -) that the long term average is allowed to move in and not generate a warning. The long term average is calculated by dividing LTTOT by LTCNT. LTRNG is then divided by this sum, multiplied by 100 and added back to the sum to find the current high value allowed. If exceeded a warning is written to OUTP and LTRNG is incremented. If not exceeded LTRNG is decremented. Over time LTRNG indicated the 'normal' range this counter moved in. A similar calculation is performed to find if the current sample is below the low range.
- STRNG. Short term range. It is similar to LTRNG except that the values used to find Short term average are taken from the table (last 48 samples).
- MAX. The highest value seen for this counter.
- MIN. The lowest value seen for this counter.
- LASTSN. The time the last sample for this record was received.
- TBLPNT. Pointer to current sample in tables.
- Spare. Spare space.
- TIMTBL. Table of times at which corresponding sample in SAMTBL was received.
- SAMTBL. Table of last 48 samples. Both tables are circular buffers in which the oldest value is overwritten by the newest.

The counter system, is intended to be universal. That is any stream of related numbers can be processed through it with only 2 prerequisites:

- The samples are integer only and must be 64 bits wide.
- The samples must not be negative.

Note that counters must be presented in the standard format. This means that for any counter subsystem there exists at least a routine to request and receive the data in the protocol required format and an analyser that transforms the raw data into meaningful information for this system. As of this writing, only the MOP counters are fully implemented but others are well on the way:

- DECnet Phase IV one each for:
  - Executor counts
  - LAPB lines
  - Ethernet lines
  - LAPB Circuits
- SNMP:
- System
- Bridge

Chapter 4
Programming Notes

This section is aimed at programmers. Whether you want to extend EMU, pick it apart and recycle code or are just interested in it's mechanisms, this chapter is for you - you sad case.

4.1 General Notes

EMU is entirely written in Macro. The choice of language was easy; In spite of protestations to the contrary, MACRO produces tighter, faster code than any other language, however in the transition to Alpha, this may no longer be the case. That and the fact that MACRO was the only language I could write in made the choice obvious. The only external software EMU relies on is VMS (V6.0 or later), DECnet (any phase, any version) and UCX (any version). In many ways, this is the hard way but it also allowed any broken feature to be fixed at home. (which is where most of this was written). It is quite modular - or not, depending on how you measure modularity. Essentially, I tried to make discrete functions as discrete modules and document the interface to each one. Each standalone program is an individual file. Library routines are (generally) functionally grouped together into single files. All files have a standard format and are extensively commented. The block comments are structured such that a simple DCL procedure can extract them and create help files. A quick tour at this level can be obtained by: $HELP/LIBR=EMU5_HLP:EMU_ROUTINES.HLB

4.2 Acknowledgements

EMU was conceived and written by myself, however there were very important contributions by others along the way. Some are as follows: Adrian Challinor for education on CALLS vs. CALLG, Alan Rawson for the DECnet NCB, Eric Von Dijk for a clever use of flags, Simon Stevens for help in documenting the undocumented, The Dutch PTT for knowingly letting me write this while paying me for other work and Bank America for unknowingly supplying the same facility.
Special mention of Ron James who at one point attempted to convert this to C and in the effort discovered a number of very useful techniques. While I couldn't use his code, (can't read C!), I am grateful for the opportunity to steal all his ideas. Finally Keith O'Brien to whom I owe a great debt. From teaching me MACRO (his fault!) to advice on technique, to writing a good third of the previous version, his influence and code remain a large part of the

present system. It is entirely true to say that without Keith's generosity and patience, EMU would not exist beyond a bunch of disconnected code.

4.3 System Tour

Following is a rather wordy explanation of how the system works at code level. This is the $0.25 tour - it is brief and skips sometimes important detail but should be useful as an introduction and supplements reading of the code (or mode correctly the code comments).

4.3.1 Startup and Shutdown

Executing START_EMU.COM gives options to clear the databases and start from fresh and then runs EMU_CONTROL detached. This process reads EMU5_DAT:PRCTBL.DAT, a list of processes it should start. It is a 2 phase startup with a flag being set by RELATER indicating when EMU_CONTROL should continue with phase 2. EMU_CONTROL acquires an exclusive lock (which if it doesn't get, exits without doing anything) and for each process started, creates a termination mailbox. If a process exits with error, EMU_CONTROL restarts it. All activity is written to EMU5_LOG:EMU_CONTROL.LOG which you can only read when the system is not running (one of the less useful design features). Every process tries to acquire the CONTROL lock and if it gets it, runs it's exit procedure and stops. Thus to stop the system gracefully, delete the EMU_CONTROL process (which EMU_STOP.COM does).

