THE DEVELOPMENT OF AN RJE/X.25 PAD: A CASE STUDY

C.W. Carey
C. Hattingh
D.G. Kourie
R.J. van den Heever
R.F. Verkroost
Department of Computer Science
University of Pretoria
0002 Pretoria

This paper describes the practical aspects of a protocol system implementation. The software for a BSC/X.25 Packet Assembler/Disassembler (PAD) was designed, developed and tested. The PAD was designed to support the communication of IBM 2780/3780 and compatible Remote Job Entry (RJE) stations over an X.25 network.

In the paper the RPAD/X.25 network environment is introduced, the target system is described, and the software design is discussed. Some attention is given to the problems and techniques of protocol specification, and the methods employed in this particular project are discussed. The implementation and testing phases of the development are described, and finally, the importance of complete, accurate and structured documentation is stressed.

1. INTRODUCTION

The increasing use of packet switching technology for data communications has resulted in the development of a variety of Packet Assembler/Disassemblers (PADs). These PADs enable non-X.25 devices to communicate over an X.25 network - the PAD 'maps' the device native protocol to/from X.25.

In this paper a case study of a PAD development is described, including the design, development and testing of the software for a particular target system enabling Remote Job Entry (RJE) devices - in particular IBM 2780/3780 and related devices - to communicate over an X.25 network. The PAD is referred to as RPAD (for RJE PAD), and its associated software as RTX (for RJE/X.25).

The RPAD/X.25 environment, the target system, and the software design are discussed. The problems and techniques of protocol specification are addressed by descibing the methods used in this particular development. The implementation and testing phases of the project are described, and some attention is given to the all-too-frequently shunned area of documentation.

An RJE or X780 device (used here as a generic term for all RJE devices supported by RPAD) is typically a station comprising some or all of the following components:

- card reader
- card punch
- printer
- floppy storage
- BSC adapter

The stations are traditionally used for remote data entry to a mainframe.

The IBM 3780 protocol used by these devices is a subset of standard IBM BSC (Binary Synchronous Communication). It operates in point-to-point or multipoint mode, half duplex, and is symmetrical at both the transmitting and receiving ends of the communication channel. RPAD, however, supports point-to-point working only.

Although RJE stations are becoming somewhat outdated, the 3780 protocol is widely emulated by non-RJE devices and used for file transfer. Indeed, 3780 emulator cards are available for IBM PC's to transfer files to/from (for example) an IBM mainframe. Moreover, through RPAD, it is possible to transfer files to/from other 3780 emulators (or equipped PCs) worldwide through an X.25 network such as SAPONET.

2. OVERVIEW OF RPAD/X.25 ENVIRNONMENT

Figure 1 illustrates the context in which RPAD is used in the X.25 environment. In the classical environment, X780 devices communicate with a host machine or similar X780 device by direct or switched (dial-up) connection using the BSC protocol. RPAD enables X780 devices and Host machines to communicate with each other across an X.25 network, thereby harnessing the advantages of packet switching, as well as widening connectivity.

