Teleprocessing Monitors and Program Structure

Roger Clarke*

This article discusses teleprocessing monitors, their impact on program-structure, and the development of such programs using the DELTA program-generator package.

Key words and phrases: DELTA, online programming, program-generators, program-structure, teleprocessing monitors, transaction processing.
CR categories: 4.12, 4.22, 4.31.

SCOPE OF THE ARTICLE

A previous article dealt with the preparation of 'self-contained' online programs using the program-generator package DELTA (Clarke, 1982b). These were defined as dialogue programs running under an operating system specifically designed for online or mixed online/batch operation. In such an environment terminal-communication is performed using inbuilt commands such as the ANSI COBOL SEND and RECEIVE verbs or extensions to the DISPLAY and ACCEPT verbs; or by CALLs to one or more special subprograms which perform the physical data transfer and then return control to the calling program in the normal manner.

This article deals with online programs of another type, those which run as subprograms to a so-called TP (TeleProcessing) Monitor or DC (Data Communications) Monitor. Commercial products of this kind include IBM CICS and the DC part of IMS/DC, the CINCOM product for IBM and IBM-compatible systems ENVIRON-1, Univac's TIP/CMS, Honeywell's TDS and Burroughs GEMCOS. The short form 'TPM' will be used in this article, and the term 'TP Program' will refer to a program running under a TPM.

The article will discuss the methods used to generate TP Programs. In order to do this however some features of TPMs must be first discussed. As the author has found few references which discuss TPMs in a suitable and supplier neutral manner, this preliminary discussion takes up a considerable part of the article.

MAIN MEMORY UTILISATION

A large installation is interacting with hundreds of active terminals. Each requires space in memory for the program it is communicating with, hence potentially vast amounts of main memory are demanded. Effective management of main memory is discussed here in isolation from the many other factors traded off by a multi-tasking operating system.

A first measure to save space is to enable a program to service more than one terminal. This is referred to as 'multi-threading', since each task 'threads' its way through the maze independently from, and unaffected by the others. This is only possible if the program is in 'reentrant' form, i.e. if all data that may vary is stored in a 'variant' or 'data' segment. The procedural segment need exist only once in main memory and is referred to as 'shared code'; a data segment must exist for each active program. A COBOL analogue to a reentrant program is a program in which no variables are defined in the Working-Storage Section (only in the Linkage Section), and Commands which result in the generation of working areas are avoided – CALL, PERFORM . . . VARYING and ALTER being the main candidates (PDV, 1979).

Another contribution is made by a virtual-storage paging arrangement in which procedural and data segments not currently in use may be unloaded onto secondary (drum/disc) storage and reloaded when next required. This involves operating system overhead, but due to the enormous speed of processors and main memory (of the order of 10^{-6} seconds) compared with secondary storage (2×10^{-2}) and especially terminal operators (1 to 10^2 seconds – human real-time), considerable net savings can be achieved. DP practitioners are recommended to Keedy (1980) as a reference on virtual memory.

Yet another step is to reduce both the size of individual programs and the amount of processing squandered in administering paged-out segments. This is achieved by requiring programs to 'die' immediately after communicating with the Operating System, rather than merely being suspended pending the arrival of the next input. It can be argued that even batch programs should be organised in this way, although with most batch input coming from secondary storage the gains are likely to be far less than in the case of online programs whose input comes from a relatively very slow human being.

Against this potential gain must be balanced the size of the additional OS subsystem (the TPM) necessary to administer these very short-lived programs, plus the additional main and secondary memory management to enable programs to pass common data to their successor or to themselves in their forthcoming reincarnation. The net effect can be (but not necessarily is) a considerable gain. This is particularly so in the case of the originally batch oriented OS which have had terminal-handling and virtual/paging facilities progressively tacked on. Such (predominantly 'mainframe') OS are consequently somewhat less

*Xamax Consultancy Pty. Ltd., 20 Leichhardt St, Leichhardt, NSW, 2040. Manuscript received January 1982, revised September 1982.

efficient in managing main memory, and the benefits shown by ephemeral programs appears correspondingly greater.

