White Paper:

Using AutoTMF, TMF and RDF to Enable Disaster Recovery for BASE24 systems

Philip J Nye

Table of Contents

Introduction

It is now possible to provide improved performance and Disaster Recovery for BASE24 applications using the standard HP NonStop products:

- NonStop Transaction Management Facility (TMF)
- NonStop Remote Database Facility (RDF)
- NonStop AutoTMF

NonStop RDF maintains a real-time, logical copy of an application database on one or more backup systems. In the event of a failure of the primary system, the application can be started on the backup system and continue to access up-to-date data. RDF achieves this by sending audit trail data, generated by the HP NonStop TMF product, across a NonStop Expand network to the backup system(s).

In addition to providing a guaranteed audit log to enable data replication, TMF in many cases can also dramatically improve application performance. To learn more about the performance of TMF with BASE24, please refer to Section 6.

An application has to be coded to support various TMF procedure calls in order to access audited files.

Prior to Release 6.0, BASE24-atm and BASE24-pos did not support audited files. At Release 6 a number of audited files were introduced, but these were used primarily for message routing and today most of the core files, such as the Transaction Log and Terminal Definition file, still remain unaudited.

Historically, the only way for applications like BASE24-atm and BASE24-pos to have supported TMF would have been through custom modifications.

However, it is now possible to automatically add support for TMF to BASE24 applications using the NonStop AutoTMF product and this means that BASE24 systems can now take advantage of NonStop RDF to enable Disaster Recovery.

Cardlink have successfully helped a number of BASE24 customers around the world to implement NonStop AutoTMF. We have some 20 years experience of running BASE24 systems using TMF and RDF.

This document seeks to pass on to potential users some of the lessons learnt from these experiences.

The remainder of the document is structured as follows:

Section 1: Provides a basic introduction to the architecture of a BASE24 system: we concentrate on BASE24-pos, but the principles apply to other members of the BASE24 family.

Section 2: Examines how some typical transactions are processed in a BASE24-pos system. Understanding message flows, database I/Os and inter-process calls is crucial to a successful implementation of NonStop AutoTMF

Section 3: Briefly introduces TMF and NonStop AutoTMF. It is assumed that Users will be familiar with TMF. We show how setting various NonStop AutoTMF commands can change the way TMF transactions are managed.

Section 4: Provides tips as to how NonStop AutoTMF may be implemented in a BASE24 environment. We show how the Nonstop AutoTMF trace facility may be used to help Users develop a detailed understanding of all the file I/O calls made by their BASE24 applications.

Section 5: Uses an example TMF and RDF configuration to illustrate how a flexible BASE24 Disaster Recovery environment can be built.

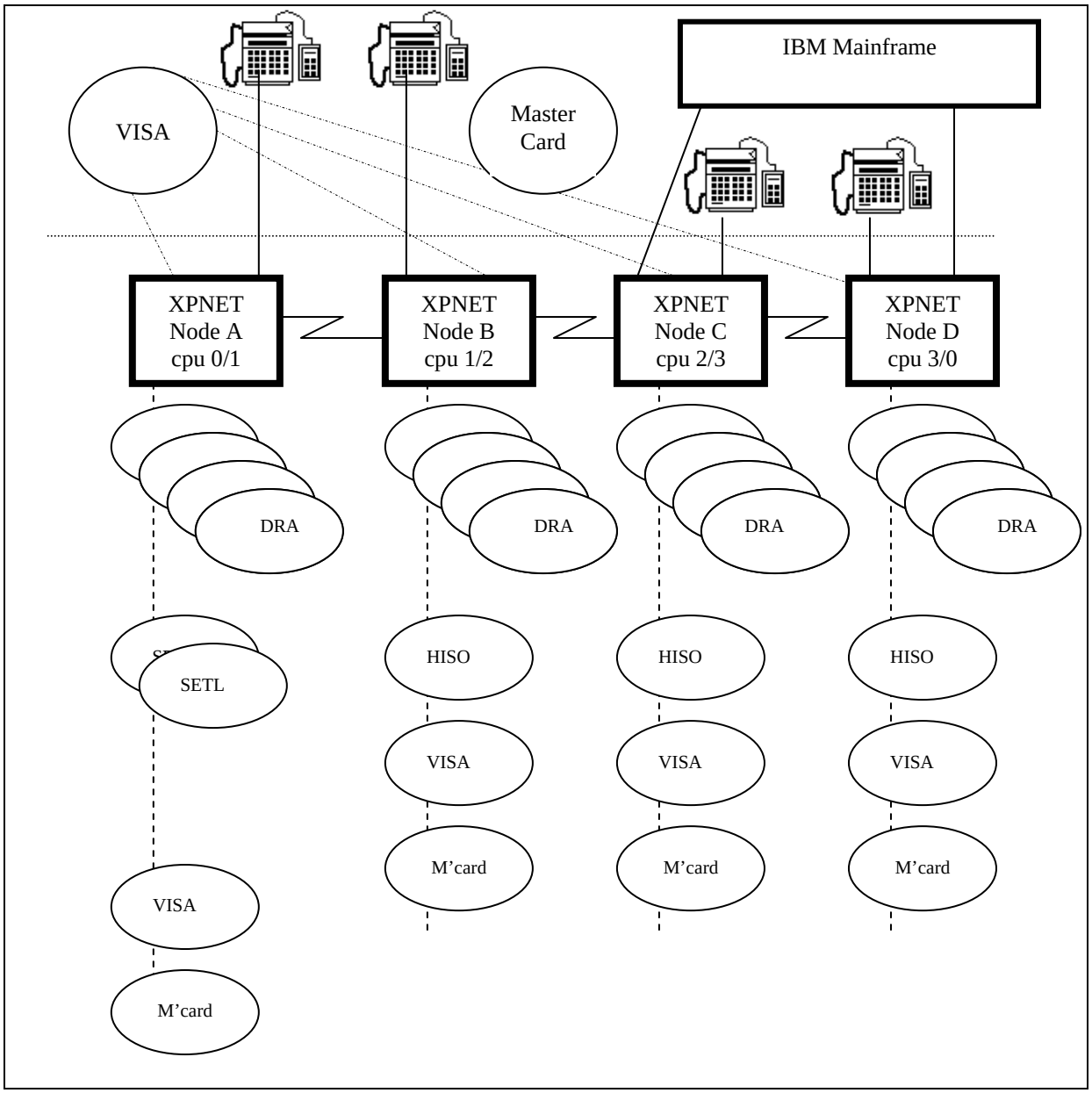
Section 6: Discusses some of the performance benefits that accrue to BASE24 applications using TMF.

1 BASE24-pos: Architecture and Concepts.

In this section we briefly examine the architecture of a BASE24-pos system and discuss some of the important BASE24 concepts. In particular, we discuss:

- BASE24 XPNET Objects: Lines, Stations, Processes and Link Processes
- BASE24 Resource Nodes
- The concept of a BASE4 Logical Network

The BASE24-literate User may decide to skip this section and move to Section 3.



Fig, 1 – BASE24 Architecture

In Fig. 1, everything above the dotted line represents something external to the NonStop system to which BASE24 'connects'. In this example, BASE24 connects to:

- various Point-of-Sale (POS) terminals (how many and which particular type of terminal does not matter for now).
- the Visa online authorizations network
- the MasterCard online authorizations network
- An IBM mainframe.

BASE24 Concepts

In our example we have chosen to configure 4 BASE24 Resource Nodes. A ***Resource Node*** *is the collection of resources controlled by a single XPNET process*. Fig. 1 shows 4 BASE24 XPNET processes (actually the primary processes), each one running in a different cpu, with the backup XPNET process running in a separate backup cpu. We call the Resource Nodes: Node A, Node B, Node C and Node D.

XPNET is a critical component of any BASE24 system. It is responsible for managing connectivity to end devices, routing messages from devices to the appropriate device handler and for managing a variety of functions such as message queuing, tracing etc.

Each XPNET Process, each Resource Node, can communicate to all the other Resource Nodes via a ***Link Process***. A separate link process exists for each connection. In Fig 1, Resource Node A could be configured with 3 Link Processes: one for Node B, one for Node C and one for Node D. A Link Process isn't a physical process: it is an internal queue managed by the XPNET process.

Each Resource Node manages communications to external entities. In our example above, Node D connects to Visa, MasterCard, POS terminals and the IBM mainframe. XPNET defines 3 objects to establish a communications connection to an external entity: *Device*, *Station* and *Line*.

A ***Station*** is a named-endpoint which can send and/or receive messages. When connecting to an X.25 network, for example, a station can be thought of as an extension of an X.25 Logical Channel. When connecting to an SNA network, a station represents a single SNA LU. Stations are defined within XPNET, but may also be used in various BASE24 application configuration files.

A ***Line*** is the link between an I/O process, one or more stations and a physical communications wire. A line is also a named entity and it is configured to access a particular Guardian subdevice or port. When a message is sent to an XPNET station, XPNET will dequeue the message from the station and send it to the appropriate Guardian I/O process.

Each Resource Node controls a number of *BASE24 Satellite* ***Processes***. In our example, each Resource Node contains 4 POS Device Handler/Router/Auth processes. (In section 2 we will examine exactly what a DRA does). XPNET allows multiple processes of a particular type to be configured and provides functionality to balance traffic across all processes. Processes do not have to be in the same XPNET: an XPNET process can send messages to any process in any Resource node via the Link Processes.

Multiple Resource Nodes can be configured together to form a ***Base24 Logical Network.*** A BASE24 Logical Network accesses a single database which is shared across all Resource Nodes.

2 BASE24 Transaction Processing and Message Flows

A BASE24 application consists of different 'processes', each performing various functions. Processes can interact: A **Requester** process can send a message to a **Server** process. The Server process performs some work and may reply back to the originating Requester. The Server process may itself become a Requester: it can send a message to a different Server.

To process a transaction, BASE24 generates a flow of messages between XPNET, Satellite and other processes.

Understanding the message flows, database accesses and inter-process calls made by a BASE24 system is crucial to the successful implementation of NonStop AutoTMF.

In this section we describe how BASE24 processes 3 different types of transaction. We look at:

- A transaction initiated at a Point-of-Sale terminal (using an APACS30 protocol) managed by BASE24 and authorized by an Issuer system directly connected to BASE24. BASE24 is configured to use Online/Offline authorization with a Negative file.

- A transaction initiated at a Point-of-Sale terminal (using an APACS30 protocol) managed by BASE24 and authorized by an Issuer via a Card Scheme network such as Visa, MasterCard, Amex etc.

- A transaction initiated at a Point-of-Sale terminal managed by another acquirer and sent into BASE24 via a scheme network (such as Visa, MasterCard, Amex etc.). BASE24 is configured to authorize the transaction using offline CAF/PBF.

We show the message flows between the various processes and see where files are accessed.

In a document of this type it is not possible to fully detail every step in the process. Rather, the intention is to provide the reader with a good level of understanding as to what takes place in BASE24.

