Structured Analysis and Design in the Redesign of a Terminal and Serial Printer Driver

The project team felt that the objectives could not be met with a traditional design approach. Structured analysis with real-time extensions and structured design provided an effective alternative.

by Catherine L. Kilcrease

This paper describes the use of structured analysis with real-time extensions and structured design in the redesign of the terminal and serial printer driver for the MPE/iX operating system on the HP 3000 computer system. The redesign project objectives were to:

- Maintain the current block mode performance (the main mode of data transfer for terminal I/O is to transfer characters in blocks of data)
- Improve HP 3000 transaction processing performance on industry-standard benchmarks by 5% to 10% through a 20% to 40% reduction in the terminal driver path lengths
- Maintain the current level of functionality
- Produce a high-quality, supportable, and maintainable product.

The project team felt we could not achieve these goals with the then-current development techniques. Object-oriented methods were ruled out because of the performance requirements. We elected to use structured analysis[1] with real-time extensions and structured design.[2]

The Redesign Project

The original driver was based on the terminal driver of the HP 3000 MPE V operating system. During its design, specification of the terminal and printer subsystem was unclear and led to many problems. Since the original driver had added many features since its first release, it was important to have a complete specification of the subsystem to meet the project goals. Structured analysis provided this.

The original driver consists of seven modules that handle the I/O between the HP 3000 file system, the MPE/iX operating system, and the data communication and terminal controller (DTC) (Fig. 1). There are two storage managers. The terminal storage manager provides the interface between HP 3000 file system read and write intrinsic calls and the terminal logical device manager or fast write concat procedure. The serial printer storage manager provides the interface between HP 3000 file system write intrinsic calls and the serial printer logical device manager. High-level I/O is the old path between the file system and the logical device managers. It generally handles non-read/write I/O (controls, opens, closes). There are two logical device managers: one for terminals and one for serial printers. The logical device managers transfer data between the user stack and the data communication buffers for reads, writes, and controls. The fast write concat procedure processes writes received from the terminal storage manager and sends them to the terminal and serial printer device manager (it provides a faster write path for terminals). The terminal and serial printer device manager communicates read, write, and control information to the data communication and terminal controller (DTC) through the Avesta Device Control Protocol. The lower interface of this protocol is the flow control manager. The flow control manager provides reliable transport between the HP 3000 and the DTC by implementing a transport protocol called the Avesta Flow Control Protocol. The storage managers and fast write concat procedure are invoked by procedure calls. The interface between the other modules in the driver and the operating system is message-based via MPE/iX ports.

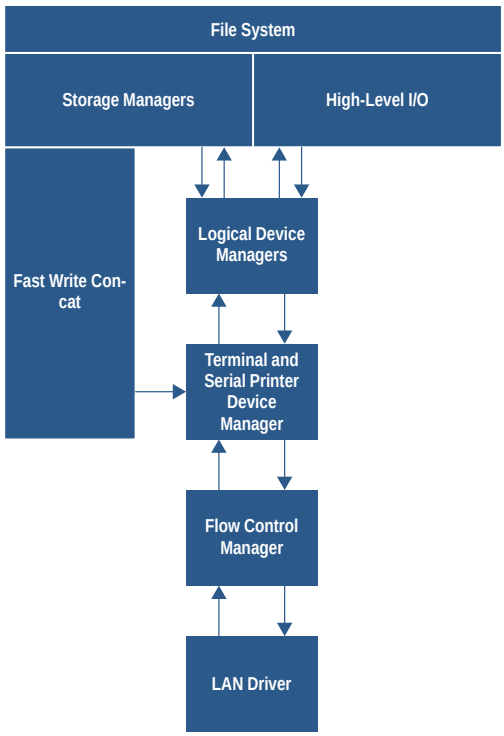
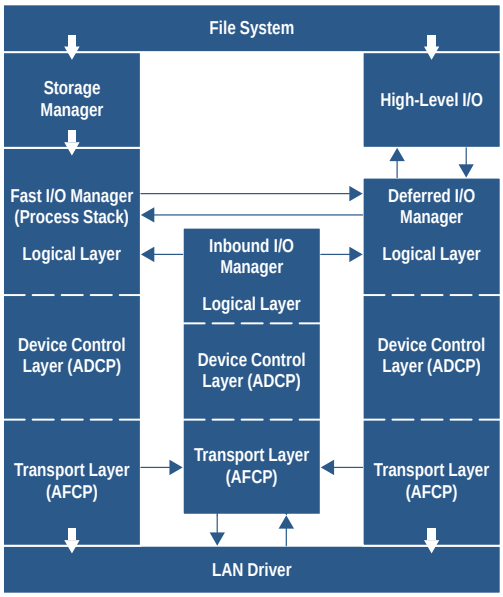