A primary factor stimulating the development of TPMs was the conservation of scarce main memory resources but a range of other factors were involved.

OTHER FACTORS LEADING TO TP-MONITORS

Centralised Terminal-Handling

The varying physical characteristics of terminals can be catered for by a central subsystem, enabling application programs to deal with a standardised 'logical' interface. This is really an argument for a Terminal Control Program (TCP) and is equally applicable to environments in which online programs are self-contained.

Centralised Formal Editting of Input Data

'Automatic' checking of the appropriateness of data in each field can be performed by the TPM (or for that matter by the TCP).

Optimised Data Stream Transmission

It can be important to keep line traffic to a minimum, especially in highly dispersed networks. A significant contribution can be made by a routine (be it in the OS, the TCP or in the TPM), which compares the desired screen image with that currently displayed and transmits the minimum data needed to effect the change. (Reducing terminal-to-processor data-flow is of course less easily achieved, requiring intelligence and storage in the terminal itself.)

Supply of Preprocessed Data Streams to the Application Program

The simplest approach is to provide the program with the current contents of the screen, irrespective of what was received in the most recent transmission. An improved service might be to supply, in addition to the data stream, tables showing which fields were changed, which remain unkeyed, which contain data inappropriate to the field definition, etc. This can be provided equally well in a TCP as in a TPM, since it is closely bound to the physical characteristics of the device.

Centralised Control over Control Flow

Software must decide which program is required to service the input from the terminal. This function can be built into the TPM. It can be performed equally well by a (relatively tiny) table-driven menu-handler embedded in the OS; or by a user-written master program which administers the menu displays and arranges for the chosen program to be run immediately after its own demise (commonly referred to as 'Call-Next-Program'). Each application program running in such an environment must admittedly comply with the requirement that it pass control back to the master program on completion, but as will be seen, a similar discipline is required with a TPM.

From the above it will be clear that these factors, important as they are, justify the creation of a Terminal Control Program to administer the link between application program and terminal, but not the conception, realisation and implementation of the altogether more complex TP-Monitor. The real reason for TP-Monitors must be sought elsewhere.

TRANSACTION-ORIENTED PROGRAMMING

It can be argued that the 'natural' way to study organisations, and to 'objectively' document their present and intended functions, is to identify 'work steps' or organisational transactions. If so (and the issue remains unresolved) then the 'natural' form that programs should take is also transaction-oriented. The argument isn't just one of structural elegance: much software development activity is presently invested in the translation of design information from one form into another, hence great savings could be made if all stages of the production line were to acknowledge the same methodological framework.

In such an environment we would then redefine the role of information-processing services as the recording and processing of organisational transactions, each transaction being triggered by an organisational 'event'. This brings commercial processing much closer in concept to process control or 'real-time' processing.

Batch Processing is then seen as an alternative means for handling large-volume or low-priority work. High-volume, long lead-time tasks need to be performed 'asynchronously' with respect to the terminal, such that the end-user can himself schedule them, but without blocking his own terminal. There is then no reason why batch (terminal-asynchronous) jobs cannot run in a transaction-oriented manner, under the same Monitor as online (terminal-synchronous) tasks.

This is very convenient when an application comprises sub-functions that are to be performed in either and both online and batch modes. One example is a sub-function like the look-up of article price and article discount (often dependent on a range of attributes of customer, article and order), which may be needed in the online program 'Urgent Quotations' and in the batch program 'Low-Priority Invoices'. Another is recovery forward from a checkpoint, when a perhaps large number of events that were originally handled synchronously are to be reprocessed in asynchronous mode.

DEFINITION OF A TPM

Much of the literature on the subject comprises the reference material of the various suppliers of commercial products. These are (quite justifiably) biased towards the specifics of their own marketing strategy, host Operating System(s), etc. See however Mills 1972, Davenport 1974, KDCS 1978, PDV 1979 and Datapro 1979a and 1979b.

A working definition is suggested as follows:

A Teleprocessing Monitor is a subsystem of an Operating System, which administers the logical and perhaps also the physical link between each terminal and the program(s) invoked to perform tasks initiated by that terminal.