2.1 *BASE24-managed Terminal, Directly-connected Issuer.*

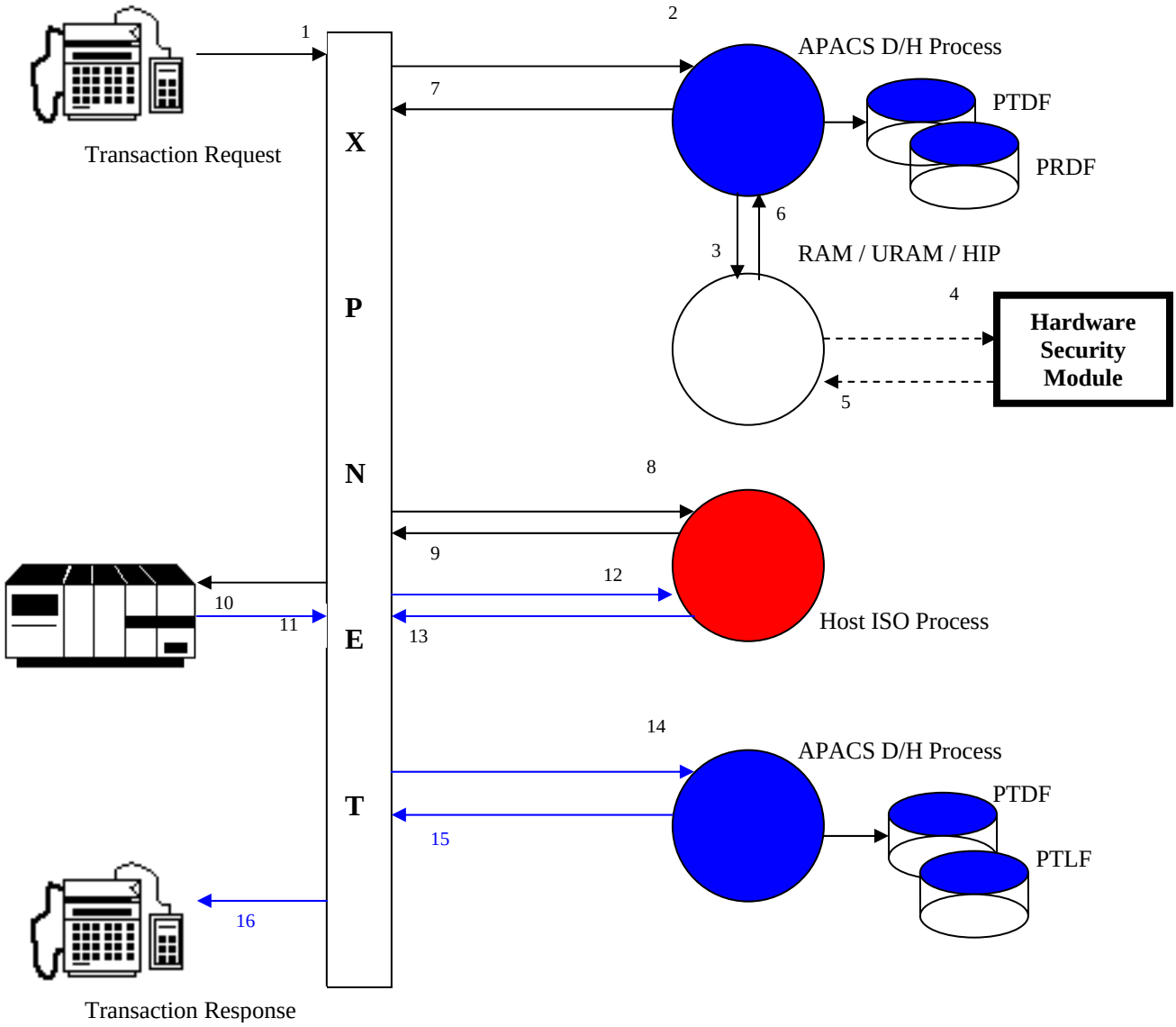


Fig. 2 – BASE24 Message Flows

1. The Point-of-Sale terminal issues a transaction request. In this example, the terminal formats a message using an APACS30 protocol.

   I/O processes running on the NonStop server pass the message to XPNET. (Note that before this can take place the device may first have to establish a connection into the system: we do not show the message exchanges for this here).

2. XPNET determines where to route the transaction: in this case an APACS Device Handler/Router/Auth process. XPNET attaches the XPNET System Header to the front of the message from the device and sends the new message to an APACSDH process.

The APACS Device Handler process checks that the transaction request message is well-formatted. It extracts the Terminal Id and Retailer Id fields from the message and reads the appropriate records from the PTDF (terminal) and PRDF files.

The APACS Device Handler then builds an internal POS 0200 Standard request message and passes control over to the Router/Authorisation component.

In our example, the Router/Authorisation component determines that is has to perform some cryptography, for example that we have to verify the Card Verification Value store in Track2.

3. The Router/Authorisation component formats a message and sends this to an interface process responsible for managing access to some Hardware Security Device (for example, a Thales HIP process, or ACI's RAM or URAM processes).

4. The HSM Interface process sends an appropriate message to the physical Hardware Security Module.

5. The Hardware Security Module performs its appropriate cryptographic function and returns a response to the HSM Interface process.

6. The HSM Interface process replies back to the calling APACSDH process.

7. The APACSDH process next determines where this transaction is to be authorised: in this case a mainframe application. The APACSDH process returns the POS 0200 Request message to XPNET.

8. XPNET sends the POS 0200 request message to a BASE24 ISO process.

9. The ISO process converts the POS 0200 request message into an appropriately formatted ISO 8583 message (in this case an ISO 0200 Request Message). The ISO process finds a BASE24 station over which the mainframe can be accessed and sends the ISO 0200 Request to XPNET.

10. XPNET sends the ISO 0200 Request out over the appropriate station to the application running on the mainframe.

11. In this example, the Mainframe application approves the transaction and sends an ISO8583-formatted 0210 response message to XPNET.

12. XPNET attaches an XPNET System header to the response message from the mainframe and sends this new message to the ISO Host process.

13. The ISO Host process converts the ISO8583 0210 response into a BASE24 POS standard internal 0210 response message and sends this back into XPNET

14. XPNET forwards the POS standard 0210 response message to the originating APACSDH process.

15. The APACSDH process receives the 0210 response message**.** It reads and updates the appropriate record from the PTDF (terminal) file and writes a record to the POS Transaction Log File (PTLF).

    It formats an APACS30 response message which is sent to XPNET,

16. XPNET transmits the response message to the originating POS device.

2.2 *BASE24-managed Terminal, Authorised by VISA Issuer*

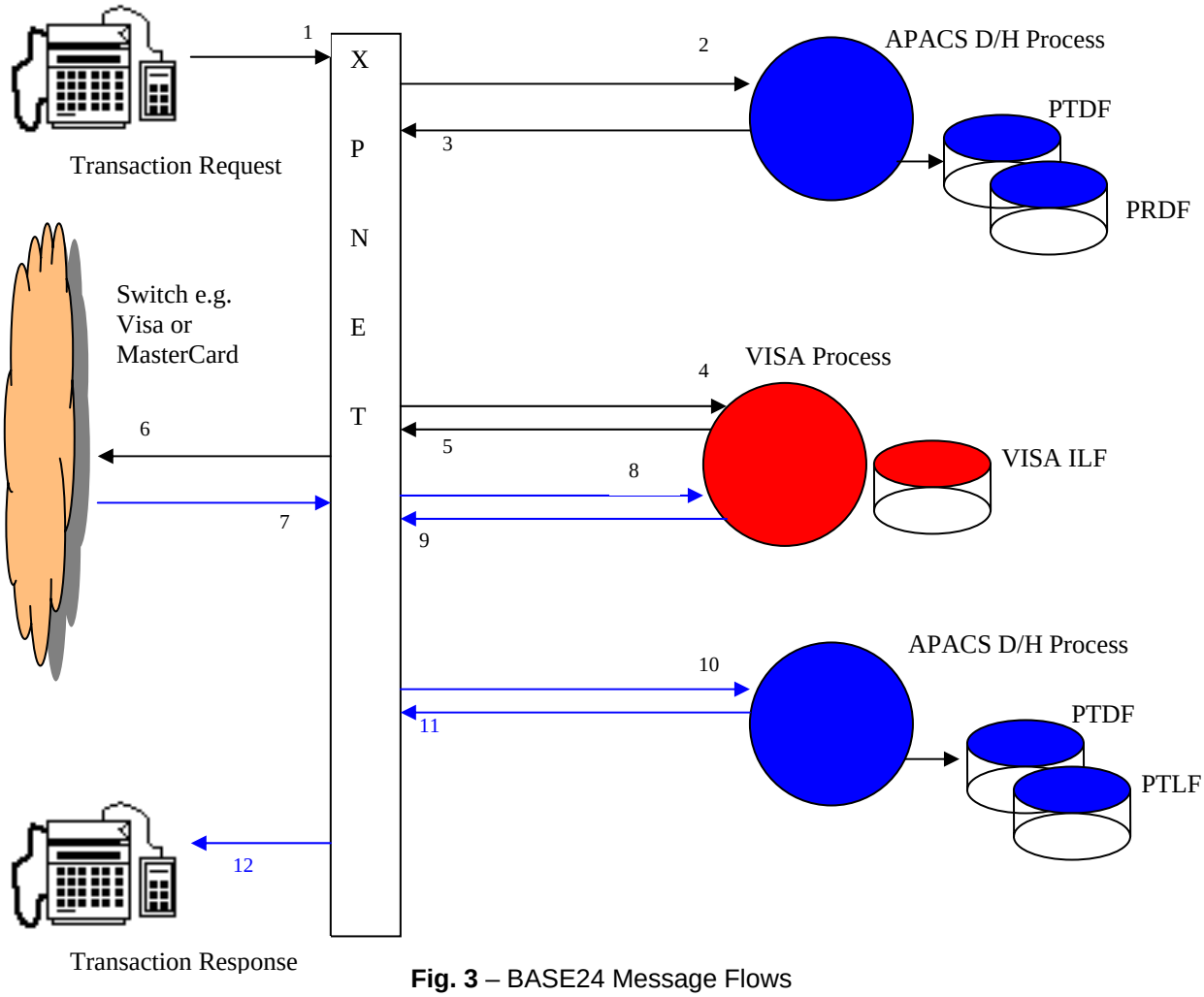


Fig. 3 – BASE24 Message Flows

1. The Point-of-Sale terminal issues a transaction request. In this example, the terminal formats a message using an APACS30 protocol.

   I/O processes running on the NonStop server pass the message to XPNET. (Note that before this can take place the device may first have to establish a connection into the system: we do not show the message exchanges for this here).

2. XPNET determines where to route the transaction: in this case an APACS Device Handler/Router/Auth process. XPNET attaches the XPNET System Header to the front of the message from the device and sends the new message to an APACSDH process.

3. The APACS Device Handler process checks that the transaction request message is well-formatted. It extracts the Terminal Id and Retailer Id fields from the message and reads the appropriate records from the PTDF (terminal) and PRDF files.

   The APACS Device Handler then builds an internal POS 0200 Standard request message and passes control over to the Router/Authorisation component. The APACSDH process next determines where this transaction is to be authorised: in this case the VISA online authorisation network. The APACSDH process returns the POS 0200 Request message to XPNET.

4. XPNET sends the POS 0200 request message to a BASE24 VISA process.

5. The VISA process converts the POS 0200 Request message into an appropriately formatted Visanet Request message (in this case a BASEI 0100 Request Message). The VISA process finds a BASE24 station over which the VISA online authorisation network can be accessed and sends the BASEI 0100 Request to XPNET.

6. XPNET sends the BASEI 0100 Request out over the appropriate station to the Visa network.

7. VISA forwards the request to the appropriate Issuer. In this case the Issuer approves the transaction and returns a BASEI 0110 response message through the VISA network and back into XPNET.

8. XPNET attaches an XPNET System header to the response message from the Visa network and sends this new message to the VISA process.

9. The VISA process converts the BASEI 0110 response message into a BASE24 POS standard internal 0210 response message. It writes a record to the BASE24 VISA Interchange Log File and then sends the 0210 response back into XPNET

10. XPNET forwards the POS standard 0210 response message to the originating APACSDH process.

11. The APACSDH process receives the 0210 response message. It reads and updates the appropriate record from the PTDF (terminal) file and writes a record to the POS Transaction Log File (PTLF).