Fig. 1. Original driver architecture.



ADCP = Avesta Device Control Protocol
AFCP = Avesta Flow Control Protocol

Fig. 2. Redesigned driver architecture.

Path trace utility traces of the original driver were analyzed to determine good opportunities for path reduction. The redesign architecture then incorporated the best reduction ideas. The performance improvement is gained from streamlining the path for the most common I/O through the use of direct procedure calls, and from a design emphasizing efficient operation.

The new redesigned driver consists of three modules with three layers in each module (Fig. 2). The three layers are logical, device, and transport. The logical layer acquires the resources needed to complete the I/O request and transfers the data from (to) the user stack into (out of) a data communication buffer. The device layer handles the Avesta Device Control Protocol, which is the mechanism to communicate with the DTC. The transport layer implements the Avesta Flow Control Protocol (transport protocol) and interfaces with the LAN driver.

The three modules in the redesigned driver are the fast I/O manager, the deferred I/O manager, and the inbound I/O manager. The fast I/O manager is invoked by the file system with a procedure call. It handles the most commonly executed I/O: reads, writes, and terminal controls (e.g., change speed, parity, etc.). It attempts to process each request to completion. If it cannot complete the processing because of the lack of some resource, the fast I/O manager will block until the resource is available. If the I/O cannot be blocked (i.e., it is no-wait I/O), then the fast I/O manager sends the I/O to the deferred I/O manager. If the request cannot be completed because the "window" is closed at the transport level, the request is also sent to the deferred I/O manager. The deferred I/O manager has a message-based interface. It handles deferred requests from the fast I/O manager and I/O requests that are made through the "old" high-level I/O path, such as open, close, and preemptive writes. The inbound I/O manager also has a message-based interface. It receives inbound packets from the LAN driver. If the packet is a reply to an I/O request, the inbound I/O manager sends a message to either the fast I/O manager or the deferred I/O manager, depending on which of the two initiated the request. It also handles asynchronous events.

Software Life Cycle

There was some concern that structured analysis and structured design documents would not fit into documents produced by the product life cycle. With our recently revised life cycle[3] this turned out to not be an issue. Our software product life cycle contains the following phases and produces the following lab documents:

Phase	Method	Document
Proposal		
Investigation		Investigation Report
Development		
Specify	Structured Analysis	External Specification Internal Specification
Design	Structured Design	Internal Design
Integration/Test		Test Plan
Support		
Discontinuance		

The external specification document describes the environment in which the product operates, the functional capabilities of the product, and the details of the product's user interface. The internal specification describes the internal requirements of the system and the internal interfaces between the system components. The internal design contains the complete detailed description of the algorithms and data structures to be used in the implementation of the product. The test plan outlines the types of tests to be used to guarantee the quality of the finished product upon release from the lab.

Training

There are four groups of people who need training in structured analysis: development engineers, inspectors, online and offline support engineers, and maintenance engineers. Training in structured design for the nondevelopment engineers is not necessary. The structured design document components are easy to comprehend. The project team took a class in structured analysis with real-time extensions and structured design at the start of the project during the investigation phase.[4] It would have been helpful to have had the training and some experience with the method before the start of the project. The structured analysis training for the nondevelopment engineers was developed by the project lead during the structured analysis phase before inspection of the internal specification.