The term 'logical link' refers to communication with an idealised terminal, independent of the physical characteristics of whatever terminal is physically on the other end of the wire. The physical interfacing tasks therefore include code conversion, synchronisation of physical data flows and line-control/protocol-handling to the extent that this is not performed by a communications network monitor. As indicated in Figure 1, these functions may be embedded in the TPM or delegated to a Terminal Control module.

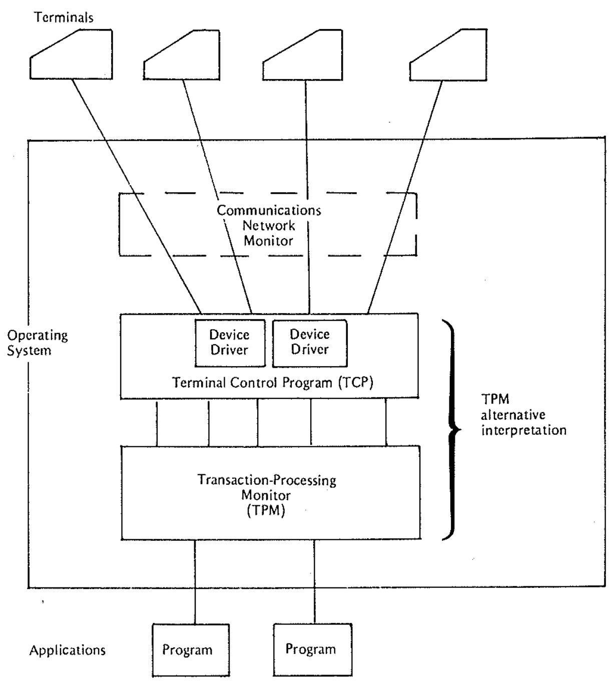
Terminals
Communications Network Monitor
Operating System
Device Driver
Device Driver
Terminal Control Program (TCP)
TPM alternative interpretation
Transaction-Processing Monitor (TPM)
Applications
Program
Program

Figure 1: A TPM and its Environment.

FUNCTIONS OF A TP-MONITOR

These are grouped according to whether or not they are assessed by the author as being essential to the above definition.

Essential Functions

Administration of I/O-Data

— Data streams from terminals.
— Data streams to terminals.

Administration of Control Flow

— Selection of the next program.
— Passing control to that program.
— Receiving control back from that program.

Administration of Working Data

— Making data available to programs.
— Receiving data from programs.

Additional Functions

Physical terminal-handling

Embedding TCP within TPM.

Interface with Permanent Data

Special commands or calls may provide the application program with an improved and/or standardised interface with data files (sequential, direct, indexed, multi-indexed, tape, etc) or database. If the DBMS software is integrated with the TPM, the resulting conglomerate is referred to as a DB/DC Monitor. Some TPMs go so far as to preclude the use of standard file-handling commands.

Starting of Asynchronous (Batch) Jobs

Screen-Definition Facilities

Utilities may be associated with the TPM to support the specification and assembly of program-independent masks. The term 'Forms Processor' is also used. At one level the preparation of the physical layout may be supported ('screen painting' or 'forms editing'); at another the specification of mask and field attributes.

Logfiles, Checkpoints, Restart/Recovery

Accounting

News Broadcasts and Terminal/Terminal Communications

Security

e.g. User-Authorisation Checking.

Testing Facilities

A further function that can be performed by a TPM relates to control-passing between the application-programs.

PASSING OF CONTROL WITHIN A TP-MONITOR APPLICATION

The simplest possible arrangement is that the TP-Monitor decides on the basis of some field in the input data stream which program is to be called. That program passes control to the TPM when it terminates, together with a data stream for transmission back to the terminal. This data stream may contain the name of the program which is to be invoked when the operator next transmits

If the field is unprotected then the operator will be able to override the default next-program call. It may be necessary to key the whole program number in, or, more conveniently a transaction-identification, function-key, or selection-number or mnemonic which is converted into the program name by table-lookup within the TPM. In this way the operator is able to remain oblivious to program initiation and termination.