    It formats an APACS30 response message which is sent to XPNET,

12. XPNET transmits the response message to the originating POS device.

2.3 *VISA Acquired, Authorised by BASE24 (CAF/PBF)*

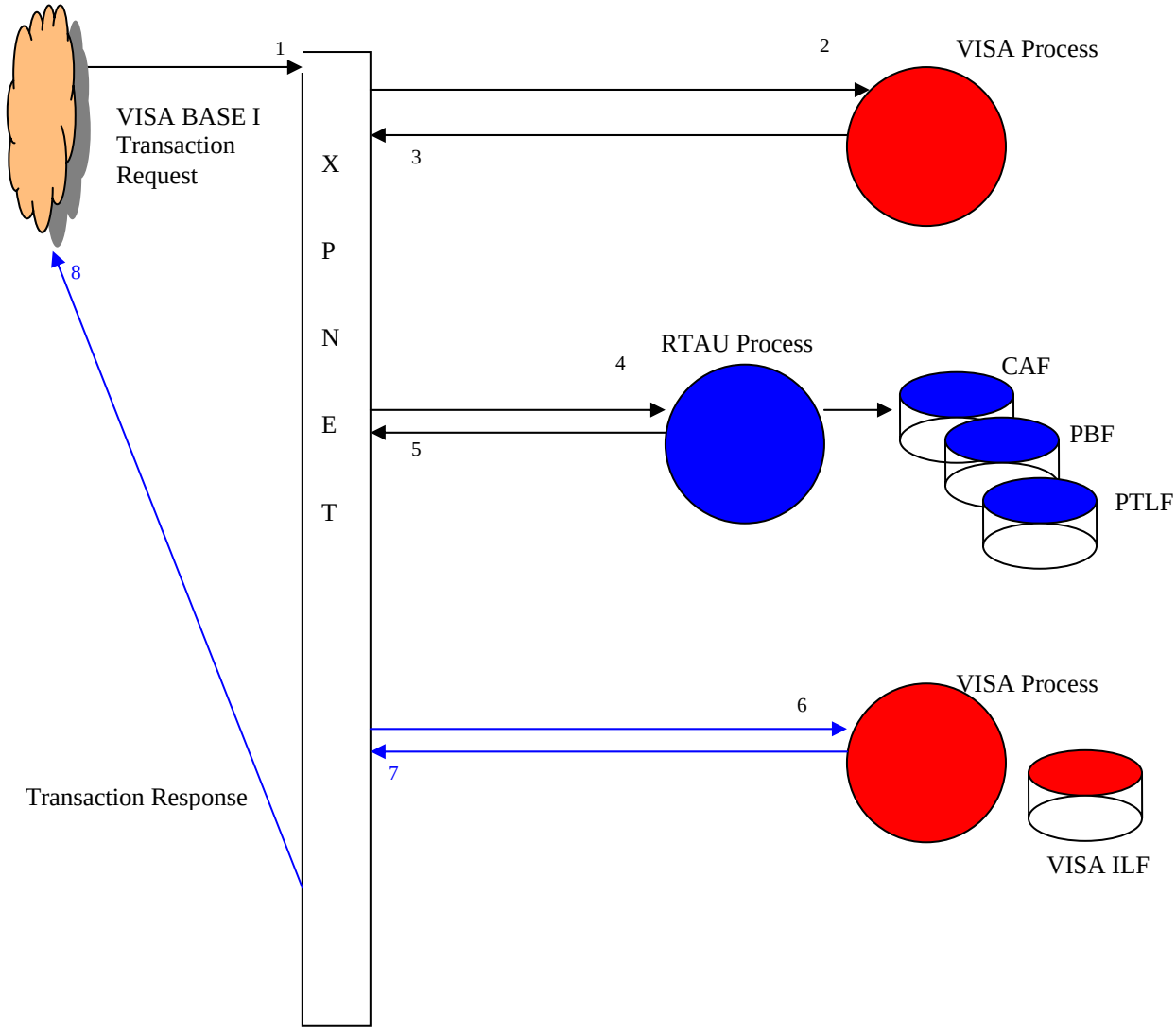


Fig. 4 – BASE24 Message Flows

1. The VISA online authorisation network receives a BASEI 0100 authorisation request message from an acquirer. It sends this to the XPNET process.

2. XPNET determines where to route the transaction: in this case a BASE24 Visa process.

3. The Visa process converts the BASEI 0100 request message into a BASE24 POS Internal 0200 request message. It identifies the name of a BASE24 Router/Authorisation process and forwards the request to XPNET.

4. XPNET sends the POS 0200 request message to a Router/Authorisation process.

5. The Router/Authorisation process identifies how this transaction should be authorised. In this case BASE24 has been configured to authorise offline using the BASE24 CAF (Cardholder Account) and PBF (Positive Balance) files. Router/Authorisation reads the CAF, to retrieve cardholder account information, and the PBF, to determine whether the cardholder has sufficient funds.

   BASE24 determines that this transaction should be approved. The PBF is updated with new balance information and a transaction is written to the BASE24 POS Transaction Log File.

   Router/Authorisation formats a BASE24 POS 0210 response message and sends this to XPNET.

6. XPNET sends the 0210 Response message to the BASE24 Visa Process.

7. The VISA process converts the 0210 response message into a BASE1 0110 response message. It writes a record to the BASE24 VISA Interchange Log File and then sends the 0210 response back into XPNET

8. XPNET forwards the BASE1 0110 response message to the VISA Network.

3 Introduction to TMF and AutoTMF

TMF is HP's Transaction Management product. It provides facilities which allow an application to protect database updates.

We briefly examine some of the important TMF concepts.

3.1 *NonStop TMF*

Invoking TMF

An application invokes the TMF subsystem by calling various TMF procedure calls. Additionally, any file required to be 'TMF-protected' must have its AUDIT file attribute configured.

Fig. 2 below compares an application writing to an unaudited (non-TMFed) and audited (TMF-protected) file. Note that most files used by the 'classic' BASE24-pos and BASE24-atm applications are unaudited.

	File with AUDIT not set	**File with AUDIT set**
Example A. Add a record e.g. Transaction Log Record	…Format record CALL WRITE	…Format record CALL BEGINTRANSACTION CALL WRITE CALL ENDTRANSACTION
Example B. Update a record e.g. POS Terminal	CALL READUPDATELOCK …modify record CALL WRITEUPDATEUNLOCK	CALL BEGINTRANSACTION CALL READUPDATELOCK …modify record CALL WRITEUPDATEUNLOCK CALL ENDTRANSACTION

Fig. 2 - Using TMF procedure calls to protect database updates

Fig. 2 shows some simple code fragments to illustrate the concepts of record-locking and using BEGINTRANSACTION and ENDTRANSACTION to protect database updates. It does not show some of the more complicated issues concerning the use of TMF, such as suspending or resuming TMF transactions, the use of the pseudo TFILE to manage concurrent TMF transactions etc.

TMF Audittrail

The TMF subsystem maintains an audittrail of protected transaction details. When an application calls ENDTRANSACTION, TMF writes details of the database changes made to a TMF Audittrail.

Once details have been added to the Audittrail, TMF informs the disc processes controlling access to those files that have been updated that the audittrail has been successfully updated. The disc processes can then release any record locks. This is illustrated in Fig.3.

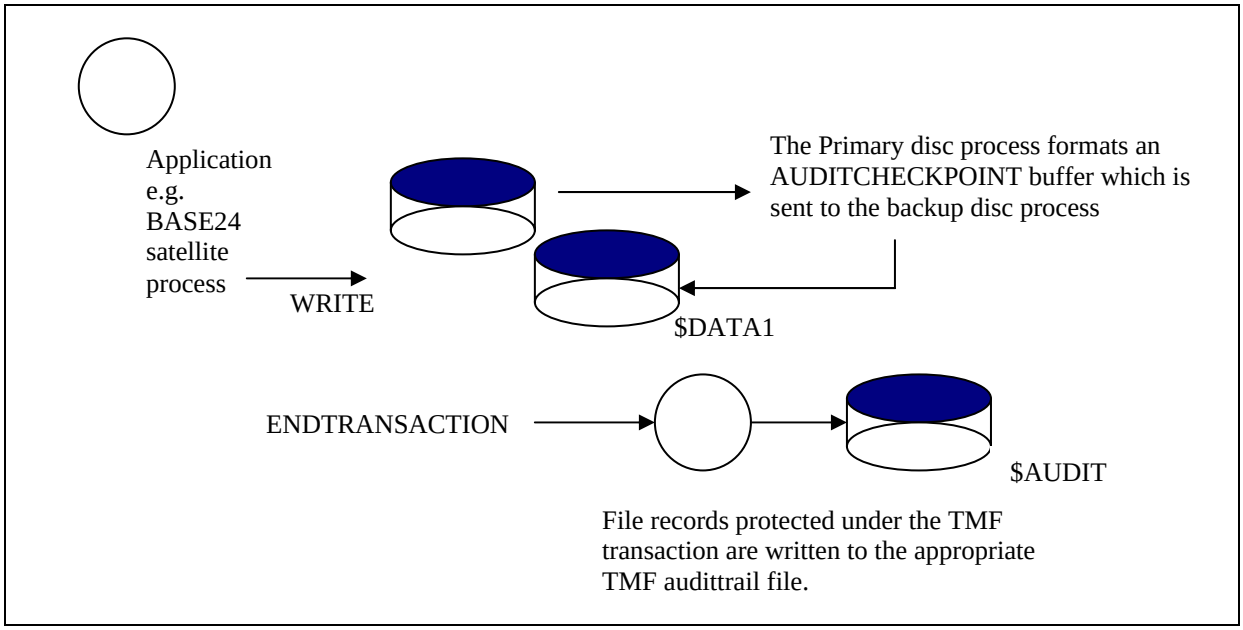


Fig. 3- TMF and its Audittrail

FUP AUDITCOMPRESS

When a record is modified and updated, TMF will write details of the record as it existed before the update and after the update to the TMF Audittrail. It may be the case that an update only changes a few bytes in a record. To minimise the amount of data written to the audittrail, the FUP AUDITCOMPRESS parameter can be used so that only those parts of the record which have been updated are written to the audittrail.

Transaction and Database Integrity

If something should happen before an application calls ENDTRANSACTION, for example:

- if the application abends,
- if the cpu running the application goes down,
- if the application itself finds some logic error and calls ABORTTRANSACTION

then the TMF subsystem will return the state of all files to the point before BEGINTRANSACTION was called. It does this by reading backwards through the TMF audittrail and 'undoing' all database updates made during the TMF transaction.

TMF 'Box-carring': Better performance at higher loads

The TMF subsystem employs a technique known as 'box-carring'. When the TMP process receives an ENDTRANSACTION call from an application, it does not immediately write data to the TMF audittrail on disc. Instead, data is written to a large, 56K buffer. The TMP process then sets a small delay, by default 0.1 second, in order to 'catch' other ENDTRANSACTION calls made by any other process. These transactions are added to the Audittrail buffer.

This small delay allows TMF to write large amounts of data to disc in a single write, significantly improving the performance of the TMF subsystem.

Data Files: Better buffer performance.

We have seen how the TMF subsystem keeps a full history of database updates in its audittrails. The fact that data has been safely written to disc means that there is no immediate need to write data to the discs holding the database files protected by TMF.

Data files protected under TMF can therefore be configured to keep data in disc cache.

The TMF subsystem ensures that data held in disc cache eventually gets written to disc. Every 5 minutes each disc process will build a list of all 'dirty' cache blocks. A dirty cache block is a block in the disc cache which contains details of either a new, deleted or modified record (plus some other disc subsystem-specific functions). 5 minutes later, the disc process will write these dirty cache blocks to disc and will then mark each block as clean: the data still remains in the cache block at this time.

Using TMF may provide substantial performance benefits to BASE24 applications: these are discussed further in section 6.

3.2 *NonStop AutoTMF*

NonStop AutoTMF automatically adds support for TMF to an application which is not 'TMF aware'.

AutoTMF does this by intercepting all file I/O calls made by an application.

This is achieved via the services of the AutoTMF run-time library: User applications already using a User Library need to use BIND to add the AutoTMF Run-time library into their own User Library.

AutoTMF 'knows' whether a particular file is to be audited and automatically adds the necessary procedure calls to allow TMF to function 'behind the scenes'. In most cases, the BASE24 User need only turn AUDIT on those previously un-audited files . However, there are some cases where a BASE24 User will need to take advantage of some of the options provided by AutoTMF to control the way TMF transactions are managed.

Managing TMF transactions using AutoTMF

AutoTMF provides, via its Command Interpreter ESCORT, various command options to control how TMF transactions are managed. These options affect factors such as:

- the number of TMF transactions generated by AutoTMF

- the lifetime of a TMF transaction
- the time records are held locked.

We show the impact of some of these AutoTMF options by using an example relevant to the BASE24 environment.

In Fig.4, XPNET sends a transaction request to a Device Handler/Router/Auth ("DRA") process. The DRA will access 3 files: the PTDF (POS Terminal Data File), the PRDF (POS Retailer File) and the PTLF (POS Transaction Log File). The DRA process will also make an inter-process call to a "URAM" process which controls access to a physical Hardware Security Module.

In our example:

1. XPNET receives a transaction request from a POS device and forwards this to the appropriate DRA process.

2. The DRA reads with lock (i.e. READLOCK) the PTDF file (to obtain the Terminal record) and then reads the PRDF file (to obtain the Retailer record). It then sends a message (by WRITEREAD) to the URAM process in order to perform some cryptographic function, for example validate an EMV ARQC. A response is returned by URAM to the DRA.

3. The DRA updates the PTDF record via a call to WRITEUPDATEUNLOCK and sends an 0200 PSTM message to XPNET.

4. Moments later, XPNET sends an 0210 PSTM message to the DRA. The DRA reads, with lock, the PTDF file. It updates information in the Terminal record and then issues a WRITEUPDATEUNLOCK to the PTDF record. It writes a transaction log record to the PTLF.