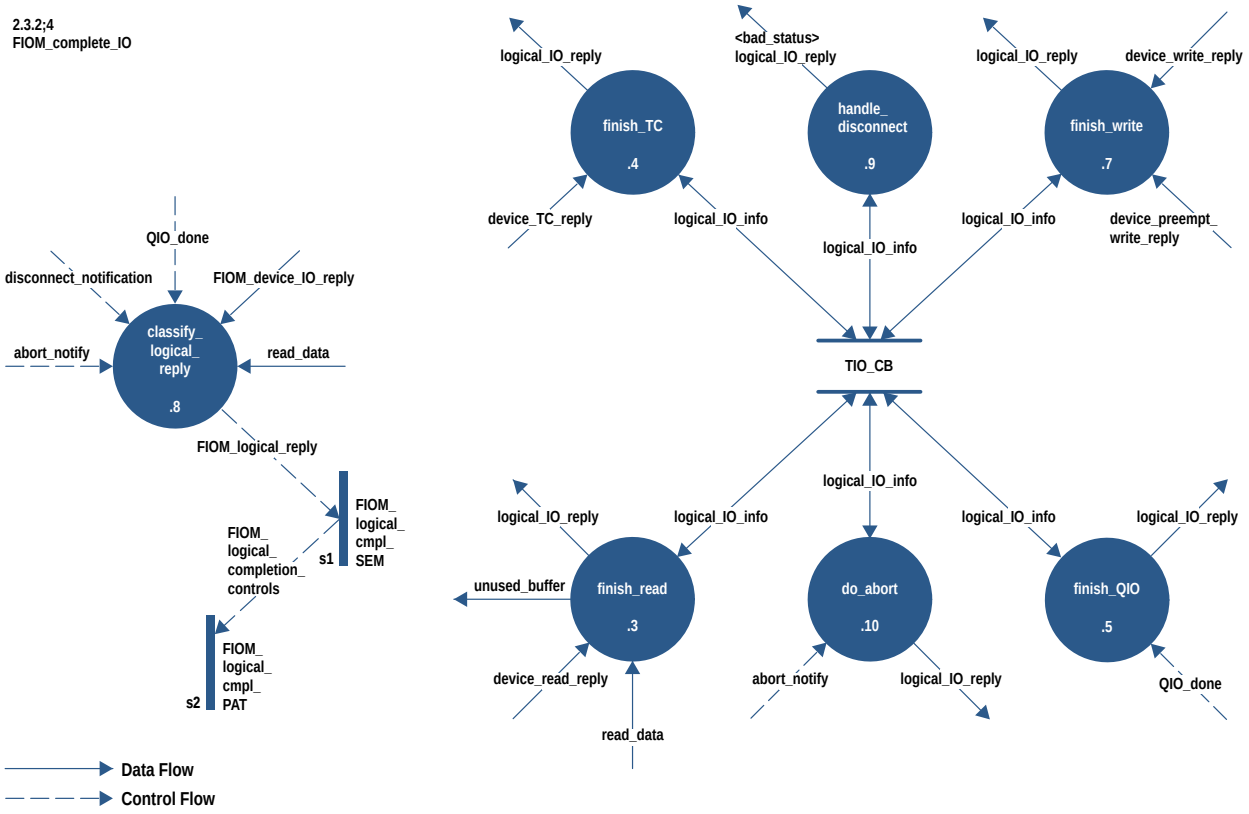


Fig. 3. Data flow diagram.

Structured Analysis Overview

Structured analysis is the use of tools to produce a structured system functional specification. A structured specification is easier to read and understand than the classical textual functional specification because it is graphical and contains many small specifications. The system is broken into small understandable pieces. The tools of structured analysis can be categorized into five functions. The redesign project used an extension of structured analysis for real-time systems. In structured analysis processes are independently data-triggered and infinitely fast (i.e., a process will transform the data when the data is present). Real-time extensions (i.e., the use of control information) allow the system to take other factors or conditions into consideration before enabling or disabling a process.