The whole of the processing may in principle be placed in one large program. More realistically it may be sub-divided into various subprograms (see Figure 2a). With

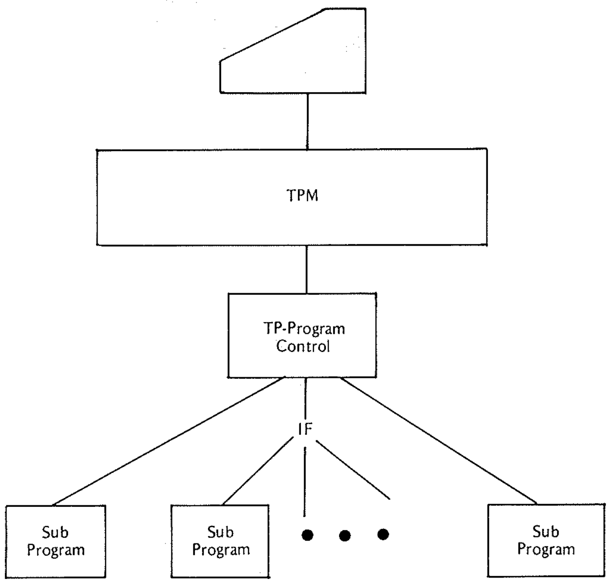

Figure 2a: Control-Passing using CALL.

complex processing main-memory requirements may still become too large, as the control program will remain in memory at the same time as the invoked processing-code.

As an alternative to the CALL mechanism, the TPM can provide the means for transferring control, as shown in Figure 2b. A chain of control is passed along, with the 'terminal' programs having contact with the screen, the processing programs only with other programs, the TPM and the permanent data. The advantages are that some main memory overhead is avoided and that only one type of data-passing is involved i.e. 'messages' to and from the TPM. A major disadvantage is that the application's structure is scattered throughout the entire application.

A further alternative has been provided by some TPM suppliers who support a 'pseudo-conversational' mode. This enables the program to be written in a self-contained form (see Clarke 1982b) with control decisions embedded within the program. The program then commences with a decision as to where within the processing it should commence (in COBOL terminology a GO TO DEPENDING; there is arguably no equivalent construct recognised by structured programming theory). The program increases in size, but this is of little consequence under an Operating System which supports virtual memory.

STRUCTURE OF TP PROGRAMS

The effects of a TP-environment upon program structure are considerable. In Figure 3a is shown a simple 'logical program' as it might be designed using Jackson's Program Design Method (Jackson, 1981). It could be implemented in that form as a self-contained program (see Clarke 1982b).

In Figure 3b the same logical program is shown as it might appear under a TP-Monitor. The first difference is that there are several 'physical programs' required to implement the single 'logical program'.

A more critical difference is that each (physical) program commences with the receipt of a message and concludes with the sending of a message. A way to describe the relationship between the two structures in Figures 3a and 3b, is to say that the first has been 'inverted' with respect to its driving file to produce the second.

Each 'TP-Program' runs as a subprogram to the TPM. Data may be transferred between TPM and TP-Program via the LINKAGE SECTION and/or via additional facilities depending on the particular TP Monitor. This article focuses on control-structure rather than message-passing.

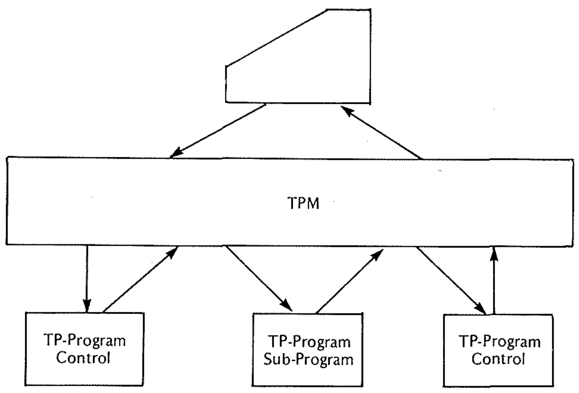

Figure 2b: Control-Passing via TPM.