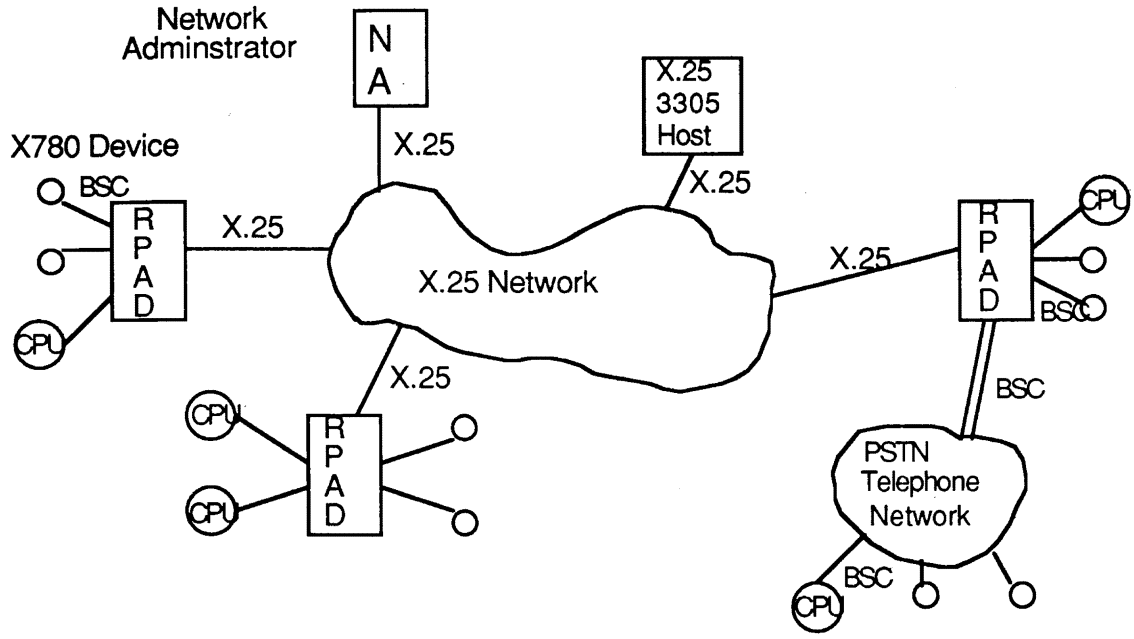

figure 1
RJE/X.25 Enviroment

The X780 device now communicates with RPAD using a direct (or switched) BSC connection (ie RPAD emulates an X780 device); RPAD maps the BSC procedures to X.25 and communicates with its remote peer RPAD using the 3305 [4] level 4 transport protocol; the remote RPAD maps the X.25 procedures back to BSC and communicates with the remote X780 device using a direct BSC connection (again, RPAD emulates an X780 device). Both X780 devices are unaware of the intervening network connection, thus obviating the need for any hardware or software modifications.

The functions of RPAD include:

- BSC communication with the X780 device
- X.25 communication with the network
- mapping BSC control procedures to X.25 according to the 3305 transport protocol
- BSC block to X.25 packet transformation
- X.25 packet to BSC block transformation

In addition, the particular PAD developed offers a number of network management facilities, including statistics, billing, and a centralised network management centre for network/PAD configuration, down-line (through X.25) loading of PAD software, billing and statistics reporting, and on-line network monitoring and.control.

3. THE TARGET HARDWARE SYSTEM

The target system is the Amdahl 4400 series of Network Concentrator (NC) and Network Administrator (NA). RPAD is essentially a suitably equipped (software) NC; the NA, an integral part of the system, is used for network/NC configuration, NC software generation, down-line loading of NC software, network control and monitoring, and billing and statistics.

3.1 Hardware

The basic PAD is a custom-built multilayer board comprising:

- 8 RS-232-C I/O ports
- 2 Z80 I/O processors
- Intel 8086 processor (BP or Block Processor)
- associated support and RAM/ROM

The 8086 processor (BP) performs the higher level protocol functions (eg X.25 packet level, BSC etc), while the two Z80 processors function as I/O processors (IOP), each servicing 4 ports (BSC or X.25).

The basic PAD is expandable to a 40 port unit by the addition of further so-called passive boards.

3.2 Software

Figure 2 illustrates the PAD software environment. Resident software includes:

- BP EXEC, the 8086 executive. Essentially a repeating loop, its main tasks are the despatch of BP processes for the 8086, buffer management, and timer services.
- IOP EXEC, the Z80 executives.
- X.25 utility BP processes: Packet Handler (XPH) and Link Handler (XLL).
- IOP processes: X.25, Async, BSC.
- PMR BP process, an interface (via X.25) to the Network Administrator.

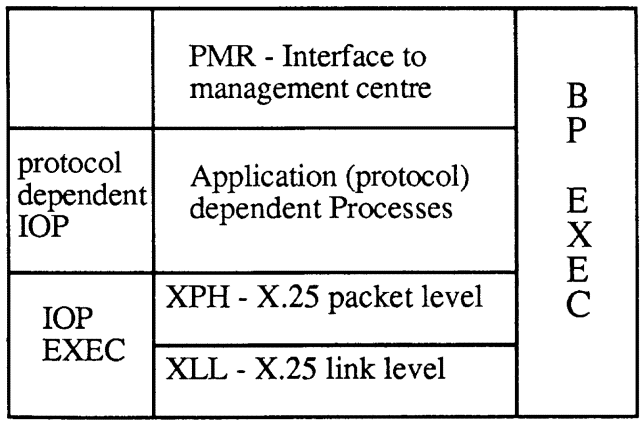

figure 2

PAD Software Environment

In addition to the IOP and BP EXECs, PMR and X.25 utility processes, protocol dependent software (IOP, and BP processes) is added to customise the PAD - eg to support Async, or SDLC, or 3270 BSC, or RJE BSC etc. This is indicated in Figure 2.

The RTX design thus amounted to the design of BP processes that would:

- perform 3780 BSC protocol function
- establish and clear calls through the X.25 network
- translate BSC to X.25 procedures and vice versa (according to 3305 protocol)
- interface with the network service provider (XPH process)
- provide a user interface for parameter negotiation, addressing etc.
- accumulate billing and statistics data
- communicate with the NA via PMR

Before discussing the design, it is instructive to examine the software environment a little more closely.

A single BP process executes on the 8086 block processor (BP) at any one time. Several processes are resident in memory, each dedicated to a very fixed function.

64K memory is available to all processes as shared memory for inter- process communication. This memory is divided up into 70-byte buffers which are managed by the BP EXEC (they are acquired and released by a process, ie a buffer is always owned by one specific process until it releases it). Inter-process communication is further effected by means of 16-byte messages (with tagged buffers), sent from the originating process to the destination process (via the BP EXEC).

BP EXEC maintains a despatch queue of inter-process messages, adds to the queue when a message is sent, and invokes the appropriate destination process when the message reaches the front of the FIFO queue. These inter-process messages (called worklist entries) have a standard format, identifying the sender process, the receiver process, an optional chain of buffers, the type of message and 6 bytes optional of information.