Function	Tool
Partition the Requirements	Data Flow Diagrams
Describe Logic and Policy	Process Specifications
Show the Flow of Control	Control Flow Diagrams†
Describe Control Processing	Control Specifications†
Track and Evaluate Interfaces	Data Dictionary

The data flow diagrams show the major decomposition of function and the interfaces among the pieces. They show the flow of data, not control. It is the system from the data point of view. Process specifications document the internals of the primitive data flow diagram processes in a rigorous way through the use of structured English, decision tables, or decision trees. They describe the rules of data transformation and the policy, not the implementation. Control flow diagrams share the same characteristics and relationships as data flow diagrams except that they deal with controlling the system. They show the flow of control in the system. A control specification converts input control signals into output control signals or into process controls. It has two roles: one to show how control is processed, and the other to show how processes are controlled (activated or deactivated). The data dictionary is an ordered list of data and control flow names and data and control store names and their definitions. Data flow diagrams and control flow diagrams can be combined together into one diagram.

Figs. 3, 4, and 5 illustrate the components of structured analysis with real-time extensions. Fig. 3 is a combination data flow diagram and control flow diagram. The solid arrows are data flows, the broken arrows are control flows, the solid vertical bars are state matrixes, and the circles are processes. The finish_read process transforms the device_read_reply (indicator that read data is ready) and the read_data (buffer of data input by the user) data flows into the logical_IO_reply data flow using information from logical_IO_info. The freed buffer flows out (unused_buffer data flow). The data dictionary entries for the data flows are:††

```
device_read_reply (data flow) = *read reply from device layer to          *
                                *logical layer. Contains read status,     *
                                *length, and data pointer.                *

                                  status
                                + length
                                + data_pointer
```

† Real-time extensions

†† Here the asterisks indicate comments, the square brackets indicate a choice of one of the enclosed items, the vertical bar means OR, and the plus sign means AND. TC means terminal control, QIO is quiesce I/O (flush outstanding input/output and wait for completion), RID means request identification number, and TIO is terminal I/O.

```
FIOM_device_IO_reply (data flow) =  [ device_read_reply
                                    | device_write_reply
                                    | device_TC_reply
                                    | device_preempt_write_reply
                                    ]

FIOM_logical_reply (control flow) =  [ logical_read_reply
                                     | logical_write_reply
                                     | logical_TC_reply
                                     | logical_QIO_reply
                                     | logical_disconnect
                                     | logical_abort
                                     ]

logical_IO_info (data flow) =  FIOM_IO_pending
                             + FIOM_IO_wait_port
                             + DIOM_IO_pending
                             + logical_RID_pending
                             + logical_abort_RID_pending

logical_io_reply (data flow) =  [ logical_read_reply
                                | logical_QIO_reply
                                | logical_TC_reply
                                ]

read_data (data flow) = buffer_ID
                      + read_status
```

The process specification for finish_read (Fig. 4) describes how data is transformed. Control information is a little trickier to understand. For example, the FIOM_device_IO_reply is transformed into a control flow, FIOM_logical_reply, by the classify_logical_reply process shown in Fig. 3. The control flow enters the state event matrix, FIOM_logical_cmpl_SEM. The state event matrix has memory, that is, it remembers the state of the fast I/O manager. From the FIOM_logical_cmpl_SEM event matrix (Fig. 5), one can see that the finish_read process is activated when the FIOM_logical_reply is a logical_read_reply and the state is read_pending. Empty boxes indicate error conditions.