4.3.2 LISTEN

The central process at this point is LISTENV5. It:

- creates a section for Ethernet buffers and maps to it.
- creates a table of Ethernet buffer addresses.
- Creates the PSR table by reading EMU5_DAT:PSRTBL.DAT. This file specifies which process (PSR) will receive which frames read from Ethernet.
- Starts the Ethernet controller in promiscuous mode and queues all (32) buffers to it. Each buffer is associated with a flag.
- when an Ethernet read completes, the flag is set and the flag is used as an index to the buffer section to quickly locate which buffer has been written.
- the buffer is scanned and compared with the PSRTBL to determine which, if any PSR should process this buffer. For any PSR this is destined for, a corresponding flag is set in the buffer and at end all corresponding flags are set to wake up the PSR(s).
- at end of each loop, Listen scans for returned buffers and reques them to the Ethernet.
- Any PSR woken up, scans the Ethernet buffer table for the buffer(s) it should process, processes them and sets a flag in the buffer indicating it is finished. When all the flags LISTEN set in it's area are matched by FLAGS set in the PSR area, the buffer is free.

Other detail in this process: LISTEN tracks the number of buffers outstanding to any process and should it reach it's (settable) limit does not pass any more buffers to it and counts a discard.

LISTEN validates each frame for validity by checking:

- Top 3 bytes of source address .eq. 0
- Top 3 bytes of destination address .eq. 0

- Protocol type field (or 802.3 len) .eq. 0

If any of the above are true, an error is counted and the frame discarded (requed without further processing).

There is a hook for special processing. Most frames that are not multicast are discarded but there are some exceptions. The hardcoded filter in use now is to catch and process all ICMP and ARP frames under IP. A hook for OSI error frames is present also but not used.

4.3.3 PSR

Protocol Specific Routines are just that - each one processes a specific protocol - or more accurately a specific frame. Thus PSRMOP processes MOP SYSID broadcasts, PSRLAT processes LAT service announcements etc. The essential purpose for a PSR is to extract the (protocol specific) address in use and any other useful information that may be present and store this in it's database. Each protocol address may appear only once in the database. A mechanism termed BOXID is used to 'tie' multiple addresses on a single device together where this is possible.

4.3.3.1 PSR Tour

PSRs are all implemented using identical mechanisms and if I was a better programmer, they would probably use shared code. As it stands it is more like copied code. Follows here is a list of generic activities any PSR will perform and following that, the specific parts of those PSRs that currently exist.

- Set local symbols THIS_PROCESS, and THIS_PROCESS_FLAG. Allows common code later.
- Request CONTROL_LOCK. When received, it means the control process has died and we join in the mass suicide. Once the exit procedure is enabled, that will run just before death.
- Map to all common sections:
  - Ethernet buffers
  - Control Section
  - PSRTBL
- Map to our DB. Each PSR conceptually owns it's database. All other processes that map to it use a BLOCKING AST and common procedure to ensure that when this PSR wants exclusive access, it can regain it. Thus each PSR uses a unique procedure to map to it's own DB. To ensure no stale data is included, once the PSR maps and loads the latest file in (if present), the file is deleted.
- Enable the exit handler. This procedure (unique to each PSR) analyses the in memory DB, sets the size it should be on next startup and writes it all out to file. The process then dies.
- Map to common event flags used. EMU uses all 64 CEFs available.
- Initialise our area in PSTRTBL. This is where LISTEN stores buffer counts associated with this PSR.
- Send a message to the relater telling it we are alive, well and waiting for work.