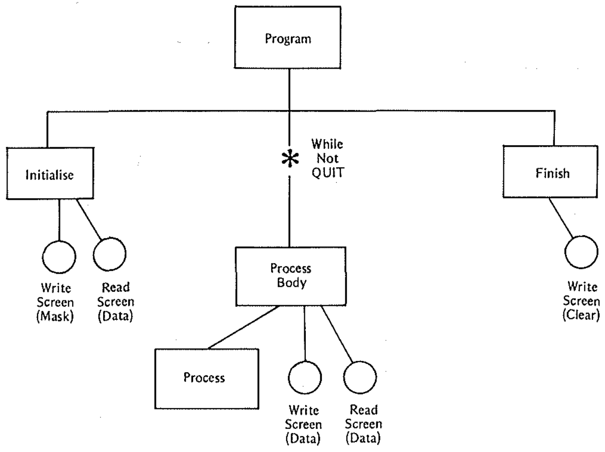

Figure 3a: Logical Structure of a Simple Self-Contained On-Line Program.

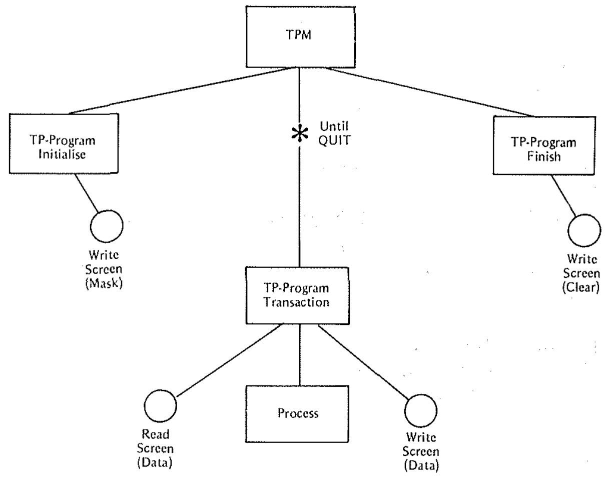

Figure 3b: Logical Structure of a Simple Transaction-Oriented On-Line Program.

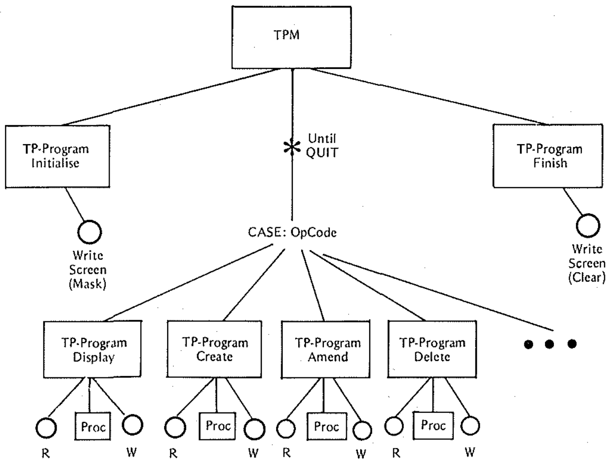

Figure 3c: Logical Structure of a Less Simple Transaction-Oriented On-Line Program.

A single 'logical program' may do more than merely, say, display a record on the screen. It may also, depending (commonly) upon an Operation Code, create, amend or delete a data record. There would be then a Selection Construct within the Processing Body, and the flow of control between the various 'physical programs' making up the 'logical program' quickly becomes tortuous.

There are three ways to handle the decision-making:

- embed it within the TP-Monitor (requiring a language powerful enough to express the various possibilities). Figure 3c illustrates this;
- embed it within a control program which remains memory-resident, invoking the chosen functions as subprograms (Figure 2a);
- perform the decision-making within a control program which passes control via the TPM to the chosen program (Figure 2b).

Where all decision-making and routing for a single logical program is performed by a control program, a means is required for identifying the current context in an accurate and efficient manner. State Transition Tables are an appropriate device (see for example Juliff 1980, Petereit 1980, Hext 1982). This article will not deal in any detail with this approach.