5. The DRA then formats a response message back to XPNET (which is forwarded to the terminal).

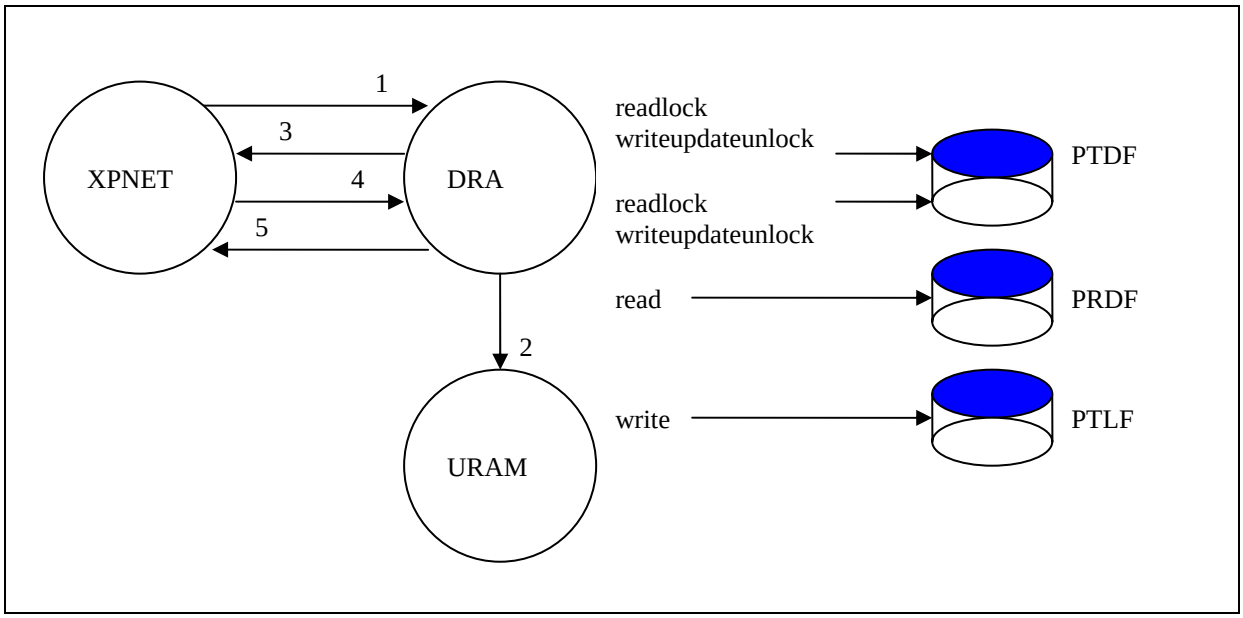


Fig. 4 – Example Message Flow (before AutoTMF)

We now show how, by setting different AutoTMF command options, we can affect the manner in which AutoTMF manages TMF transactions. All three files in this example, the PRDF, PTDF and PTDF are first audited.

(Note: It is assumed that the reader has already read the AutoTMF User Guide and is familiar with the command options).

3.2.1 Example 1: AutoTMF Default Options

We first consider the case where no explicit configuration of the AutoTMF Command options is performed i.e. we simply use the default AutoTMF **GLOBAL** values:

- **ATMF ON**
- **ATMFABENDNOAUDIT OFF**
- **ATMFAUTOCOMMIT 0**
- **ATMFCOMMONTX ON**
- **ATMFISOLATION WEAK**
- **ATMFMAXTIME 16**
- **ATMFMAXUPDATE 32**
- **ATMFNOWAIT OFF**
- **ATMFREADTHRULOCKS ON**
- **ATMFSEPARATETX OFF**
- **ATMFSKIPNULLRECS OFF**
- **ATMFTXHOLDOFF 0**

- At step (2), AutoTMF intercepts the DRA call to READLOCK the PTDF record and automatically starts a **common** TMF transaction.

- At step (3), AutoTMF intercepts the call to REPLY (which is the mechanism by which the DRA sends a message to XPNET).

  At this point the DRA has already issued a call to WRITEUPDATEUNLOCK, so AutoTMF knows that the record lock has been removed and because **WEAK ISOLATION** is in effect, AutoTMF calls ENDTRANSACTION to commit the TMF transaction.

  The **Global ATMFNOWAIT** parameter is set to **OFF**, so AutoTMF waits for the call to ENDTRANSACTION to complete.

- At step (4), AutoTMF intercepts the second DRA call to READLOCK the PTDF record and automatically starts another **common** TMF transaction. The DRA issues a second WRITEUPDATEUNLOCK to again update the PTDF record.

  The DRA then writes the PTLF record. At this point AutoTMF knows that a **common** transaction has already been started and so is able to perform this PTLF write using the single common transaction.

- At step (5), AutoTMF intercepts the call to REPLY. AutoTMF recognises that there is an active TMF transaction waiting to be committed. There are no record locks in place for file activity associated with this transaction and so AutoTMF calls ENDTRANSACTION.

In this example we see that to process a single business transaction AutoTMF generates two TMF transactions.

3.2.2 Example 2: AutoTMF – SEPARATETX(1)

In addition to the same AutoTMF GLOBAL parameters as in the previous example, SEPARATETX is configured against the PTLF file. This is achieved via the ADD ATMFF command:

ADD ATMFF $volume.subvol.PO*, SEPARATETX, MAXUPDATES 1

- At step (2), AutoTMF intercepts the DRA call to READLOCK the PTDF record and automatically starts a **common** TMF transaction.

- At step (3), AutoTMF intercepts the call to REPLY

  At this point the DRA has already issued a call to WRITEUPDATEUNLOCK, so AutoTMF knows that the record lock has been removed and because **WEAK ISOLATION** is in effect, AutoTMF calls ENDTRANSACTION to commit the TMF transaction.

- At step (4), AutoTMF intercepts the second DRA call to READLOCK the PTDF record and automatically starts another **common** TMF transaction. The DRA updates the PTDF record.

  However, because SEPARATETX has been configured against the PTLF file, AutoTMF doesn't use the common TMF transaction to protect the write to the PTLF. Instead, it suspends the common TMF transaction and begins a **separate** TMF transaction. Immediately after the write completes, because the MAXUPDATES parameter has been set to 1, AutoTMF calls ENDTRANSACTION to end the **separate** transaction.

- At step (5), AutoTMF intercepts the call to REPLY (which is the mechanism by which the DRA sends a message to XPNET). At this point the DRA has already issued a second call to WRITEUPDATEUNLOCK, so AutoTMF knows that all record locks associated with the TMF transaction have been removed. AutoTMF calls ENDTRANSACTION to commit the **common** TMF transaction.

In this example we see that to process a single business transaction AutoTMF generates three TMF transactions.

3.2.3 Example 3: AutoTMF – SEPARATETX(2)

In addition to the same AutoTMF GLOBAL parameters as in the previous example, SEPARATETX is configured against the PTLF file AND the PTDF file. This is achieved via the ADD ATMFF command:

ADD ATMFF $volume.subvol.PTDF, SEPARATETX, MAXUPDATES 1

- At step (2), AutoTMF intercepts the DRA call to READLOCK the PTDF record and automatically starts a **separate** TMF transaction.

  The DRA issues a WRITEUPDATEUNLOCK call to update the PTDF record. AutoTMF intercepts the call and, because **MAXUPDATES 1** is in effect, it calls ENDTRANSACTION to commit the TMF transaction.

- At step (3), AutoTMF intercepts the call to REPLY. At this point there are no active TMF transactions waiting to be committed.

- At step (4), AutoTMF intercepts the second DRA call to READLOCK the PTDF record and automatically starts another **separate** TMF transaction.

  The DRA issues a WRITEUPDATEUNLOCK call to update the PTDF record. AutoTMF intercepts the call and, because **MAXUPDATES 1** is in effect, calls ENDTRANSACTION to commit the TMF transaction.

  The DRA then issues a call to WRITE the PTLF record. AutoTMF intercepts the call and begins a **separate** TMF transaction. Immediately after the write completes, because **MAXUPDATES 1** is in effect, AutoTMF calls ENDTRANSACTION to end the separate transaction.

- At step (5), AutoTMF intercepts the call to REPLY. At this point there are no active TMF transactions waiting to be committed.

In this example we see that to process a single business transaction AutoTMF generates three TMF transactions. However, each of these transactions is committed as quickly as possible.

4 Implementing AutoTMF into BASE24: Tips

When using AutoTMF to provide support for TMF in a BASE24 environment, there are a number of questions a User should ask:

- **Why am I using TMF?** Is this to enable the provision of Disaster Recovery using the services of NonStop RDF? Is this to provide additional data security of BASE24 data, by using TMF dumping? Is it to take advantage of the performance benefits in disc I/O achievable via TMF?
- **Which BASE24 files do I want to protect with TMF?** All data files? Just the (P)TDF and (P)TLF files ?
- **Are there any files which AutoTMF may not be able to protect?**
- **Are there any files which need AutoTMF to be configured in a particular fashion**?
- **How should I implement AutoTMF?** A phased approach? Big-Bang?

It is also important to realise that a BASE24 system is not simply about online transaction processing. When implementing AutoTMF it is necessary to understand the influence of all subsystems and programs that access files. For example:

- Users can modify the various BASE24 database files through Pathway.
- BASE24 supports a number of 'Batch'-type processes such as POS Settlement, Refresh and Extract.
- Users may have developed any number of bespoke, in-house applications.

In this section we provide various examples to illustrate how to configure a BASE24 application to take full advantage of AutoTMF.

4.1 *Files created programmatically by BASE24 processes*

A number of BASE24 processes create files programmatically. Typically, these processes will create a new file based on a 'template file'. For example:

- POS Transaction Log File. Created by the BASE24 POS Settlement process.
- Interchange Log Files. Created daily

4.1.1 BASE24-POS Transaction Log

Possible Implementation Approach:

1. Stop the BASE24 network.

2. Turn AUDIT on the Template PTLF file e.g.

```
FUP ALTER $vol.xxxxTPLT.POYYMMDD, AUDIT
```

3. Turn AUDIT on any existing POS Transaction Log file e.g.

```
FUP ALTER $vol.xxxxPTLF.PO050202, AUDIT
FUP ALTER $vol.xxxxPTLF.PO050201, AUDIT
```

4. Ensure that BASE24-POS Settlement and any other BASE24 Satellite processes updating the PTLF are prepared for AutoTMF.

   The BASE24 Extract process will need to be prepared for AutoTMF because it writes certain control information to the header record (i.e. relative record 0).

   NOTE: The simplest approach is to prepare ALL BASE24 Object files to use AutoTMF. One exception to this rule is the XPNET program. XPNET should NOT be prepared to use AutoTMF.

5. Start the BASE24 network.

4.1.2 BASE24 Interchange Log Files

Possible Implementation Approach:

1. Stop the BASE24 network.

2. Turn AUDIT on the Interchange Template file e.g.

```
FUP ALTER $vol.xxxxTPLT.ILYYMMDD, AUDIT
```

3. Turn AUDIT on any existing Interchange Log file e.g.

```
FUP ALTER $vol.xxxxVISA.IL050202, AUDIT
```

4. Ensure that all BASE24 Satellite processes which create or update the Interchange Log File are prepared for AutoTMF.

5. Start the BASE24 network.

4.1.3 Files created by BASE24 Refresh

Some files used by BASE24 are created by the BASE24 Refresh process, using data typically derived from some external system. Examples include:

- Cardholder Account File (CAF).
- Positive Balance File (PBF)
- Negative Files (ANEG, BNEG, NEG)

There are two types of Refresh: A ***Partial*** refresh and a ***Full*** Refresh.

Partial Refresh

A Partial Refresh takes data from the supplied input file and updates the CAF/PBF/NEG file in situ. Files refreshed using a Partial Refresh can be audited.

Full Refresh

A Full Refresh takes data from the supplied input file and first builds a new target file. The BASE24 Refresh process will, for example:

- create a temporary intermediate file called NEWCAF
- once the temporary file had been loaded, rename the current CAF to OLDCAF and rename the NEWCAF to CAF.
- Send a message to a configured list of BASE24 processes to inform them that the CAF file should be closed and re-opened.

An audited file can generally NOT be renamed.

AutoTMF currently provides a mechanism to intercept a rename and replace it with a complex set of *close*, *write* and *open* operations. However, the file being renamed must not be opened by any other process. This limitation must be taken into account when considering a BASE24 system.

4.2 *Terminal and Retailer Files*

Possible Implementation Approach:

1. Stop the BASE24 network.

2. Turn AUDIT on the terminal and retailer file e.g.

```
FUP ALTER $vol.xxxxDATA.PTDF, AUDIT

FUP ALTER $vol.xxxxDATA.PRDF, AUDIT
```

3. Consider setting the AUDITCOMPRESS option:

```
FUP ALTER $vol.xxxxDATA.PTDF, AUDITCOMPRESS
```

4. Ensure that BASE24-POS Settlement and any other BASE24 Satellite processes updating the terminal or retailer files are prepared for AutoTMF. This includes Device Handler Router Auth processes, Auth processes and Pathway Server processes.