- Wait for our flag to be set indicating a message is waiting for us to process. The flag is set by the Listener when it determines a frame in the EBUFFS area should be processed by this PSR.
- Here starts the part that is unique to each PSR though the logic is common:
  - Clear the flag
  - Call LOCATE_PSR. This routine searches the specified PSRDB for the specified address and if not found, creates a new record. If created, it assigns the next (unique) BOXID or optionally assigns one provided in the call, initialises the PSR record header and indicates a new record was created. If the record is found it simply increments the access count and exits normally.
  - If the record is created, the PSR stores parts of the Ethernet frame in the PSRREC and finishes this section. If the record is not created, those parts of the frame that are recorded are checked against the frame and if changed, overwrites the old data and sets a flag indicating this record is updated.
- ensure we are in the relater DB. If we are the relater has set a flag in this record indicating such. If not create a relater frame with our address in it. Some PSRs are able to detect that this frame also belongs in another PSRDB. If so, again the corresponding bits in this record will indicate if it is in that DB and if not, include this information in the relater frame. Some protocols send node names in the frame. The NAMER DB is implemented as a pseudo PSR and as such, if the name has not been sent or it has changed, include this in the relater frame. At end, if a relater frame has been created for any of the above reasons, send it.
- Clear our bit in the Ethernet frame indicating we are finished with the buffer and go back for more. If no more buffers waiting, wait for the next time the flag is set.

4.3.3.2 PSR Database

Each PSR implements a separate database in shared memory only. It is recorded to file on shutdown and reloaded and erased on startup. The PSR conceptually owns the database and is responsible for it's sizing and if necessary, resizing. Any process mapping to a PSR database uses a common routine that ensures the PSR created the database and allows the PSR to regain exclusive access should it be deemed necessary. Details of the contents follow:

All PSRDB databases have a commonly formatted first record:

Table 4–1: PSR First Record

EMU Symbol	Length	Description
COM_DBHDR_L_ENTRIES	0	Number of physical entries
COM_DBHDR_L_RECSIZE	4	Size of each record
COM_DBHDR_L_MAXENTRIES	8	Max number of physical entries
COM_DBHDR_L_FLAG	12	EMUPID of the DB owner

This allows common routines to map and search the DBs.

The remainder of the DB is specific to each PSR. A common header is used for each record:

Table 4–2: PSR Common Record Header

EMU Symbol	Offset	Description
COM_HDR_L_FLAGS	0	PSRID
COM_HDR_L_BOXID	4	Unique device id
COM_HDR_L_PTYBITS	8	Other protocols present
COM_HDR_L_SYSCTL	12	System control flags (SYSDEF)
COM_HDR_Q_LSTHRD	16	Last time heard
COM_HDR_Q_LSTALT	24	Time last alert sent
COM_HDR_Q_FSTHRD	32	Time 1st heard
COM_HDR_L_STATUS	40	Current status
COM_HDR_L_ACNT	44	Count of access this rec
COM_HDR_L_LEN	48	Len of KEY - Protocol addr
COM_HDR_L_HOWSET	52	How this addr found
COM_HDR_Q_LOCKFIELD	56	Count of current accesses
COM_HDR_L_READLOCK	56	Count of read accessors
COM_HDR_L_WRITELOCK	60	Count of write accessors

Notes:

1. PSRID is the ID assigned to the PSR at generation. It corresponds to the flag used to wake the process up (and other uses).
2. BOXID is as described above: an assigned (and changeable) ID that associates records in this and other PSRDBs with a specific device (computer, comms device etc.)
3. PTYBITS. A bit pattern in which a bit is set to indicate the protocols this BOXID is running. The bit set corresponds to the associated PSRID.
4. SYSCTL. System Control Flags. These are:
   - bit 0 ; Rec is deleted
   - bit 1 ; Update this rec in DB. Forces update on next cycle.
   - bit 3 ; Don’t poll. Exactly that. Don’t poll this device. There is no interface to control this as yet.
   - bit 4 ; Pseudo node (alias)
   - bit 5 ; Rec has been added to CNTDB. Hmmm ... this is a facility not yet complete.
   - bit 6 ; Disable CNTPOLL. Same as 5

- bit 7 ; Propagated PSRDB -> EMUDB. If during polling a station does not answer (see EMU_CONFIGMON) this bit (when 0) forces an EMUDB record to be created with the address only. As will be described later, EMUDB is the configuration database built solely by polling. It naturally does not create records when nothing is received via polling but this flag forces the address to appear in the DB.

5. Last heard, Last alert and 1st Heard are all standard VMS binary times showing exactly what they describe.
6. Current status. Not used. Intended to hold the latest state determined for an address (available or not etc) and used to alert when status changes.
7. Count of accesses. Each time a lookup finds this record, this field is incremented. As the search is sequential, it was thought that sorting the records accesses most often to the top of the section would speed up searching. It is probably true but as the search time is insignificant, this remains a solution looking for a problem.
8. Len of Key. Essentially the len of the protocol address that forms the primary key for this record. Some addresses are variable length and this field also allow a common routine to do all the searching.
9. How this address found. Originally a debug filed showing the process that caused this record to be created. As this screwed up other processing, it was changed to reflect the owning PSR and as such is a useless waste of space.
10. Locking. Each .long is a count of readers/writers currently accessing this record. The locking routine is such that is allows shared reads or shared writes but not shared read/write. The mechanism determines the access required and if not currently allowed, waits a short time and then completes (usually).