Two classes of TP Program can be identified. Those which exchange messages not only with other programs but also with the terminal are referred to here as Terminal-Handling Programs. Those which communicate exclusively with other programs perform a strictly processing function and are referred to here as Data-Processing Programs. With a little care there is no reason why the latter should not be able to be invoked alike by terminal-synchronous (online) and terminal-asynchronous (batch) tasks. The use of TP-Monitors to control batch processing is not further discussed in this article.

A general structure is suggested in Figure 4, sufficient to cater for most eventualities. The possible processing functions (not all of which need necessarily be relevant to any given program) are enveloped by functions catering for the communications from and to the TPM. The actual implementation of these communication functions depends on the particular software environment.

On the basis of this general structure a set of macros will be discussed which provides close support to the programming phase. First a brief introduction to program generators is in order.

PROGRAM GENERATORS IN GENERAL

An assembler converts a source file directly into executable form. A compiler deals with a source file differently organised and sequenced from the object code it is to create. A program-generator differs from them in the following ways:

- the source-code is function- rather than procedure-oriented;
- its output may be a high-level language, for input to a compiler. Early versions of compilers often used such a two-step technique by outputing an assembler program. In the case of program generators, however, this is not necessarily a temporary measure, as it caters for multiple incompatible target compilers;
- as with the more modern assemblers and some compilable languages, it commonly includes a macro-language and processor, such that the source-language is user-extensible.

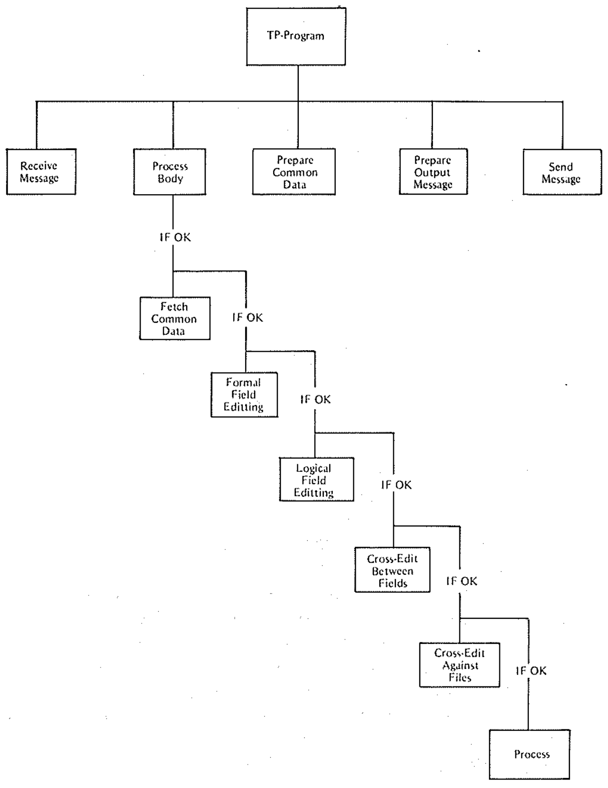

Figure 4: A generalised structure for TP-Programs.

Few satisfactory products are marketed. Most are specific to particular machines, e.g. MANTIS and UFO under particular IBM operating systems, NoCode from General Automation, LINC from Burroughs NZ, and the cutely-named 'The Last One', a UK product generating Basic in CP/M and UNIX environments. Philip's PET/MAESTRO development-machine incorporates generator functions. Clarke (1982a) provides an introduction to the topic.

A powerful generator package with which the author is familiar is independent of both its host software environment and its target environments. DELTA is a Swiss product, marketed since 1976 in German-speaking areas and since 1980 also in the UK and Australia, with some 150 installations to date.

ONLINE PROGRAMS USING DELTA

The primary objectives of the development phase (quick and cheap development, a clean product, portability and low maintenance and enhancement costs and lead-times) can be readily supported by DELTA together with some customised macros.

Several developments in Germany and Switzerland, notably at Systema GmbH, Mannheim (Clemens 1981, but see also Ahrens 1981 and Thurner 1981) have used DELTA in a context of control programs using State Transition Tables. The structure suggested in Figure 4 requires extension to serve the purpose of such control programs, but because of its relative simplicity will be used below to illustrate the use of the DELTA tool-kit.