Interfaces between processes are kept clean by defining a small set of message types (events) allowed on each particular interface between two processes.

4. DESIGN, SPECIFICATION, IMPELEMENTATION AND TESTING

As detailed in section 3.2, the resident software of the PAD (independent of the protocol supported) includes:

- X.25 support (physical level: XLL, and packet level: XPH)
- the three executives
- an interface to a network management centre (PMR)

Supplementary to these basic processes residing in each PAD are the processes designed to support the specific protocol which the PAD offers to its user, in this case the RJE 3780 protocol.

4.1 Software Design

The software was designed subject to the following two considerations:

- A transport handler compatible with ISO standards was not feasible in this implementation. This resulted from the fact that RPAD interfaces not only to peer RPADs, but also to existing transport implementations in foreign PADs and IBM host/front-end processors. The transport protocol traditionally designed for higher level communications supporting BSC device connections, is the 3305 (also known as BPAD) protocol [4]
- A single protocol handler process, which functionally seems to be the cleanest design, gave rise to an extremely complex process. Since this single process would have to support both the 3780 protocol to the device, and the 3305 transport protocol to the peer PAD/host, its state machine model would be represented by the cross-product of the state machines of the two protocols.

For these reasons the protocol software of RPAD was designed as three distinct processes: two

protocol handler processes (one each for 3780 BSC and 3305), and a user interface process. The following processes were thus defined :

- Device Manager (RDM): A multi-instance process emulating the 3780 protocol to the device. Each RDM handles a single device. It interfaces with the IOP resident process for data to/from the device, and to the transport handler for data to/from the network.
- Transport Handler (RTH): A single-instance process supporting the 3305 transport protocol to its peer on the remote side of the network. It interfaces with the RDM process on the device side and XPH (X.25 packet level process) on the network side. RTH also performs connection establishment and clearing, addressing, application hunting, block/packet transformations and error recovery procedures.
- User Interface (RUI): A single-instance process assisting with outgoing call establishment. RUI intercepts call establishment data from the user, allows for negotiation of certain attributes and eventually translates the user call establishment data from the user's format to the format expected by the 3305 protocol.

The design of the RTX processes and their inter-relationships are illustrated in Figure 3.

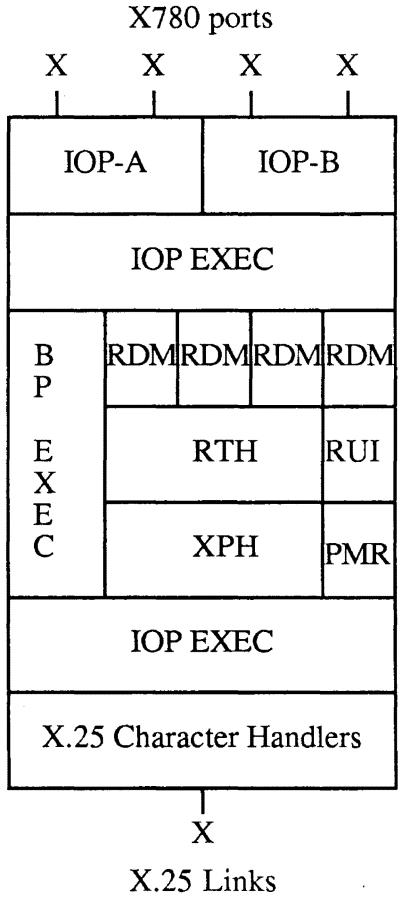

figure 3
RTX Software Architecture

4.2 Protocol Specification Methods

There is now almost universal recognition of the need for a thorough specification of a software product before proceeding to the coding/ implementation phase. In general, the effort involved in specification helps to clarify the nature of the problem at hand, and enhances the software system in terms of its modularity, portability, reliability, maintainability, etc. [1,2].

It is also worth noting that a good product specification greatly facilitates the coding effort. In terms of text-book descriptions of the software development process, completion of the product specification marks a point at which the systems analyst is able to disengage somewhat from the process, while the programmer then proceeds with coding [2].

4.2.1 Specification as a Top-down Process

In practice the level of specification may vary greatly, depending on the purpose for which it is done. At one extreme, the mere breakdown of the product into functional processes accompanied by a verbal description of each process function (as, for example, the functional units described in the preceding section (4.1)) may be regarded as a specification. The purpose of this specification is to clarify the way in which the major high-level functions of the product are to be achieved.

At a more detailed level, the variations in expected input to the (functionally determined) processes may be described together with the expected outputs. In communicating processes (be they sequential or concurrent), the time dependencies between input and output should also be carefully specified. The way in which this was done for the product at hand is set out in 4.2.2 below.

Frequently, such an input/output analysis may suggest how each functional process may be broken down into logical units or modules, each of which caters for its own level of functionality This level of specification for the product at hand is described in 4.2.3.

Finally, the way in which each smaller module achieves its functionality may be described. The level of detail given at this point should be sufficient for the programmer to commence his task. Clearly this will be determined by the resource environment in the sense that a programmer with a sound knowledge of the problem at hand may require only the briefest of detail, in contrast to an inexperienced novice. At any rate, it is somewhat Utopian to imagine that the systems analyst may be entirely removed from the scene after this level of specification has been provided. In a practical non-standard application (such as the product being described) a degree of post-specification interaction between systems analyst and programmer is inevitable. The approach taken to specification at this level is described in 4.2.4.