NAME:
2.3.2.3;3

TITLE:
finish_read

INPUT/OUTPUT:
read_data : data_in
device_read_reply : data_in
unused_buffer : data_out
logical_IO_info : data_inout
logical_IO_reply : data_out

BODY:
transfer data (if any) to destination, doing backspace processing and freeing unused buffers during the transfer;

send logical_IO_reply with status from device_read_reply or read_data msg;

Fig. 4. Process specification for process finish_read.

The Project and Structured Analysis

During the investigation phase, the team considered five different architectures. The final architecture is a refined version of one of them. At the start of the development phase, we needed to start the specification of the system, define the architecture, determine the changes that were needed in the TIO support modules and operating system, and update the investigation report. Determining the changes we needed from the MPE/iX operating system lab and the project that handled the driver configuration modules would have made more sense in the design phase and not the specification phase, but it could not wait until then.

2.3.2-s1;4
FIOM_logical_cmpl_SEM

event / state	FIOM_logical_reply = "logical_read_reply"	FIOM_logical_reply = "logical_write_reply"	FIOM_logical_reply = "logical_TC_reply"	FIOM_logical_reply = "logical_QIO_reply"	FIOM_logical_reply = "logical_disconnect"	FIOM_logical_reply = "logical_abort"
closed						
open_pending						
idle		finish_write/ idle			do_disconnect/ close_pending	do_abort/ idle
read_pending	finish_read/ idle	finish_write/ idle			do_disconnect/ close_pending	do_abort/ idle
QIO_pending				finish_QIO/ idle	do_disconnect/ close_pending	do_abort/ idle
TC_pending			finish_TC/ idle		do_disconnect/ close_pending	do_abort/ idle
close_pending						
close_timer_running						

Note: If an entry is blank, it is an "impossible" condition which should not be encountered due to subqueue restraints, etc. If the condition is hit, error code (not shown) will take appropriate action.

Fig. 5. State event matrix.

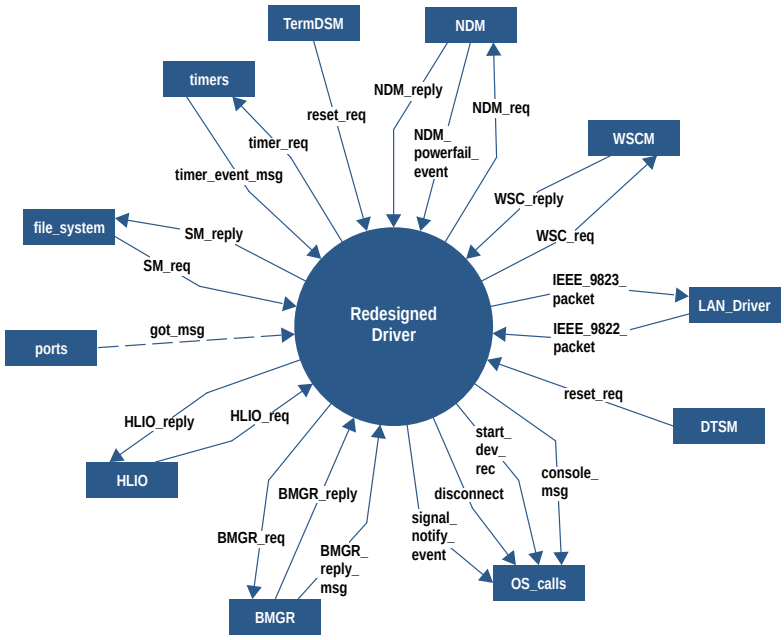


Fig. 6. Context diagram.

We started structured analysis using the fragmentation technique. This technique selects a set of inputs and outputs and creates a fragment model of processes that transform that set of data. The composite model is created by grouping the fragments. We tried to keep the system specification separate from the architecture. We broke the system into parts based on the type of I/O. For example, one fragment modeled the read path. It specified what happened with read requests that were processed completely without blocking for a resource, read requests that blocked for a resource (such as a datacomm buffer), and read requests that could not be processed because of a lack of a resource but could not block (no-wait read). The last type of read request could not be processed in a procedure call environment, but needed to be handled in a message-based environment (deferred I/O manager) so that the user process could continue running even though the read request had not been completely processed.