In addition to the facilities provided by the product itself, a set of macros is required, to generate from a short list of parameters the appropriate program shell, the internal decision-making structure, and the communications with both the TPM and the permanent-data handling-environment. The actual processing can be coded in COBOL or PL/1, or where portability is important, exclusively in invocations of DELTA macros.

The invocation of the program structure can be nested within the generation of the basic program shell:

```
.*------------------------------
. PROG-progname, AUTHOR=xxxx, DATE-WRITTEN=xx/xx/xx,–
.              TYPE=(DPP, COMMON=NO, FORMAL=NO,
                   XEDIT-FILE=NO),–
.              MASK=xxxx,–
.              MSG=(xxxx,xxxx,xxxx, . . .)
.*------------------------------
```

The Keyword-Parameter 'TYPE' controls the class of program to be generated (Terminal Handling will be in this case excluded), and particular sub-functions can be selected out (or if preferred, selected in), with a default-list applicable. The list shown above would exclude the sub-functions 'Fetch Common Data', 'Formal Field-Editting' and 'Cross-Editting Against Reference-Files'. The Keyword-Parameters MASK and MSG define those screen-related data areas and TPM-communication records that are to be invoked from the Data Dictionary.

The other sub-functions are made available, and one or more 'Locations' defined in each, to enable the processing code to be inserted. These Locations are named EDIT-FLD, XEDIT-FLDS and PROCESS. The code would be inserted in the following form:

```
.*------------------------------
.SL=EDIT-FLD
.ADD TESTRANGE, CUST-DISCOUNT, (0,6)   , 417
.ADD TESTLIST,    CUST-SLSZONE,  (1,4,5,6), 418
.*------------------------------
.SL=XEDIT-FLDS
.ADD TESTEXCL, (CUST-DISCOUNT NOT = ZERO) , –
.              (CUST-SLSZONE = 6) , –
.              419
.*------------------------------
.SL=PROCESS
.ADD MOVE, CUST, xxxx, (NAME, ADDRESS1, ADDRESS2,
       POSTCD)
.ADD LASTUPDAT, CUST
.ADD PUT, NEW, CUST
.ADD PREPLOG, CUST
.ADD PUT, APPEND, LOG
.ADD PREPMASK, (MSG = 'CUSTOMER CREATED')
.*------------------------------
```

The processing code is generated by the minor macros invoked here. In each case the interfaces to error handling and exception-processing functions are generated automatically.

In practice even deeply-nested sub-functions can contain low level selection- and iteration-constructs in addition to the sequential processing of the above example. Of the several DELTA tools available for this task the 'pseudo-code' interpreter SPP is the most suitable.

The handling of logical records, and the generation of physical file and record definitions and handling are well established DELTA techniques and tools. In this case the following would suffice to specify the links to the Data Dictionary:

```
.*------------------------------
.ADD LR-CUST, UPDATE-ONPLACE
.ADD LR-LOG,  EXTEND
.*------------------------------
```

The program end is signified, together with the field-name which contains the Next-Program name:

```
.*------------------------------
.ADD PROGEND, (NEXT=xxxx)
.*------------------------------
```

The above approach is generalised and simplified. Nonetheless the experience of Systema GmbH with a more powerful model is relevant: during 1980/81 typical program-modules required 25-40 lines of specification at the design stage and 150-250 lines of coding. The (COBOL) code generated was of the order of 1500-2500 lines.

OVERVIEW OF THE MACRO STRUCTURE

The design and construction effort to provide such a macro-set is not small. It is important to identify the different levels of abstraction, and to recognise those nested functions which may vary, if only subtly, between one program and another. These should be implemented in separate macros if flexibility is to be maintained.

Three broad levels of abstraction are useful (although the classes are clearly not disjunct):

– the programmer interface;
– nested macros to provide program flow control and the logical interfaces;
– deeply-nested macros to handle the physical interfaces.