Each level of specification is accompanied by an associated verification process. Usually this is at a fairly informal level, and involves thinking about test-case inputs to the design which might render unacceptable outputs. However, if appropriate specification languages are used it may be possible to formally verify that each module, as well as the entire system, will react according to specification - ie yield predicted output for all possible inputs. This formal verification may also occur at the code level, where statements of the specification language serve as imbedded assertions in the code (in the form of comments) and are such that at each point an assertion can be shown to be valid provided that foregoing assertions are valid. At the limit, the formal verification process may be automated [3].

During the project at hand, no large-scale formal verification proofs were attempted, although loop-invariants were sometimes defined and used in manual exercises to prove the correctness of particularly complex sections of code. Neither was a formal specification language used to specify the product. In retrospect, it would have been both interesting and challenging to have done a detailed specification of the product in terms of LOTOS, the specification language proposed by ISO FDT subgroup C. In [6] an introduction to LOTOS is given, together with an outline of how aspects of the product might be specified in this language.

4.2.2 Message Flow Specification

Expected inputs (and outputs) to the RPAD functional processes (RUI, RTH and RDM) are inter-process messages whose format and meaning depend, on the one hand on the hardware/software environment, and on the other hand, on the protocol being implemented. Syntactically they appear as so-called worklist entries - ie a specific data-structure imposed by the operating system which also imposes a semantic interpretation on certain fields of the structure. The residual semantic interpretation of each worklist entry is determined by the product designers, so that each worklist entry acquires a unique name and code combination to designate

its meaning (eg Connect Request <2-3.01>).

These worklist entries are limited in number so that the specification challenge is not so much the determination of expected inputs and outputs, but resolving the inter-relationship between them. (Note, however, that it is obviously important to list and document each possible worklist entry, together with a description of its meaning, as part of the specification.) The asynchronous full-duplex nature of the communication renders it particularly important to precisely specify the time-dependencies between input and output in a clear but concise manner. The specification method explained below was considered particularly appropriate for this purpose.

The method starts with the identification of discrete protocol functions (eg connection establishment for successful autocall, connection establishment for unsuccessful autocall, etc.). Each such function involves the exchange of messages between two entities (A and B) representing the two communicating RPADS. These entities in turn contain processes which exchange messages not only internally between themselves, but also between the local X780 device(s) and themselves. For the present purposes, these X780 devices will also be referred to as processes.

As a general comment, it should be noted that while the present design did not follow the precise layered design proposed by ISO [7] because of the peculiarities of the environment (eg the 3305 protocol is a non-ISO transport layer), the functional processes defined above are in fact layered, in the sense of being distinct entities which are oriented towards communicating with a peer entity. Hence, XPH communicates with a peer XPH on the other side of the network (or with an equivalent process if communication is through a device implementing the 3305 protocol) and 'services' its local RTH process. Similarly RTH communicates with a peer RTH (or equivalent process) on the other side of the network, and services RDM.

At this point, there is a deviation from the conventional layered model, in that RDM may be regarded as being divided into two halves. The 'top' half of RDM communicates with its peer BSC process located in the local X780 device, while the 'bottom' half communicates with a peer 'bottom' half RDM across the network. The top half uses IOP to relay its messages, and the bottom half, the lower layer levels (RTH and XPH). (RUI and PMR are considered to be service processes.) By virtue of the above, there is a logical ordering of message flows through the processes, in that outbound messages from a device flow from device to IOP to RDM to RTH to XPH to network, and vice-versa for inbound messages. For the present discussion purposes, therefore, the highest layer process will be regarded as IOP, followed by RDM, RTH, then XPH.

Each protocol function may now be specified on a so-called message flow diagram, illustrated in figure 4. In such a diagram, entity A and its associated processes are represented on one side of a page, and entity B with its processes on the other. Each process involved in the protocol function being described is represented by a vertical line drawn down the page. Such 'process lines' are ordered by drawing the highest layer process line of entity A on the extreme left, followed by the next highest layer process line, etc. The ordering for entity B's process lines is the mirror image of that of entity A on the right hand side of the page. The location of the process line for a service process is somewhat arbitrary, but should be as close as possible to the layered process with which communication is most frequent.

Message flows (ie the sending and receipt of worklist entries) are represented by horizontal lines (called message lines) between process lines, with an arrowhead designating the receiving process. Each message line is annotated by the unique worklist entry name and code combination, so that the type of message may be easily identified. The higher up on the diagram that a message line appears, the earlier in time the message is considered to have been sent. However, there is no particular time scale associated with the vertical axis, so that the only temporal statements which can be deduced from the diagram are of the type 'message P was sent (received) before (or after or simultaneously with) message Q'.

A vertical line alongside a process line is used to designate a timer which starts at the highest point of the line and elapses at the lowest point (where an arrowhead is placed). Further brief annotations are permitted on the diagram to remove possible ambiguous interpretations. (Eg 'ignore' to denote that a process has ignored a message.)