It became increasingly clear when we tried to tie the structured analysis fragments together that not taking the architecture into consideration was a problem. The goal of the redesign was to improve the performance while maintaining the same level of functionality. We had captured the functionality of the driver in our fragments based on type of I/O. However, each fragment contained fast paths (fast in terms of number of instructions—the fast path could block on a resource) and slower paths (required the message-based interface, which is much slower than a procedure call interface). It was difficult to figure out how to combine all the fast paths, which were spread out across many fragments. This is where one major difficulty with structured analysis arose—how to relate the functional specification to the architecture. The architecture had been selected as the best way to achieve the performance goals. We felt that to create the specification without consideration of the architecture would make the design phase more difficult. We stopped structured analysis work for awhile, and concentrated on completing the architecture. Hatley and Pirbhai[5] helped us resolve the architecture-versus-specification dilemma.

Viewing the system from the data point of view carried over into our parallel architecture discussions. At one point, we physically simulated data flowing through the driver using pens and erasers as data and people as modules. This helped us visualize the interface operation and problems that arise from a mixed procedural and message-based environment.

For complex areas, we used existing code wherever possible to derive decision trees and state transition diagrams. As we became more comfortable with structured analysis, we were able to assign work to each member. Material was reviewed and discussed at project meetings.

One aspect of structured analysis is the iterative nature of the method. One makes a first pass at the data flow diagram, and then discards or revises it until satisfied. Once we felt satisfied with our architecture, we set aside our old structured analysis work and started again. This time our approach was to use structured analysis to specify the system given the architecture instead of specifying the system independent of the architecture. We felt this was necessary to meet our performance goals and to help clarify the interfaces. Where before we based the specification on the type of I/O, this time we based the specification on fast (able to complete within the driver), deferred (needs operating system help to complete), or inbound paths. Using the data interviewing technique, we started with the context diagram (Fig. 6) and the level 1 and 2 diagrams (Figs 7 and 8). The level 1 diagram has three processes: DIOM, FIOM, and IIOM, which make up the redesigned driver. The level 2 diagrams (of which Fig. 8 is an example) have a process for each layer of the architecture and some general utility processes. After these diagrams were done, we broke the work up by process. We were able to use many of the diagrams from the earlier structured analysis work.

When reviewing data flow diagrams and process specifications, issues, questions, and problems were easy to detect. They tended to stand out on the diagrams. It was easy to see if data was missing or wrong or hadn't been initialized. For example, in the finish_read process specification (Fig. 4),