The list below illustrates what is included in each of these levels. Low-level macros can themselves invoke further macros.

Programmer Interface	Logical Level	Physical Level
PROG	THP/DPP (program-structure)	
	RC-TAB-ANSI	RC-TAB-SPEC
TESTRANGE	ERRMSG	PHERRMSG
TESTLIST	ERRMSG	PHERRMSG
TESTEXCL	ERRMSG	PHERRMSG
LASTUPDDAT		MOVE
PUT		
PREPLOG		MOVE
PREPMASK		MOVE
LR-CUST	LR-ROUTINES	FD-CUST ISFILE
		ISREC
		PR-CUST-1
		PR-CUST-2
PROGEND	SELNPROG	PHSELNPROG

CONCLUSION

Transaction-oriented on-line programs exhibit a variety of forms, depending on the particular TP-Monitor in use, and the using organisation's experience and philosophy.

Design and development of such programs using an advanced program generator such as DELTA enables standardisation of methods of working, savings in development and maintenance, improved planning and control, and portability of product.

ACKNOWLEDGEMENT

Die Hilfe von verschiedenen Kollegen im deutschsprachlichen Raum wird dankend anerkannt, insbesondere Hrn Karl-Heinz Clemens und Hrn Dr Rainer Petereit von der Firma Systema GmbH, Mannheim.

REFERENCES

AHRENS, Klaus-D. (1981): 'Eine Methode ist nur so gut als das Werkzeug' article-series, 'Computer-Woche Deutschland', April-May 1981.

CLARKE, Roger (1982a): 'A Background to Program-Generators for Commercial Applications', *Austral. Comput. J.*, 14, 2, May 1982.

CLARKE, Roger (1982b): 'Generating Self-Contained On-Line Programs Using DELTA', *Proc 9th Australian Computer Conference*, Hobart, August 1982.

CLEMENS, Karl-H. (1981): 'Workshop: DB/DC mittels DELTA', Course Notes, Systema GmbH, Mannheim, Germany 1981.

DATAPRO (1979a): 'A Buyer's Guide to Data Communications Monitors', Reports on Data Communications C15-010-101 to 110 *Datapro*, May 1979.

DATAPRO (1979b): 'Teleprocessing — The Modern Marriage of Computers and Communications', Solutions: Communications CS10-150-101 to 108, *Datapro*, June 1979.

DAVENPORT, R.A. (1974): 'Design of Transaction-Oriented Systems Employing a Transaction Monitor', *Proc ACM*, 1974, 222-230.

HEXT, J.B. (1982): 'State Transition Tables', *Austral. Comput. J.*, 14, 1 February 1982.

JACKSON, M. (1981): 'An Introduction to Jackson Structured Programming and Jackson System Development', South-West Universities Regional Computer Centre, Bath, 1981.

JULIFF, Peter (1980): 'Program Control by State Transition Tables', *Austral. Comput. J.*, 12, 4, November 1980.

KDCS (1978), 'KDCS-Benutzerhandbuch', Bundesminister des Innern, Bonn, Germany, 1978 (Kompatibler Daten-Kommunikations System).

KEEDY, J.L. (1980): 'Virtual Memory', *Austral. Comput. J.*, 12, 2, May 1980.

MILLS, D.L. (1972): 'Communication Software', *Proc IEEE*, 60: 11, November 1972.

PETEREIT Rainer (1980): 'Architecktur und Entwurf moderner kommerzieller Software'. *Proceedings of the International Congress for Data Processing*, AMK, Berlin 1980.

PDV (1979): 'COSMOS: Allgemeine Beschreibung', PDV Unternehmensberatung fuer Datenverarbeitung GmbH, Hamburg, Germany 1979.

THURNER, Reinhold (1981), 'Softwareentwicklung heute', Article-series Sysdata (Germany) October 1980 — April 1981.

BIBLIOGRAPHICAL NOTE

Details were published in the May 1982 issue of this Journal. Since then the author has returned to Australia. He is acting as a consultant, primarily in the field of commercial software development. He is associated with Software Solutions Pty Ltd, for whom he manages the program generator package DELTA.