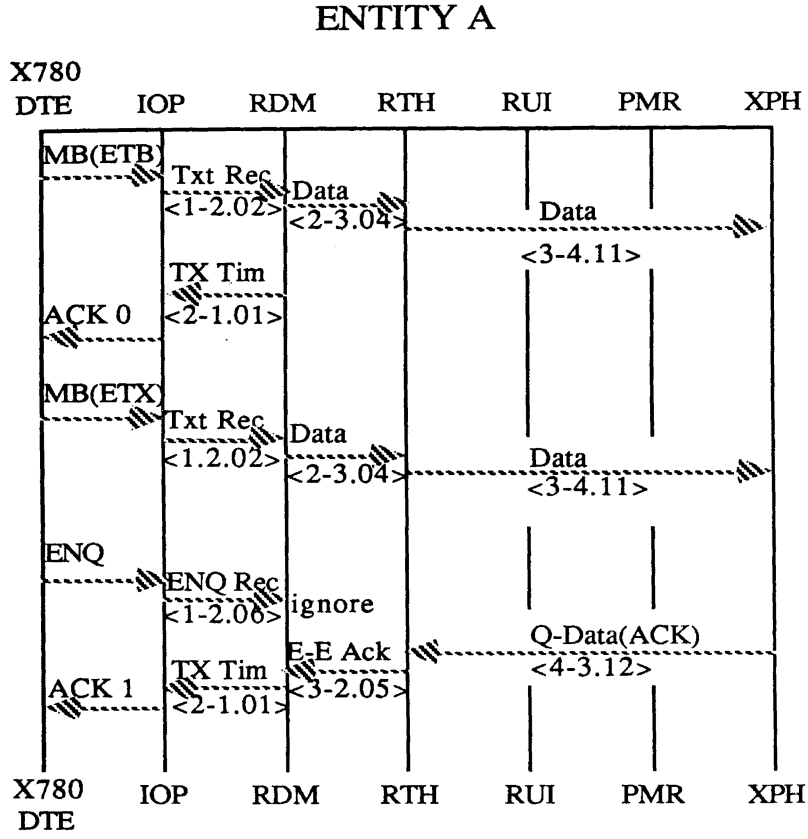

figure 4

Message Flow Diagram

4.2.3 State Diagrams

While message flow diagrams give a good visual representation of the temporal relationship between the messages which are exchanged, they do not provide clear guidelines for further decomposition of the processes involved. State diagrams fill this vacuum for the protocol-handling processes, namely RTH (handling the 3305 protocol) and RDM (handling the BSC protocol).

In a state diagram each node (or state) may be viewed as a distinct submodule. A transition to a new (or possibly the same) node is indicated by a directed arc, labelled by the message sent out by the originating node. Note that it is not implied that the destination node receives this message; rather, the destination node indicates which submodule will handle the next message received by the process. The actual destination of the message is apparent in the message flow diagrams, not in the state diagrams.

There is a relatively simple heuristic method for moving from the message flow diagrams to a set of state diagrams. (Note, however, that other equivalent state diagrams might be possible.) This heuristic is encapsulated in the following propositions which hold (with some exceptions) :

a) Each process in the present implementation runs to completion. Hence, once it has received a message it cannot be interrupted by another message.
b) Because of proposition a) a process may be considered to be in a given state until it sends out the last of its messages to non-service processes. Usually one and only one such message is sent out.
c) By virtue of propositions a) and b), a process line in a message flow diagram may be

divided into segments which end just after a message line from that process to a non-service process. These segments represent process states.

d) Within each segment determined in c) there will ordinarily be only one message line from a non-service process entering the segment, followed at a later point in time by only one message line from the segment to another non-service process. These message lines together with the state of the previous segment provide a firm guide to the state of the current segment.

e) Exceptions to the propositions a) to d) occur when, for example, an incoming message is ignored by a process. Such situations are dealt with on an ad hoc basis.

In this way, state diagrams were deduced from message flow diagrams for the aforementioned processes. It is interesting to note that in the case of RTH, the full 3305 protocol was not implemented, but a subset of the protocol which would still respond sensibly to other existing implementations (eg the Comm-Pro 3305 implementation) at a remote location. Hence, the state diagrams obtained for the present implementation differ slightly from the published version [4].

These state diagrams could form the basis for describing the state of an entire communicating system (including all non-service processes within two communicating RPADS together with the state of the communications channel). Tools exist for examining the progression of communicating systems from one such global state to the next to identify potential deadlocks, non-reachable global states, etc. [10]. Without these tools, verification for these conditions was done manually on a per process basis.

The fact that ISO FDT subgroup B has focused on the specification of protocols in terms of state diagrams demonstrates the wide acceptance that this method of specification has gained. Indeed, a Pascal-like language called Estelle has been designed to specify the semantics of a state diagram representation [9]. In the present study, specification in this way was not considered.

Finally, it is worth noting that, although the RUI process is not state-driven, the message flow diagrams nevertheless provided a valuable guide to decomposition of this process into submodules.