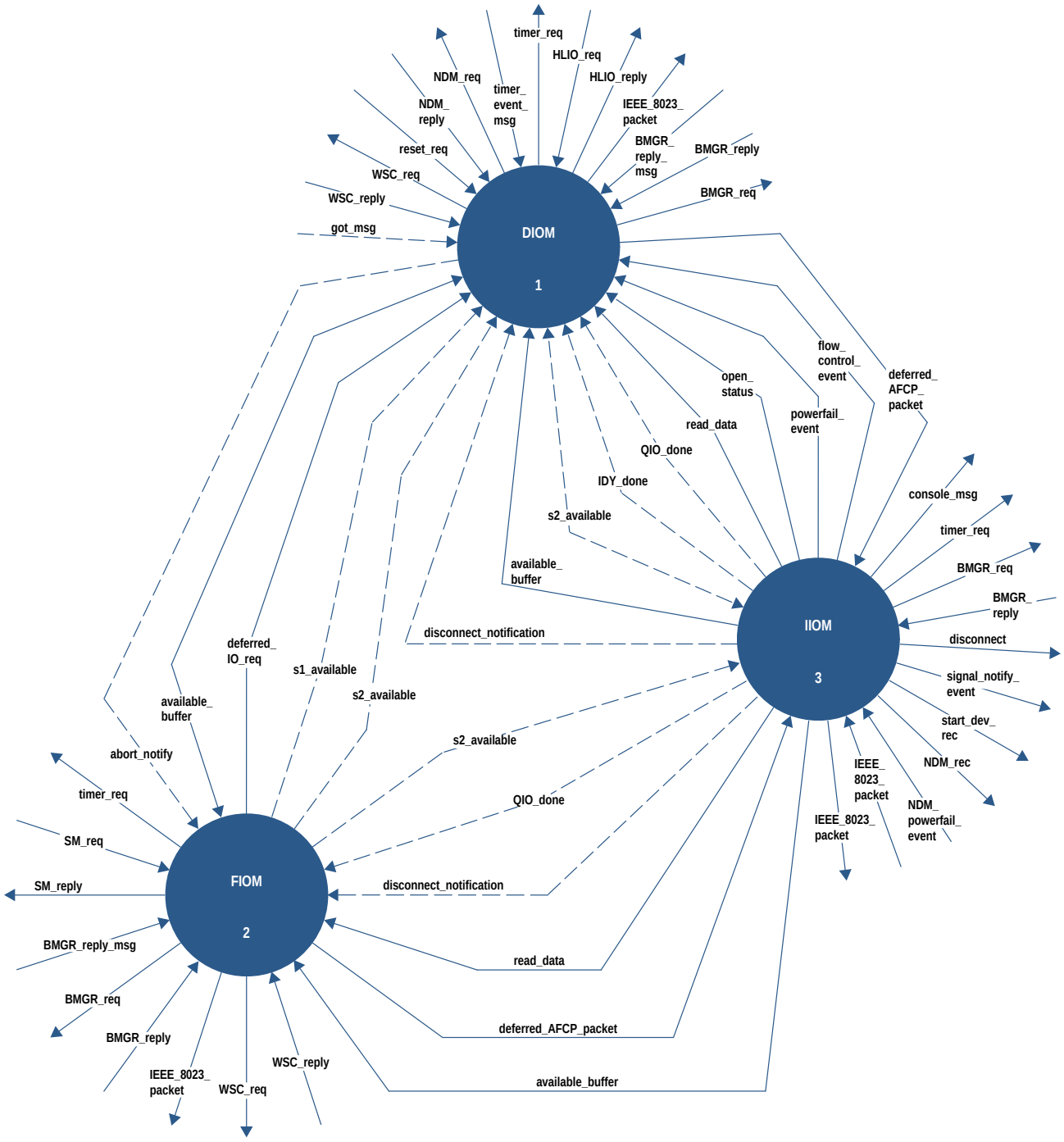


Fig. 7. Driver level 1 diagram showing the three major modules: the fast I/O manager FIOM, the deferred I/O manager DIOM, and the inbound I/O manager IIOM.

unused_buffer was a data flow out of the process but the buffer wasn't freed in the first draft of the specification. We kept a list of issues and questions and their resolutions during the structured analysis process.

A month before the external specification was due, we stopped structured analysis work and concentrated on writing the external specification document. Much of it was taken from existing documents since our upper and lower interfaces didn't change. The top-level structured analysis diagrams (see Fig. 6) were used to determine interfaces and to help define the TIO support and operating system changes that the driver required.

Originally, we did not have an internal specification in our plans. However, the internal design was appropriate for the structured design but not the analysis, and we needed a document for the structured analysis work. Therefore, we split the design period into two periods and added an internal specification document. The internal specification contained