5. Start the BASE24 network.

4.3 *Store and Forward Files*

Possible Implementation Approach:

1. Logoff from interfaces using a particular SAF. STOP BASE24 Host or interchange processes using the SAF file.

2. Turn AUDIT on the SAF data file, if one already exists, and/or the SAF Template file:

```
FUP ALTER $vol.xxxxSAF.SAF, AUDIT

FUP ALTER $vol.xxxxTPLT.SAF, AUDIT
```

3. Consider setting the AUDITCOMPRESS option:

```
FUP ALTER $vol.xxxxSAF.SAF, AUDITCOMPRESS
```

4. Ensure that BASE24 Satellite processes updating the SAF (e.g. Host and/or Interchange processes) are prepared for AutoTMF.

5. Start the BASE24 Host or Interchange processes and logon (if appropriate).

4.4 *OMF files*

Possible Implementation Approach:

1. From PATHCOM, Freeze and STOP SERVER-SEC.

2. Turn AUDIT on any existing OMF data file, if one already exists, and the OMF Template file

```
FUP ALTER $vol.xxxxOMF.Y0502020, AUDIT

FUP ALTER $vol.xxxxTPLT.Yyymmddx, AUDIT
```

3. From PATHCOM, Thaw and START SERVER-SEC.

4.5 *Special Considerations*

Experience implementing AutoTMF at a number of BASE24 sites has uncovered a few scenarios which require specific AutoTMF configuration:

4.5.1 Files needing to use the HIDEAUDIT attribute

Some of the BASE24 Pathway servers call FILEINFO (or FILERECINFO) to return the **type** parameter to determine the file type. However, BASE24 does not expect the <audit> bit to be set and this results in the file type being identified incorrectly. For Relative files, this may result in the wrong procedure call subsequently being used to position into the file.

The solution is to 'hide' all signs of audit from the calling server. This is achieved via the AutoTMF **HIDEAUDIT** attribute. For example:

```
ADD ATMFF $volume.subvol.IDF, HIDEAUDIT
```

4.5.2 Files needing to use the RECORDTX attribute

AutoTMF is only able to commit a TMF transaction if all record locks have been removed. Some BASE24 Device Handler processes, SPDH for example, leave Terminal records locked when a transaction has to be sent to some external Issuer for authorisation. This could potentially leave AutoTMF unable to commit any transactions, as illustrated in Fig. 5 below:

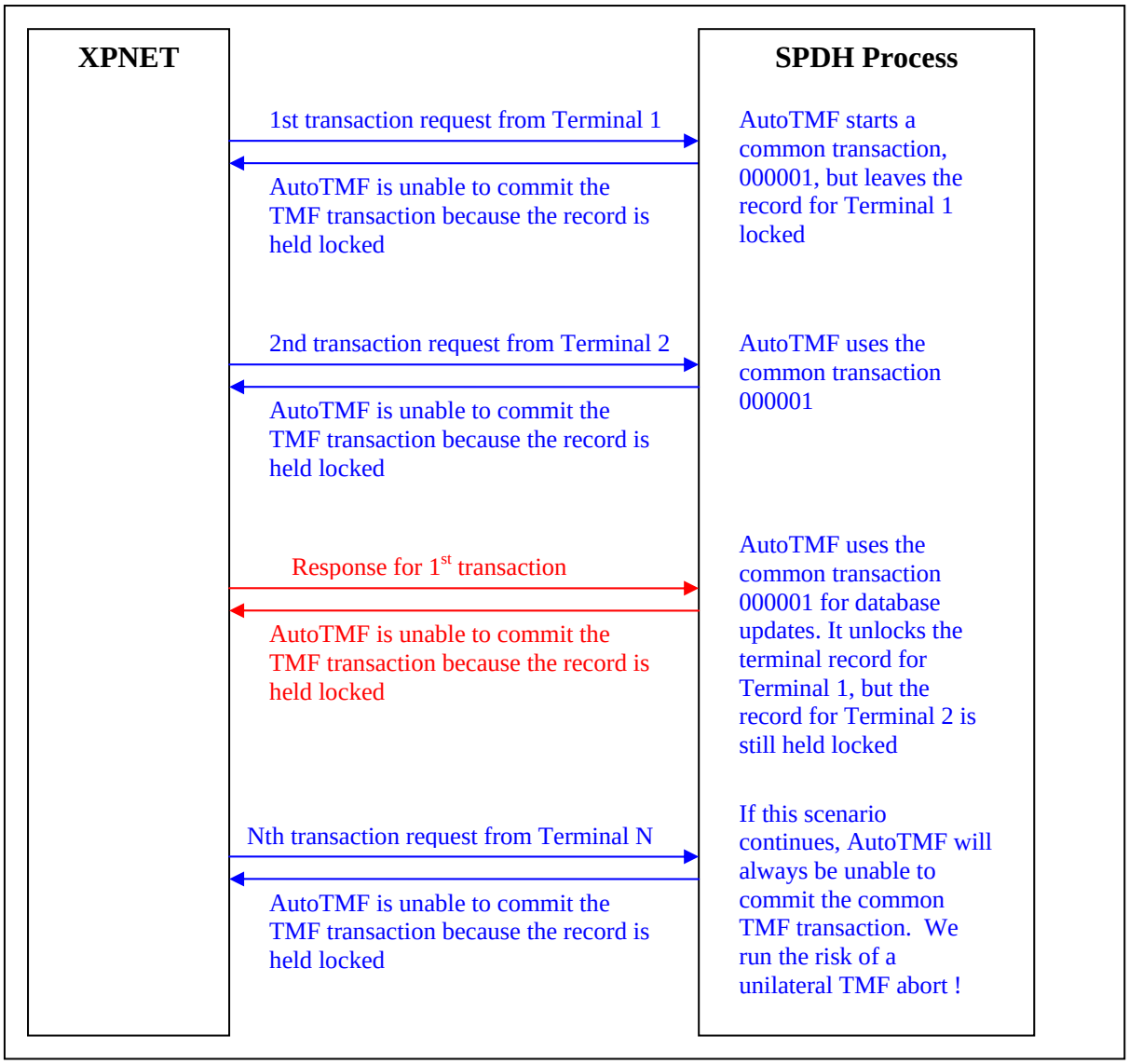


Fig. 5 – Device Handler processes leaving Records locked

If the scenario described in Fig. 5 were to continue, AutoTMF would be unable to commit its automatic TMF transaction and the TMF transaction would run the risk of being aborted unilaterally, for example if it reached the configured TMF AUTOABORT period. As a consequence of the unilateral abort, all updates performed during the course of the TMF transaction will be undone.

AutoTMF provides a mechanism to overcome this problem. It is able to start a new transaction when it recognises that a different key is being used to access a particular file. This mechanism is invoked by using the AutoTMF **RECORDTX** attribute.

RECORDTX has to be added as both an AutoTMF Fileset attribute for the file being updated AND an AutoTMF program attribute for the program performing the update. In the case of our example above, where the file would be the PTDF and the program would be the SPDH object, we would need to

```
ADD ATMFFILESET    $volume.subvol.PTDF, RECORDTX
ADD ATMFPROGRAM  $volume.subvol.SPDH, RECORDTX
```

There are some scenarios where it is not possible to use **RECORDTX**: these are documented in the AutoTMF User Guide.

Identifying that RECORDTX may be required needs a detailed understanding of the locking activities performed by application processes. However, AutoTMF provides a powerful trace utility which can be used to identify problems such as this.

4.6 *Using AutoTMF TRACE*

The AutoTMF TRACE facility provides an excellent way of understanding the file I/O activity of user processes.

The following are examples of output from ESCORT TRACEs performed on a BASE24 test application. These have been annotated to show some important features of TRACE functionality.

4.6.1 TRACE Example 1: RECORDTX on process but not on file

In our first example, we use a BASE24 Device Handler process which leaves locks on records. To prevent the problem described in section 4.5.2, we have attempted to configure **RECORDTX.** However, this has only been configured against the program file:

ADD ATMFP $B2404.PS6TOBJ.ASPDH, RECORDTX

```
16:28:47:126($TAS1) *** Start Trace *** Process started
($TAS1) Creator id 99,255,Program $B2404.PS6TOBJ.ASPDH
($TAS1) Term $SUDO, Library $B2404.XPNET.SKELB
($TAS1) AutoTMF Program: RecordTX
```

- The BASE24 SPDH satellite process $TAS1 has been configured with the **RECORDTX** parameter.

```
16:28:47:129($TAS1) KEYPOSITIONX($B2404.TES1DATA.PTDD1:6)
($TAS1) Position: Exact,Len=16,Key "123456700333001 "
16:28:47:130($TAS1) AutoTMF BEGINTRANSACTION(\SYS1(1).1.2789758)
```

- $TAS1 issues a READUPDATELOCKX call against a Release 6 POS PTDD1 file for terminal id “123456700333001”. AutoTMF intercepts this call and starts an automatic TMF transaction.

```
16:28:47:131($TAS1) READUPDATELOCKX($B2404.TES1DATA.PTDD1:6,LC=0,CLC=0,3514,tag=0)
($TAS1) Position: Exact,Len=16,Key "123456700333001 "
16:28:47:132($TAS1) AutoTMF RESUMETRANSACTION(Null Tx)
```

- AutoTMF suspends the currently active TMF transaction.

```
16:28:47:134($TAS1) KEYPOSITION($B2404.TES1DATA.ARSP:10)
($TAS1) Position: Exact,Tag="RN",Len=6,Key "TAUR04"
16:28:47:135($TAS1) READ($B2404.TES1DATA.ARSP:10,0) Error 1
16:28:47:136($TAS1) FILEINFO($B2404.TES1DATA.ARSP:10) Error 1
16:28:47:148($TAS1) KEYPOSITION($B2404.BSTSA0.HLF:11)
($TAS1) Position: Exact,Len=8,Key "55444981"
16:28:47:149($TAS1) READ($B2404.BSTSA0.HLF:11,300)
($TAS1) Position: Exact,Len=8,Key "55444981"
```

- $TAS1 reads records from the ARSP and HLF files.

```
16:28:47:155($TAS1) REPLYX(,1978,0,tag=0)
```

- $TAS1 issues a REPLYX (in this case to the parent XPNET process). TMF transaction `1.2789758` can't be committed at this time because the PTDD1 record is still locked.

```
16:28:47:156($TAS1) READUPDATEX($RECEIVE:0,0) Nowait
16:28:47:157($TAS1) FILEINFO($RECEIVE:0) Sender: $H1AN
16:29:21:176($TAS1) AutoTMF $RECEIVE inherited(msgtag=0,Null Tx)
16:29:21:177($TAS1) AWAITIO($RECEIVE:0,236) Sender: $H1AN
16:29:21:178($TAS1) FILEINFO($RECEIVE:0) Sender: $H1AN
16:29:21:179($TAS1) FILEINFO($RECEIVE:0) Sender: $H1AN
16:29:21:180($TAS1) KEYPOSITIONX($B2404.TES1DATA.PTDD1:6)
($TAS1) Position: Exact,Len=16,Key "123456700333002 "
16:29:21:181($TAS1) AutoTMF RESUMETRANSACTION(\SYS1(1).1.2789758)
```

- $TAS1 activates the previous common TMF transaction before calling READUPDATELOCK. Notice that this is for a different terminal record. We were expecting $TAS1 to have started another transaction because RECORDTX was set. However, we gave forgotten to also set RECORDTX on the PTDD1 file.