4.2.4 Pseudo Code Specification

Each submodule was specified in terms of structured English or pseudo code. It was considered appropriate to standardise some aspects of this effort to improve readability and facilitate the coding of the product. These guidelines are now given:

For each of the functional processes, RDM, RTH and RUI, all significantly complex data-types were initially identified and diagrammatically represented.

Guided by the state diagrams, each functional process was divided into procedures. All procedures start with a p_; all global variables (ie global with respect to one of the functional processes RDM, RTH or RUI) start with a gv_; all global types start with a gt_ and all global constants start with gc_. An early proposal to include a numbering system within this naming convention was initially followed, but proved too cumbersome to maintain and was later dropped.

All procedures start with a prologue containing the following information: procedure name, date, specifier's name, a brief abstract describing the procedure's functions and peculiarities, a list of called and calling procedures, the inputs catered for by this procedure and the outputs generated. Apart from the usual conventions of indentation, further specification was left to the discretion of the specifier.

The documents produced along these lines formed the basis for the program specifications. These specifications were heavily relied upon during the coding. They were such that they could be used by a small team of selected students who, under project-team guidance, assisted in the less complex coding areas.

There was some debate as to whether these program specifications should be amended to reflect adaptations made during coding. The motivation for doing this would be to provide a reliable entry-point for maintenance programmers at a future date. In the present context it was felt that the specifications up to state diagram level, together with the actual code (itself clearly documented by comments reflecting at least part of the program specifications) provide an

adequate entry-point. This was motivated by the fact that those likely to do maintenance (personnel not involved in development) were already sufficiently informed of the hardware and software design of the product, had developed similar products, and would therefore find a program specification unnecessarily detailed. Subsequent experience, at a stage just prior to handing over the product during which the maintenance programmers were afforded the opportunity of acquainting themselves with the product, tended to vindicate this view. However, there may well be circumstances in which an up-to-date version of the program specifications should be provided.

4.3 Implementation

As previously mentioned, the protocols (more accurately the processes driving the protocols) were specified by means of state machines. From the pictorial representation defining the states, procedures were developed handling the cross-product of each state and each possible input message to the process. Although a substantial part of the input message set (process interface) of the process is invalid for any particular state, the code caters explicitly for each possible combination. Most of the input messages cause a branch to one of a much smaller set of error procedures.

Both the device manager (RDM) and the transport handler (RTH) are largely state driven, with the exception of a few functions (management, statistics and billing) which are independent of the protocol state machine. These functions are handled in separate procedures.

RUI is basically a parser and a translator. It takes the contents of the so-called 'Call Request Block' received from the user (in which several different options exist for the user to specify his connection destination and parameters) and fills out the required parts of the X.25 Call Request Packet, ie it fills out the fields with default information as well as with information explicitly supplied by the user. RUI provides all addressing translation information, except local and remote hunting (rotary destination lists) which is done by RTH.

The implementation language is largely PASCAL, chosen for features including programming ease, speed of development, debugging time and certain compiler features. Minor sections of the code are implemented in 8086 Assembler, mainly used for time-critical code and facilities not available in PASCAL (eg operations on register and stack contents).

The environment/hardware used for implementation includes the target hardware (as described earlier) and an Intel Microprocessor Development System (MDS). The MDS development provides a primary level of unit testing and debugging facilities before the code is transferred and tested on the target hardware. All code is developed, compiled and linked into loadable modules before it is transferred.

4.4 Testing

4.4.1 Testing Overview

The matter of adequate code testing tends to be sorely neglected in many software implementations. This is borne out by the proverbial scepticism towards computers so frequently encountered outside the world of computer-specialists. Reasons for this neglect may be found at many levels, some of which are suggested below:

- Firstly, there is the psychological disposition of the programmer who has applied himself with such devotion to his coding task, that he finds it difficult to imagine that he may have overlooked various logical errors.
- There is the fact that products tend to be developed against tight time schedules. Hence, when time-limits for the various phases are overrun, management may be tempted to cut back on testing time, rather than postpone the delivery date.
- Also, testing is not always the most exciting activity, and may be done in a shoddy fashion, merely to ' get the product off our back' .
- There is an art to designing tests, particularly tests to trace those paths through the code which only arise in the rarest of circumstances. Acquiring this art takes time and effort.
- It is extremely difficult to test communications software comprehensively, especially to

simulate the time inter-dependencies which arise from the asynchronous nature of communication in the real environment.
- Finally, test facilities are sometimes expensive, both in terms of man-hours, and in terms of hardware/software resources either to facilitate the test process, or to accurately reflect the environment in which the product is to function.

Clearly, in any system of significant size, a totally comprehensive test is simply not feasible. Even a formally verified system merely verifies correctness with respect to a specification which may itself be flawed in some way. However, even though the notion of what constitutes an adequate test procedure may be somewhat fuzzy, to ignore or underplay the importance of adequate testing is to run the risk of unreliable software. This must inevitably rebound on management in the form of dissatisfied clients, loss of reputation and loss of profits. Hence it is important that the software manager not only carefully budgets for testing (both in terms of time and cost), but that he insists that the testing be rigorously done.