The remainder of each PSR rec is specific to that PSR. Each is documented in the corresponding PSR source file.

4.3.3.2.1 PSR Mapping

As indicated, mapping to the PSRDBs (in fact, individual memory sections) uses common routines. MAPLVB_DB routine in file MAP_SECTIONS.MAR is used by any process other than the PSR owner to map to a particular PSRDB. The address of a MAP_STRUCTURE is passed to this routine and if successful, returns the section addresses in the structure:

Table 4–3: Map Structure

EMU Symbol	Access	Description
SYS_MAP_Q_ADDR	Write	Returned 1st and last addresses of section
SYS_MAP_L_PID	Read	PID of DB to map to
SYS_MAP_L_KEY	Read	Offset in DB rec where primary key found
SYS_MAP_L_LCKID	Write	VMS assigned Lock id
SYS_MAP_L_SPR	None	Spare/Align

Table 4–3 (Cont.): Map Structure

EMU Symbol	Access	Description
SYS_MAP_L_LOCK	Read	Address of Lock name
SYS_MAP_L_SEC	Read	Address of section name

The routine always sets up a Blocking AST to allow the PSR owner to regain exclusive access if required. If not supplied, UNMAPLVB is set as the routine which runs as a result of receiving the blocking AST.

In addition, the Lock Value Block is written by the PSR owner when it has exclusive access. The value written is the size (in pages) of the section and all other users read this to determine section size to map to. Should resizing occur the PSR:

1. Request the lock in EX mode thus causing the blocking AST to deliver to all other processes mapped.
2. The blocking AST routine runs, withdrawing from the section and reques the lock request in CW mode.
3. The PSR writes out it's memory section, deletes the memory section and calculates a new size. The section is created, the size is written to the LVB and the lock is converted to CW.
4. All other processes the acquire the lock, map the section using the size in the LVB and normal relations resume.

4.3.4 Relater Tour

The relater is process which receives frames from PSRs and constructs what might be called the device level database. This database relates the disparate addresses together and forms a view of the network as boxes. A box in EMU terminology is a device running one or more protocols that is detected by the system.

When a PSR receives a frame from the Listener, one of the main checks it does is to determine if the relater knows about this address. If not a frame is sent and the relater acknowledges receipt by setting the corresponding bit in the PTYBITS field in the PSR record. It also creates (or adds to) a relater DB record.

The theory is simple (and at odds with the code!): Any frame contains the protocol and address it is running on. In some cases, it can be detected that other protocols are also present. For example any Ethernet address beginning with AA-00-04-00 runs DECnet regardless of the protocol the address was received on. Additionally some protocols send the node name in the frame. This information is sent to the relater where it assigns a BOXID, creates the PSR records (if necessary) and stores the related BOXIDs in a single relater record. The Relater DB is implemented as a PSRDB but most of the common fields are not used. The data part is an array in which a BOXID appears in the position corresponding to the protocol. For example, if a MOPSYSID frame containing Ethernet address AA-00-04-00-14-04 is received from the Listener by PSRMOP, the MOP processor recognises this as a MOP address, a DECnet address and an Ethernet address and sends this to the relater along with the assigned BOXID. The MOP address will exist (it has been created by the sending process) but the others may or may not. The relater searches it's own database for this BOXID on this protocol and if not

found, creates a new record. Each other address appearing in the relater frame is searches for in the corresponding DB and if found, the BOXID is added to the relater record at the offset corresponding to the PID of the DB. If not found it is created then added. BOXIDs are allowed to be duplicated within any single PSRdb in the case where a device can run multiple address (IP for example). In the case where disparate addresses are related together and the BOXIDs don't match, one is changed.

The relater then becomes the focus of most user oriented searches. If an IP address is searched for and found, the corresponding relater record is located and what appears on the screen is not simply the IP address but the entire family of protocols running on the device that IP address appears on.

4.3.4.1 Relater Frame

Following is a list and explanation of each field that can appear. This is constructed piece by piece by calling CREATE_RELATER_FRAME repeatedly until all information is included and then calling SEND_RELATER_FRAME to dispatch it.