4.6.2 TRACE Example 2: RECORDTX on both process and file

RECORDTX has been configured to both the program and the file accessed by that program:

ADD ATMFFILESET $B2404.TES1DATA.PTDD1, RECORDTX
ADD ATMFPROGRAM $B2404.PS6TOBJ.ASPDH, RECORDTX

With this configuration, AutoTMF should automatically start separate transactions for transactions accessing the PTDD1 file using different record keys.

```
12:13:10:345($TAS1) *** Start Trace *** Process started
($TAS1) Creator id 99,255,Program $B2404.PS6TOBJ.ASPDH
($TAS1) Term $ZN014.#PTPKZVC, Library $B2404.XPNET.SKELB
($TAS1) AutoTMF Program: RecordTX
```

- AutoTMF confirms that **RECORDTX** was set on the program

```
12:13:10:919($TAS1) AutoTMF TFILE Open(8, maxtx=100)
12:13:10:920($TAS1) FILE_OPEN_($B2404.TES1DATA.PTDD1:7,RW/SH, KS, ATMF record tx)
```

- AutoTMF confirms that **RECORDTX** was set on the file. We can also see that PTDD1 is opened for read/write-shared access and is key-sequenced. Also note that AutoTMF knows that with RECORDTX in place, $TAS1 could require the support of concurrent TMF transactions, so it opens the pseudo TMF TFILE.

```
12:13:12:393($TAS1) READUPDATEX($RECEIVE:0,0) Nowait
12:13:12:394($TAS1) FILEINFO($RECEIVE:0)
12:13:12:394($TAS1) AutoTMF $RECEIVE inherited(msgtag=0,Null Tx)
12:13:12:395($TAS1) AWAITIO($RECEIVE:0,2458) Sender: $H1AN
12:13:12:395($TAS1) FILEINFO($RECEIVE:0) Sender: $H1AN
12:13:12:396($TAS1) FILEINFO($RECEIVE:0) Sender: $H1AN
12:13:12:398($TAS1) KEYPOSITION($B2404.TES1DATA.PRDF:16)
($TAS1) Position: Exact,Len=19,Key "0701001000105040   "
```

```
12:13:12:399($TAS1) READ($B2404.TES1DATA.PRDF:16,1261)
($TAS1) Position: Exact,Len=19,Key "0701001000105040   "
12:13:12:400($TAS1) FILEINFO($B2404.TES1DATA.PRDF:16)
12:13:12:407($TAS1) KEYPOSITION($B2404.TES1DATA.NEGAMEX:34)
($TAS1) Position: Exact,Len=22,Key "375200000000003    000"
12:13:12:409($TAS1) READ($B2404.TES1DATA.NEGAMEX:34,100)
($TAS1) Position: Exact,Len=22,Key "375200000000003    000"
12:13:12:412($TAS1) AutoTMF BEGINTRANSACTION(\SYS1(1).1.2798267)
```

- AutoTMF intercepts the call to WRITE a record to the BASE24 POS Transaction log file. It starts a common TMF transaction.

```
12:13:12:415($TAS1) WRITE($B2404.TES1PTLF.PO030730:35,1612)
($TAS1) Position: Current=0/2,Next=0/0
12:13:12:416($TAS1) AutoTMF RESUMETRANSACTION(Null Tx)
12:13:12:418($TAS1) KEYPOSITIONX($B2404.TES1DATA.PTDD1:7)
($TAS1) Position: Exact,Len=16,Key "123456700333001 "
12:13:12:419($TAS1) AutoTMF BEGINTRANSACTION(\SYS1(1).1.2798268)
```

- AutoTMF intercepts the call to READUPDATELOCKX a record to the PTDD1 terminal file. With RECORDTX now set on both the program and the file, AutoTMF starts a new TMF transaction.

```
12:13:12:420($TAS1) READUPDATELOCKX($B2404.TES1DATA.PTDD1:7,LC=0,3514,tag=0)
($TAS1) Position: Exact,Len=16,Key "123456700333001 "
12:13:12:422($TAS1) AutoTMF RESUMETRANSACTION(Null Tx)
12:13:12:465($TAS1) KEYPOSITIONX($B2404.TES1DATA.PTDD1:7)
($TAS1) Position: Exact,Len=16,Key "123456700333001 "
12:13:12:467($TAS1) AutoTMF RESUMETRANSACTION(\SYS1(1).1.2798268)
```

- AutoTMF intercepts the call to WRITEUPDATEUNLOCKX a record to the PTDD1 terminal file. It resumes the previous TMF transaction

```
12:13:12:468($TAS1) WRITEUPDATEUNLOCKX($B2404.TES1DATA.PTDD1:7,LC=1,3514,tag=0)
($TAS1) Position: Exact,Len=16,Key "123456700333001 "
```

- With RECORDTX, AutoTMF will commit the TMF transaction.

```
12:13:12:469($TAS1) AutoTMF ENDTRANSACTION(\SYS1(1).1.2798268)
12:13:12:487($TAS1) AutoTMF TFILE Completion(\SYS1.1.2798268)
12:13:12:488($TAS1) AutoTMF RESUMETRANSACTION(Null Tx)
12:13:12:489($TAS1) KEYPOSITIONX($B2404.TES1DATA.PTDD1:7)
($TAS1) Position: Exact,Len=16,Key "123456700333001 "
12:13:12:490($TAS1) AutoTMF BEGINTRANSACTION(\SYS1(1).1.2798269)
```

- AutoTMF intercepts another call to READUPDATELOCKX the same record to the PTDD1 terminal file. The previous TMF transactions has just been committed, so AutoTMF has to start another TMF transaction

5 NonStop RDF

HP's ***NonStop RDF*** (Remote Database Facility) is a product which works in conjunction with HP's ***NonStop TMF*** to logically replicate database updates made to audited files on a primary system to one or more backup systems.

The NonStop RDF architecture is illustrated in Fig. 6 and discussed below.

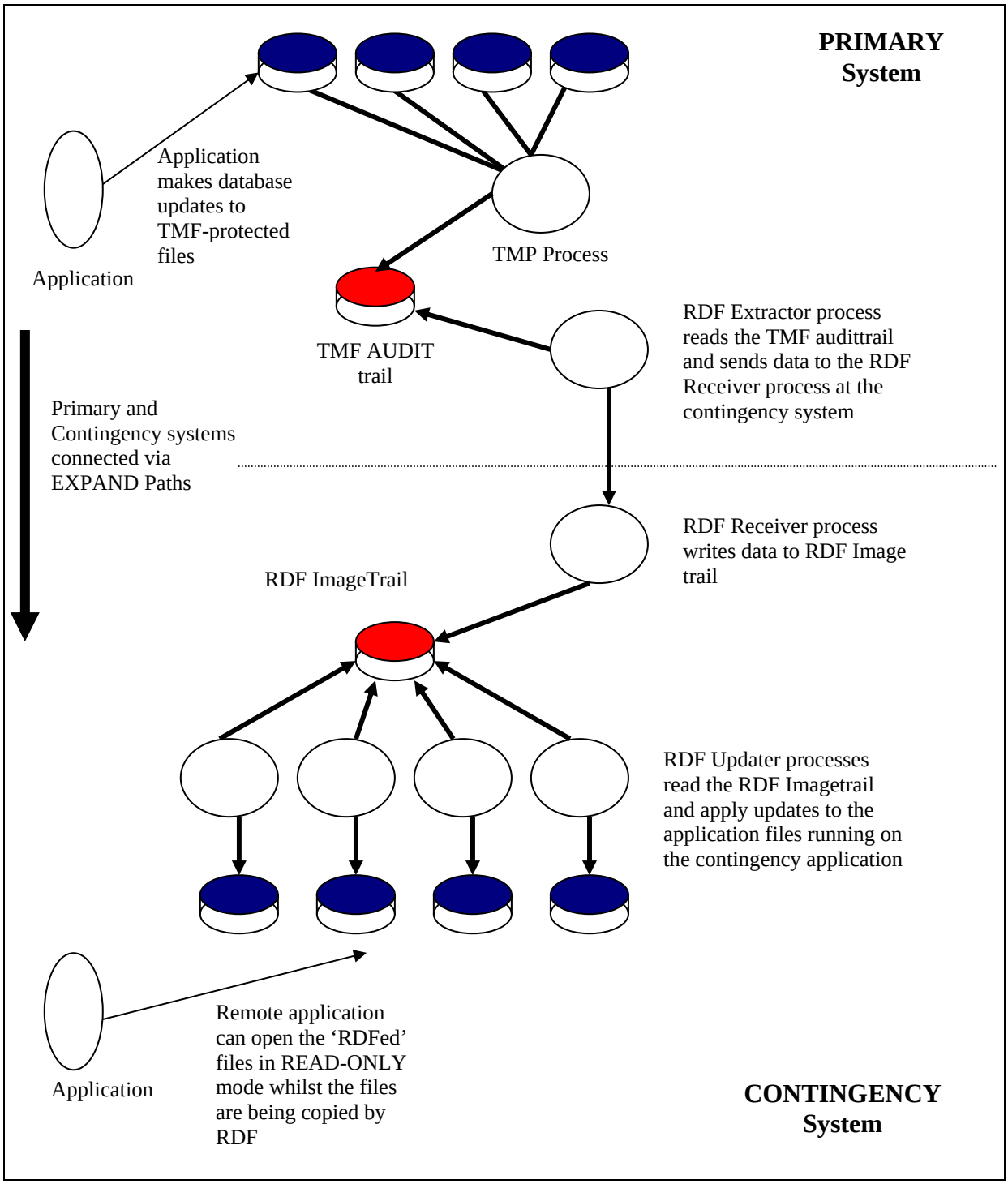


Fig. 6 - Basic RDF Systems Architecture

The Guardian Disc Process and the NonStop TMF subsystems work together to write details of database updates made to audited files to the TMF Audittrail file(s).

On the primary system, RDF Extractor processes read the TMF audittrails and sent audit data for files that have been configured to be RDF-protected to the appropriate RDF image trail on the backup system.

On the backup system, RDF Updater processes read the RDF Image trail discs and update the appropriate files.

RDF demands that the contingency database be opened in 'read-only' mode whilst it is the target for database updates. This means that a BASE24 application can not be active on the backup system.

6.4 *NonStop TMF and RDF: An example Configuration*

NonStop TMF and NonStop RDF are both highly configurable products. In this section we provide an example to illustrate how a flexible BASE24 environment can be supported.

In our example, Fig. 7, a NonStop server has been configured to run 3 separate BASE24 Logical Networks:

- **PRO1**. This Logical Network has its database spread across discs labeled $B201 - $B217, all configured to use the TMF Master Audittrail. Not all of the data files maintained by PRO1 are protected by RDF. The files that are RDF-protected are configured on discs labeled $B201 - $B212

- **PRO2**. This Logical Network has its database spread across discs labeled $B218 - $B229. These have been configured to use the TMF Auxiliary Audittrail disc $AUX01. PRO2 is not RDF-protected.

- **PRO3**. This Logical Network has its database spread across discs labeled $B230 - $B240. These have been configured to use the TMF Auxiliary Audittrail disc $AUX02. PRO2 is also not RDF-protected.

On the backup system, RDF is configured with 3 RDF Image trail files. RDF Updater processes have been configured to use these image trails. This is illustrated in Fig. 8.

This example illustrates a number of points:

- TMF and RDF can support Multiple BASE24 Logical Networks running on single NonStop Server. Not all these Logical Networks need to be RDF-protected.

- TMF and RDF can be configured to replicate only specific files in a given BASE24 Logical Network.

- It is possible to run BASE24 Logical Networks on the backup system provided that these do not access files that are the target of RDF Updaters. In our example, it would be possible to run both PRO2 and PRO3 on the backup system whilst PRO1 is running on the primary system.