There are several possible approaches to organising and scheduling tests. At the start of the project under discussion, several discrete test phases were identified and scheduled into the overall project plan. The test phases were designated by the following names, and scheduled to occur in the order given, once coding had been completed.

- Unit Testing
- Integration Testing
- Alpha Testing
- Acceptance Testing
- Beta Testing

The issues addressed during each of these test phases will now be discussed.

4.4.2 Unit Testing

Once a compilable module of code is affirmed to be syntactically error-free, it may be fed with a well-chosen set of test inputs and verified to produce expected output. In the MDS/PASCAL environment a particularly useful development tool was available for this type of debugging, known as PSCOPE-86. This affords the user such facilities as inserting breakpoints, single-stepping, high-level code patches, access to program symbols, etc. PSCOPE was used extensively during this phase of testing.

In general, unit tests are designed and carried out by the programmer responsible for the unit. However, in the case of one of the functional processes (RUI), an extensive set of test-cases was designed by an independent party. This was considered necessary because this particular process, being a user interface, is in practice likely to be subjected to a wide spectrum of unacceptable human inputs as its main source of stress.

4.4.3 Integration Testing

Integration testing commences when the unit-tested modules are ready to be integrated with one another and tested as a system or part thereof. Inter-process communication errors are sorted out during this phase of testing.

As with unit testing, the integration tests are designed and executed by the programmers. The purpose of this series of tests is to ensure clean interfaces between all RTX processes. They basically consist of putting all the relevant processes together into the RPAD, using an X780 device to feed in messages to the RPAD, examining the resulting messages on the network side, and vice-versa. Programmable line monitors are used to capture and generate messages on the network side.

There are two levels of interfacing problems which arise here.

In the first instance, the four processes specific to RTX (ie IOP, RDM, RTH and RUI) must have clean interfaces. Although the close liaison between members of the project team might appear to militate against errors at this level, testing inevitably brings to light several unforeseen

problem areas. (As a general comment, it is noted that the fact of working in a small team greatly facilitated communication. This is in contrast to previous experience of large project teams liaising mainly through formal meetings and documentation, where the scope for misunderstandings and misinterpretations is increased.)

However, the foregoing processes form only part of the RTX software, the remainder consisting of the pre-existing operating system and utility processes. Because information about the interfaces to these processes is largely documentation-dependent, the scope for misunderstandings widens, and the consequent imperative of thorough testing increases.

4.4.4 Alpha Testing

In the alpha test procedures, the prime purpose is to verify that the implementation as described by the Functional Specification operates correctly. This means that all ' normal' modes of operation described by the specification should be exercised. Where the Functional Specification makes claims about dealing with abnormal conditions, tests for these conditions should also be devised.

The general approach is to try to identify so-called equivalence partitions [10] , each of which represents a set of scenarios which are roughly equivalent, such that if one scenario (or a representative sample of scenarios) from the equivalence partition has been tested, then others within this partition are likely to work as well. In deciding on sample choices to represent an equivalence partition, an attempt is made to incorporate the notion of boundary-value analysis - ie to choose scenarios which are in some sense ' at the fringes' of the equivalence partition [10] .

Ideally the alpha tests should be designed by an independent party not directly involved in the project. Since no such individual could be found, the design was left to a team-member who had not been heavily involved in programming or in unit and integration testing. An alpha test plan was set up, designed to test such aspects as user services, BSC interface, 3305 protocol support, network control and management, and the stress limits of the PAD. For each test in the alpha test plan, the purpose of each test was outlined, the recommended test procedure given, and the expected results of the test were stated.

The test environment should ideally allow for maximally configured RPADS to communicate through an X.25 network, using each of the members in the X780 family of devices. Monitoring and logging facilities should be available to analyse traffic at the external (BSC and X.25) interfaces, together with mechanisms to generate and/or delay test traffic at these points. Furthermore, it is desirable to have facilities for the easy re-configuration and swapping of equipment.

Since the actual facilities available to the project team fell short of these requirements, a number of improvisations had to be made to approximate the required test environment.

4.4.5 Acceptance Testing

This level of testing may not be required in a conventional development environment. However, in the present case, the product was to be marketed and maintained by the client. The latter party therefore felt the need to test the product at this stage. Furthermore, it had not been possible to carry out the complete set of alpha tests due to lack of local facilities. Hence, it was recommended that the alpha tests be completed at the client's site as part of the acceptance test procedures.

4.4.6 Beta Testing

The purpose of the beta tests is to test the product in an operational environment. The tests should ideally be performed by a potential client, with only minimal assistance from the project team members. This affords the opportunity of, inter alia, verifying that the user and system manuals are clear and comprehensible.

5. DOCUMENTATION

Approximately half of the time spent on the entire development effort was devoted to documentation. Thrashing out requirements with the client and drawing up an acceptable Requirements Specification are never easy tasks. Though fundamental to the usefulness of the developed end product, these tasks are often sadly neglected. Preceding any design, the functionality of the end product was reviewed and approved by both the client and the developers; another time-consuming but essential effort.