- BOXID. longword. Sending process's current BOXID
- Sender. longword Sending PSRID
- Message type. longword. Symbol Add, Delete or Start. The targets are either Added to all PSRs or Deleted from all PSRs. In the case of start, only the sender field is used.
- Targets. longword. Number of targets to follow.
- Follows is a list of PSR/Addr targets. It is an unaligned structure:
  - .long process id (SYS_C_PID_xxxx) of target PSR
  - .long len of following addr
  - protocol address of the record in this PSR. This is in the format expected in the receiving PSR. That is:
    - DECnet addresses are 2 bytes
    - IP addresses are 4 bytes
    - NOVELL addresses are 10 bytes
    - OSI address are variable lens
    - and so on.

Maximum relater frame len is 512 bytes. If it is assumed the average len of an address is 8 bytes (actually a bit high) then this leaves room for up to 31 relationships in 1 frame. In the unlikely event this is not enough, the sender can continue in another frame - Each relationship is an independent item.

4.3.5 Configuration Pollers

There are 2 and are distinguished from other possible pollers such as Counter pollers or Test pollers (neither of which exist much beyond a twinkle in the eye). The Configuration pollers are divided into WAN and LAN types largely because WAN traffic tends to be relatively heavy and across slower and more expensive links as compared to LAN types. The LAN types (Ethernet, LAT, MOP, SCS and IPX) are polled for more often than the WAN types (DECnet, IP and OSI). Other than that, they are identical in structure.

Each protocol poller is implemented separately and it is beyond the scope of this section to describe them completely, particularly as this is done in the source code already (EMU_CONFIGMON.MAR). This section is more an overview on approach.

In general, each protocol has a routine that is called giving the address to poll and the address of a buffer to place the received data in. In the more complex protocols (WAN mostly) a function code is also implemented such that a section is polled for on each call. The LAN protocols tend to be quite simple and the data is formatted directly from the received frame and sent to the database. The WAN protocols are much more complicated and intervening routines format the data into standard arrangements before they are written. In all cases, the resulting data is written to EMUDB.

Developers Note: The access routines are functionally standalone and could be incorporated into, and used by other management packages. It is the purpose here to supply these difficult and often tedious routines for other purposes beyond EMU. Some of the modules to look in (should you be so sad as to pursue this) are:

- EMU_CMIP.MAR
- GETSNMP.MAR and GETSNMPROW.MAR
- EMU_MOP.MAR
- GETDN4.MAR
- GETOSI.MAR
- EMU_IPX.MAR

Should you manage to get through this code, the idea will be obvious and all the others should be easily located. It is not intended to take a unix approach here (It was tough for me so it should be tough for you) but simply to avoid repeating what has been said better elsewhere and cluttering up what is intended to be a succinct description.

4.3.6 Main Database - EMUDB

EMUDB is a disk resident, RMS indexed file and contains all the system knows about the network it is sitting on. Each record contains exactly one parameter from exactly one address and is indexed by BOXID. If the system crashes and the PSRs cannot save their data, EMUDB is useless as the main index is lost. There is a possibility of rebuilding the PSRDBs from EMUDB and an effort is underway to complete that but as of now, if the system crashes, delete everything and start over. EMUDB fields are as follows:

- Protocol: The EMU assigned PID this entry is associated with. The list of PIDS assigned is in _EMUSYSDEF.MAR
- BOXID :System generated unique physical device id

- Table :EMU defined table this param is in. Tables are defined in _EMUDBDEF.MAR.
- Instance: The specific instance of an entity. This is a 16 bit CRC of the parameter value.
- Param : The parameter number of this particular instance. If the protocol assigns parameter numbers, these are used. If not, EMU defines them.

The primary key is concatenation of all of the above and dups are not allowed. Any other key or combination may be duplicated A 6th .long is provided in the key field for use as processing flags. This field is defined as part of the key but never used as such.

Following this is the value of the specified parameter. It is kept in 'simplified' protocol specific format. All access to (and complete descriptions of) the DB is through common routines found in EMUDB_ACCESS.MAR

4.3.7 User Interface

This is currently SMG and can be run multiple times on the same node without interference between them. It is implement carefully to preserve the boundary between the system and the interface as it was intended from the outset to replace it. This has not happened for a variety of reasons. Laziness, ineptitude and trying to regain a life are just a few. However, it is ready for some enterprising soul...