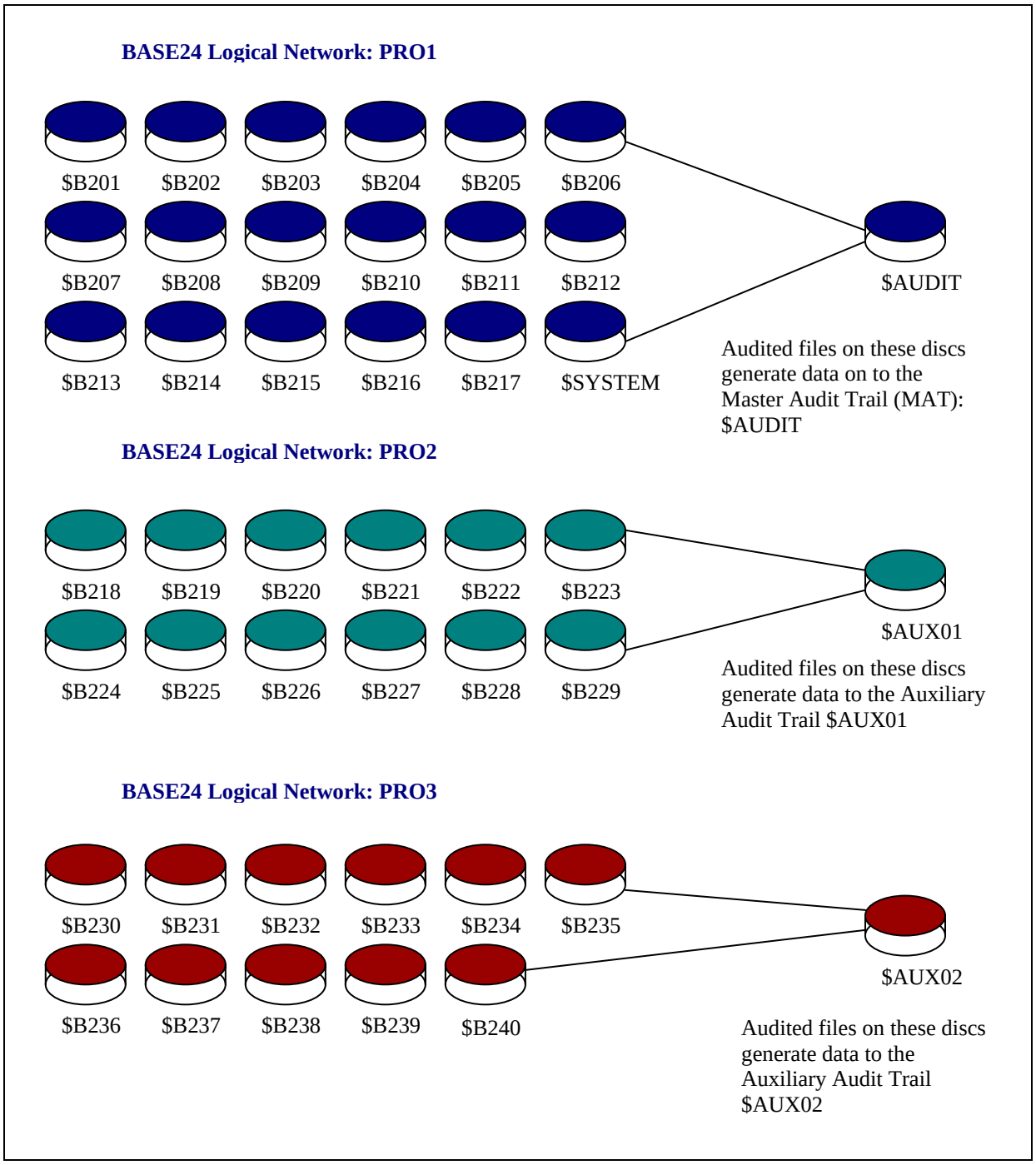


Fig. 7– An Example TMF Configuration to support multiple BASE24 Logical Networks

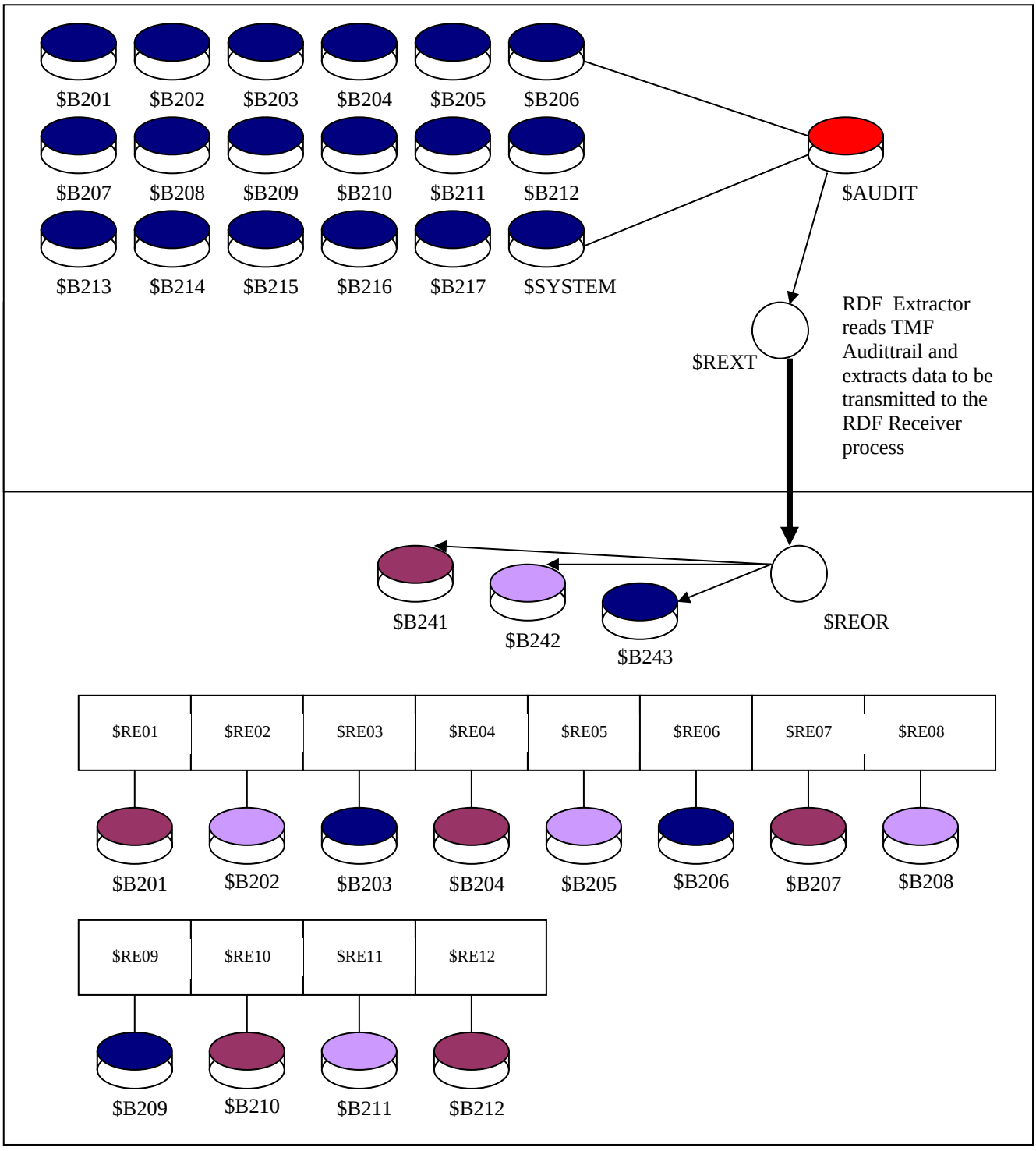


Fig. 8 - An Example TMF Configuration to support multiple BASE24 Logical Networks

6. Performance Benefits

Using AutoTMF can provide substantial improvements in disc I/O performance, especially as transaction volumes increase. This is due to the more efficient buffering techniques used by TMF

Making full advantage of these improvements can help to reduce transaction response times, eliminate potential I/O bottlenecks and improve batch response times (including processing such as BASE24-POS Settlement cutover).

In the following sections we highlight performance benefits using some real Customer examples.

6.1 *PTLF and its alternate Key files*

When BASE24 processes perform a WRITE to an audited PTLF (POS Transaction Log File), the record is held in cache, initially in a cache block marked as dirty.

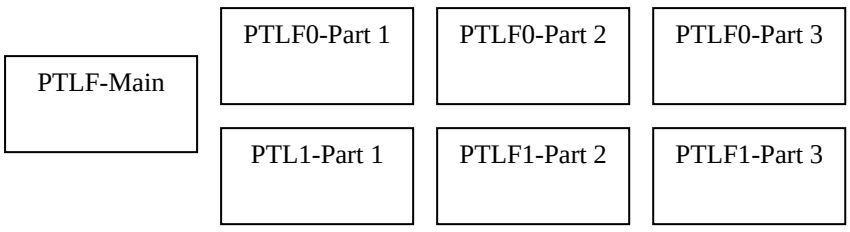
At TMF Control Point intervals, the dirty cache blocks are written to disc (and then marked as clean). TMF will write contiguous dirty cache blocks to disc in a single write operation of up to 56K bytes.

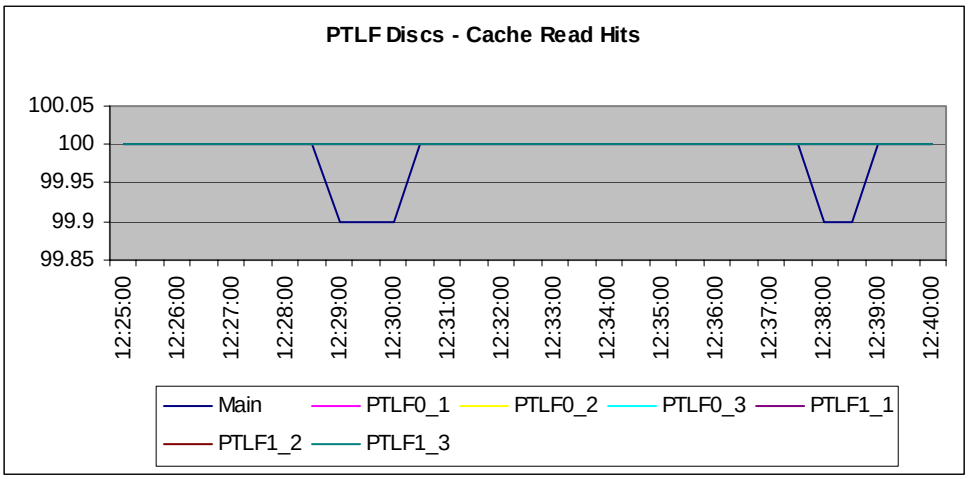
If the average length of a record written to the PTLF is 1000 bytes, then a 56K buffer will write 56 PTLF records to disc in a single physical write.

The same principle also applies to each of the PTLF alternate key files, although the situation here is somewhat different because these are Keyed files. At the start of the day, the index and data block will be located contiguously and at Control Point intervals these will be written to disc in 56K writes. As the day advances, records will be added to index and data blocks across the entire alternate key file. This is especially the case where a particularly random key is used (for example, cardholder). It becomes more likely that dirty cache blocks will not be contiguous.

Tip: Configure DP2 cache to keep all the alternate key files in memory. This may require alternate key files to be partitioned, but it affords the possibility of cache read hits of 100% and minimizes physical disc I/O.

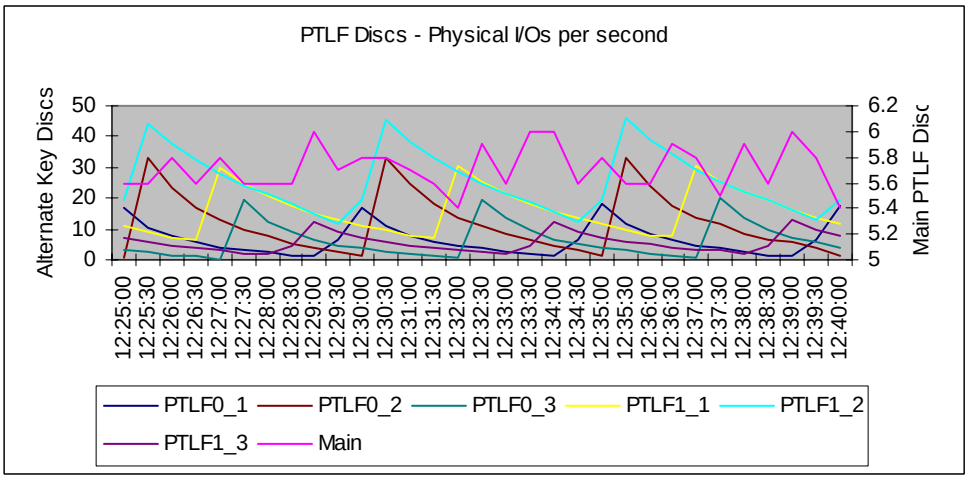
The following examples show the performance of a PTLF running at nearly 110 transactions per second. The main PTLF was file configured as a Format 2 file on a single disc. Only 2 PTLF alternate key files were configured, each across 3 partitions:





Cache Read Hits

At 110 tps, read hit rates of 100% are achieved

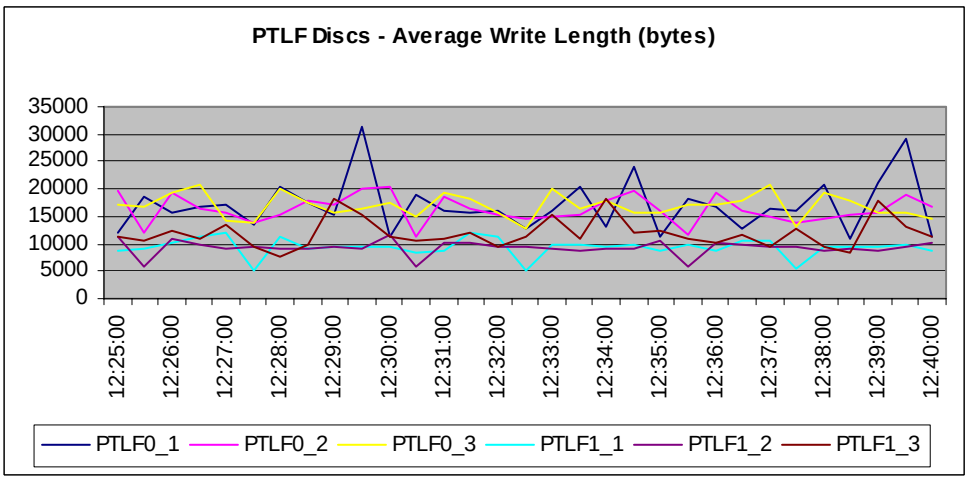


Physical Disc I/Os

At 110 tps:

the main PTLF disc receives around 6 physical writes per second.