In any development effort of substantial size, it is imperative that the design is done soundly, and is divorced from syntactical complexities during the coding effort. Additionally, if the designers delegate the coding effort partly to assistants, the consistency of the design documentation is of fundamental importance for the duration of coding.

The hierarchy of development documents includes:

- Requirements specification (14 pages): A thin, factual document listing the features to be supported by the end product.
- Functional specification (166 pages): Detailed description of the features supported and functionality of the product. This document can be used as a basis for an operation manual (if required).
- Design specification (292 pages): Bulky, technically detailed document intended at the development and maintenance staff only. It details the implementation of each function as put forward in the functional specification. There is one global design specification for the entire project.
- Program specifications (RDM 154 pages, RTH 182 pages, RUI 57 pages): One document per process, detailing its implementation to the level of specification language. It contains all information required for the coding and maintenance of the process.

Although all the processes were initially largely developed in specification language, this proved to have served its purpose once coding was completed. It seemed unreasonable to maintain the specification language in the Program Specification document after coding and testing were completed. Well commented code alone should assist maintenance sufficiently and should be seen as part of the final Program Specification.

In addition to the above hierarchy of design documents, test specifications are supplied, listing the functions required of the product and test scenarios and data required for the testing of each function. A test specification was drawn up for each of the alpha and beta tests. Ultimately, these test specifications should be drawn up by an independent party, so as to ensure unbiased testing procedures.

Internally, during the development effort, loose points were documented as encountered (and distributed among the members of the team) using a set of design notes. This is a convenient way of unambiguously documenting changes affecting the design.

6. CONCLUSION

The most prominent conclusion, when thinking about a project like the RTX development, is that the size of the project was grossly underestimated.

The full development involved approximately 18 months, nearing two years if preparatory work preceding the official start of the project is taken into consideration. A full-time staff of three people were involved, with help from various other people only involved on a part-time basis for short durations of time during the development.

The initial discussion and assimilation of requirements, the preparation of the functional specification, the design and documentation of the design took approximately half of the overall time invested in the project. This was, however, time and effort well invested and paid off in the long run during the development. The design was clean, sound and unambiguous. Complex coding issues could be referred to the Design Specification without the need to have the document updated at any time during the coding and testing phases. It is of fundamental importance that the design issues of a large project should be divorced from the coding issues. Each in itself is

complex enough to require the full concentration of the person involved at any specific time.

Complete and consistent documentation assisted in developing solid, reliable and well structured (maintainable) code which performed well under test conditions. It also contributed substantially to making accurate estimates during the project for the phases yet to be completed, and the meeting of set deadlines.

As an indication of the effort involved, the number of pages per document was given in the previous section. The final code size (in bytes object code) per process was:

- RDM: 14337
- RTH: 19975
- RUI: 8106

The size of the resident software is given by:

- IOP: 10633
- XPH (X.25 packet level): 12607
- PMR: 13392

The advantages of a small team (three people) were considerable in terms of (i) time saved because no scheduled meetings were necessary, (ii) effective and instant communication of design issues and/or changes, and (iii) scheduling of the development and testing resources for various individual needs during initial coding and unit testing phases.

One handicap of the development effort was the single-user MDS. This system was heavily used during the coding, compilation and unit testing phases (before the code was transferred to the target hardware) and tight time schedules resulted. Furthermore, the development operating system (ISIS on the MDS) and the operating system of the target hardware (CP/M during loading stages) differed. This resulted in some transfer incompatibilities and tools facilitating this transfer were developed.

7. REFERENCES

1. Bochman G. & Sunshine C., Formal Methods in Communication Protocol Design, *IEEE Transactions on Communications*, Vol. COM-28, 4, April 1980, pp 624-631.
2. Gane C. & Sarson T., Structured Systems Analysis: Tools and Techniques, Englewood Cliffs, New Jersey: Prentice Hall, 1979.
3. Polak W., An Exercise in Automatic Program Verification, *IEEE Transactions on Software*, **SE-5**, No. 5, September 1979, pp 453-457.
4. GTE TELENET, A Packet Assembly/Disassembly Protocol Specification for Binary Synchronous Communications (BPAD/3305), April 1980.
5. ISO Information processing Systems - Open Systems Interconnection - LOTOS - A Formal description Technique Based on the Temporal Ordering of Observational Behaviour. March 1985.
6. Kourie D.G., A Partial RJE Pad Specification to Illustrate LOTOS. Presented at the 1985 Computer Science Lecturers' Association Conference (to appear in *QI*).
7. Day J.D. & Zimmerman H., The OSI Reference Model, *Proceedings of the IEEE*, **71**, No. 12, December 1983, pp 1334-1340.
8. Rudin H., An Introduction to Automated Protocol Validation, *Proceedings of the IFIP Conference on Data Communications*, September 1982.
9. Vissers C.A., Tenney R.L. & Bochman G.V., Formal Description Techniques, Proceedings of the IEEE, **71**, No. 12, December 1983, pp 1356-1364.
10. Myers G.J., The Art of Software Testing, New York: John Wiley & sons, 1979.