4.3.7.1 Query Overview

A query is started by filling out a QUERY_STRUCTURE. This structure is passed to the search routines and is updated on each 'find'. It allows for any single database to be searched either forward or backward, the relater record to be looked up (thus finding all BOX data) and finally formatting and presenting the data as a series of menu selected items. The formatting routines are protocol specific and are often (especially in the ASN.1 encoded protocols - SNMP and CMIP) hugely complex. Full explanations are provided in the sources - see EMU_UILIB.MAR.

4.3.7.2 Report Overview

A report is generated from a report parameter file. This file is created using the menus which select the parameters to include in the report. The report is generated in spreadsheet format, that is it is easily imported to Excel or other PeeWee program. See EMU_RPT.MAR for full details. EMU_REPORTING.MAR provides a DCL interface to the report subsystem.

4.3.8 Alert Mechanism

A basic alert mechanism (that doesn't work very well) is in EMU_ALTLIB.MAR and is executed with ALERT.MAR, a standalone process that fields alerts, stores and displays them. Essentially, an alert generator acquires an alert packet (memory section), fills in the alert parameters and signals ALERT to process it. The ALERT process then stores the raw frame on disk, then translates and displays the alert if an alert display process is running. A number of unused mechanisms are in play here and need some explanation:
What is present is the lower layer of a 2 layer process. What is intended is a smarter alert than is present on most systems however this part has never been completed. The raw alert, as stored and displayed, is intended to be further processed to allow a much wider context to be alerted. The concept of 'related' and 'recent' events is built in but not implemented in this version. The definition of recent and related are in the architecture section of this manual. Currently, the usefulness of this process is it's ability to review all events recorded. As a monitor, it is limited (generous!) and some work is required to remove restrictions on parameter specification.

4.4 Summary

EMU was originally conceived as a tool to assist me in my day to day job as a network manager and to that end it has been very successful. As it developed (and became an end in itself) it became a commercial endeavour and managed some limited success here also. Along the way it also served as a teaching tool and hobby. In this it was successful. Finally it became an imprisonment, an activity that demanded more time and effort than anyone could supply. This aspect of it was entirely successful. Now it is a gift - Anyone who uses, develops it or even has an interest is welcome to do as they see fit. I would be pleased to hear from you and if time permits, (I have a life now!) assist in your endeavour. I can be contacted at system@ccci4.demon.co.uk. The code is well documented and as readable as Macro code can be. It is also modular enough to take functions out and reuse them elsewhere. EMU was very much a reaction to the poor quality of network management packages available - a situation that IMHO has not changed. Without prejudice to any group, I would suggest that not enough network people can program or even understand what can be done with a good system and programmers in general do not understand what network managers need. That is the gap I am attempting to bridge however, it is a bridge too far. The multiple skills required and the sheer size is beyond a single person working part time.

Index

A

Alert Format, 3–10
Alert Formatting and Transmission, 3–13
Alert generation, 3–12
Alert log files, 3–14
Alert Parameters and translation, 3–13
Alert processing, 3–12
Alert Subsystem, 3–10
Autogen, 1–2

B

BOXID, 2–3
BOXID Modification, 3–8

C

CNTRPOLL.DAT, 3–15
CNTRPRC.DAT, 3–15
CNTRPRC Record, 3–16
Core System, 2–1
Counter Processing files, 3–15
Counter Processing System, 2–5
Customer Network - Effect on, 1–2

D

Database Access, 2–5
Data typing, 3–13

E

EMU network, 2–3
EMU_CNTPOLL.EXE, 3–15
EMU_CNTRPC, 3–15

F

Features, **1–2**
Future incarnation, 1–1

I

Identity, 2–3

L

Listener, 2–1, 3–1

N

Network Views, 2–2
non-goals, 1–3

P

Physical Topology, 3–11
Protocol Address, 2–3
Protocol type formats, 3–3
PSR, 2–1, 3–3
PSR Database Overview, 2–4
PSR - General processing, 3–6
PSR interface, 2–2
PSR Record Header, 2–4, 3–4
PSR Records, 3–4
PSR Record Structure, 2–4

R

Relater, 3–7
Relater Database, 3–7
Relater Frame, 3–8

S

Segmentation, 3–11
SNMP, 1–3
System Control bits, 3–6
System extensions, 1–2
System Functions, 1–1
System goals, 1–1

T

TCPIP Services, 1–2

U

UCX, 1–2