The alternate PTLF key discs see more writes at the start of each Control Point

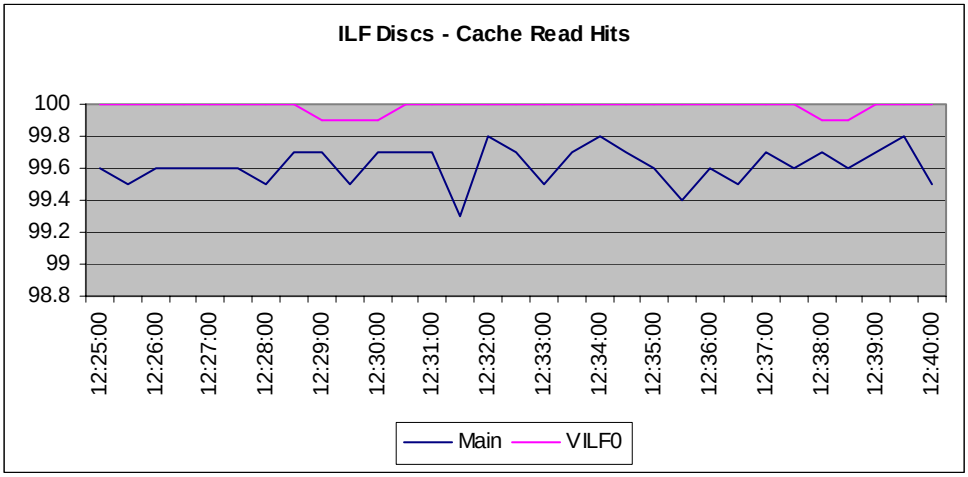


Average bytes per write

Typically 10K – 20K per physical write

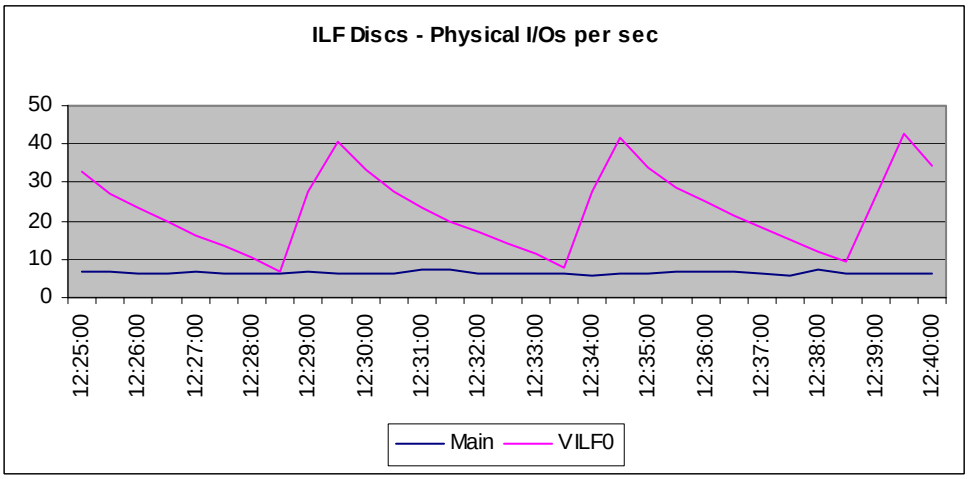
6.2 *ILF and its alternate Key file*

The following graphs show utilization metrics for discs holding the Visa Interchange Log file and its alternate key. Both files were configured on separate discs. Sufficient DP2 cache was configured to hold a complete days' ILF alternate key file in cache.



Cache Read Hits

At 60 tps, read hit rates near 100% are achieved

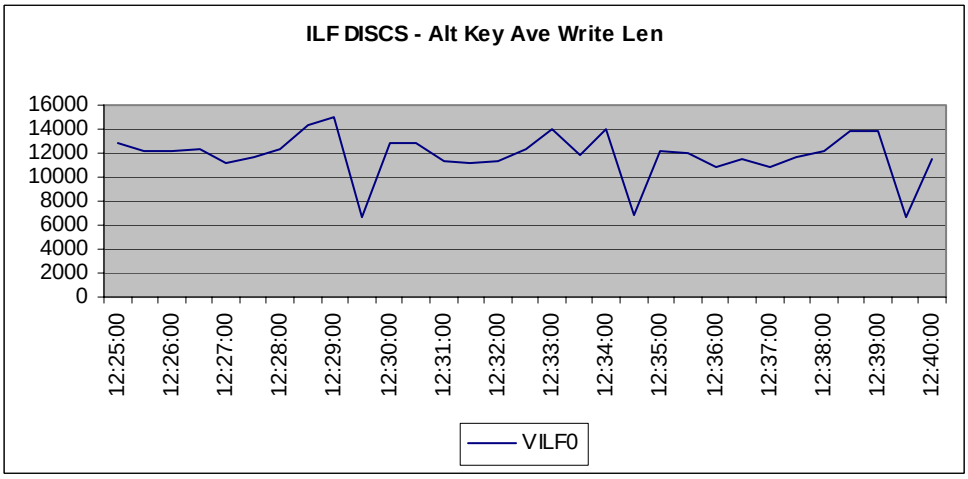


Physical Disc I/Os

At 60 tps:

the main ILF disc receives around 6 physical writes per second.

The alternate ILF key disc sees more writes at the start of each Control Point

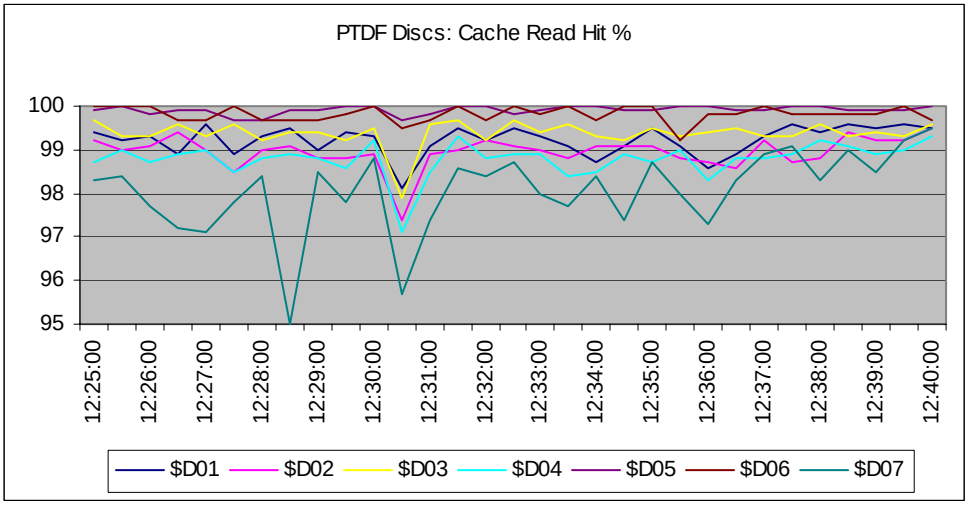


Average bytes per write

Around 12K per physical write

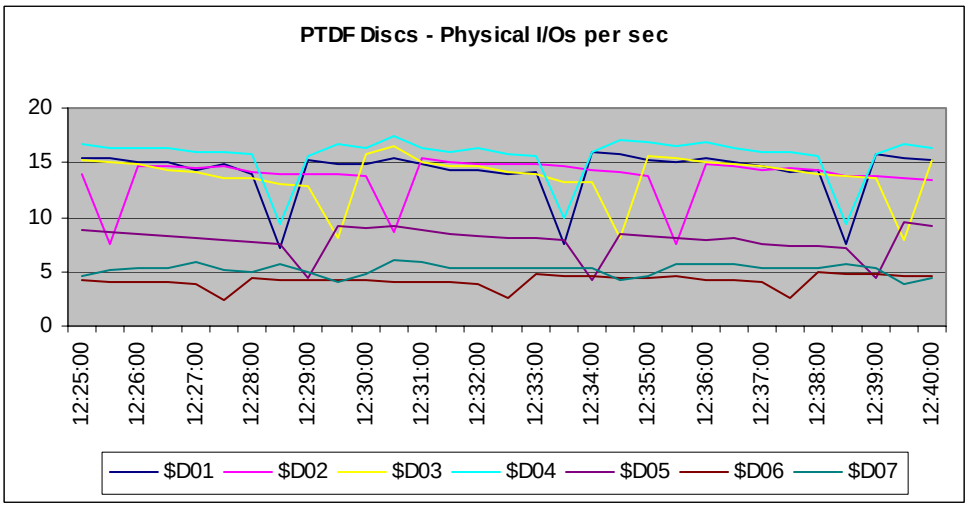
6.3 *POS Terminal Definition File (PTDF)*

The following graphs show utilization metrics for discs holding a partitioned PTDF containing over 200,000 records and processing some 60 transactions per second. Each PTDF record is read and updated twice per transaction.



Cache Read Hits

At 60 tps, read hit rates of between 99-100% are achieved for most partitions

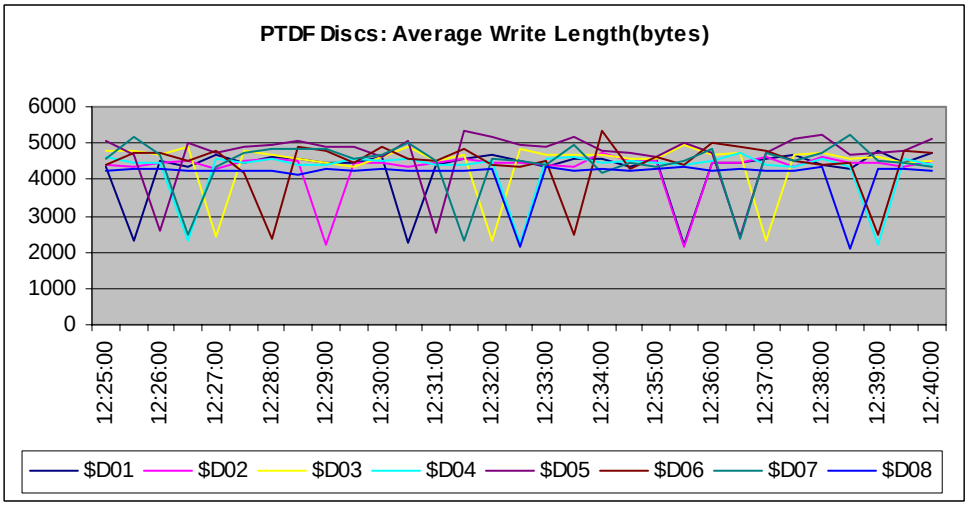


Physical Disc I/Os

At 60 tps:

Up to 18 Physical I/Os to the busiest partition.

These I/Os are writes of dirty cache blocks at Control Point intervals.



Average bytes per write

Between 2-5K per physical write i.e. typically a single 4K block is written per write during the Control point intervals. This suggests that access is random across the PTDF file.

6.4 *Other Performance Benefits:*

- Significant reduction in the overhead associated with Block Splits.
- Speed-up in the time to complete BASE24-POS Settlement cutover (increasing benefits for Users with larger Terminal and Retailer bases).
- Improvements in Batch Processing times.
- With data such as Transaction Log data and Terminal data protected, Users may consider whether it is necessary to run BASE24 XPNET processes in 'Hot Standby' mode. (Note: this has to be considered on a customer-by-customer basis).

6.5 *Increase in Resource utilization*

Although most BASE24 Users will see improvements in disc I/O, adding TMF support:

- Will require additional cpu utilization for the TMF $TMP process.
- Is likely to lead to a small increase in the cpu utilization of DP2 processes controlling files which are now audited.

About the Author

The views and opinions expressed in this document are solely those of the author. Philip Nye has 20 years experience working with BASE24, on BASE24 systems running at 10 million business transactions per day. He first implemented a Disaster Recovery solution into BASE24, based on TMF and RDF, in 1987. Phil holds a BSc Honours Degree in Computer Science from Brunel University, UK and founded Cardlink in 1995.

Phil can be contacted at: phil@cardlink.co.uk