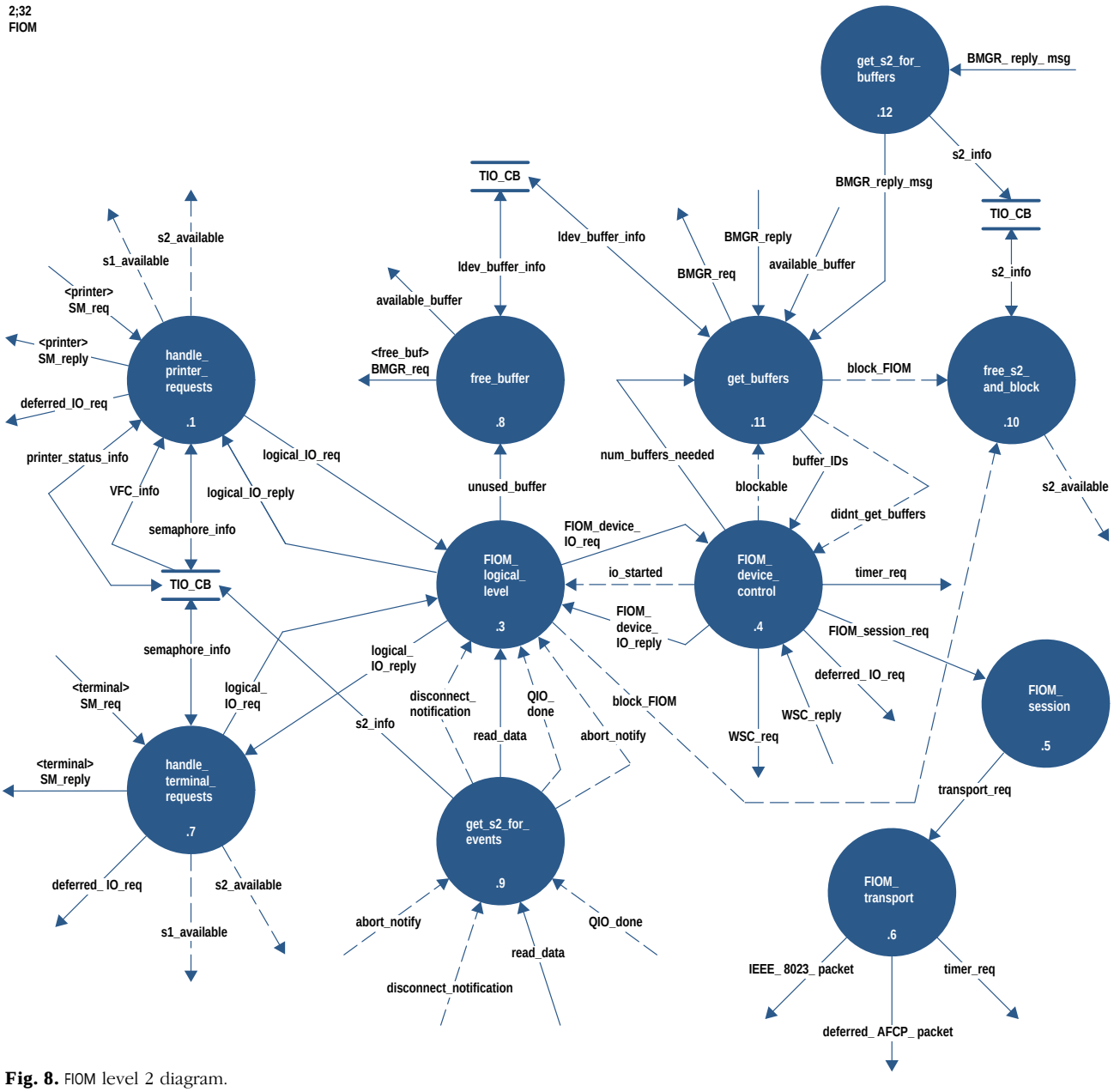


Fig. 8. FIOM level 2 diagram.

a brief introduction about the context diagram and descriptions of the interfaces. The rest of the document was generated using Teamwork, a software tool for structured analysis and structured design from Cadre Technologies, Inc.

Structured Analysis Recommendations

Data flow diagrams generally "feel right or wrong." Tom DeMarco[1] encourages engineers to throw away diagrams several times. The use of structured analysis to specify a system naturally raises the questions that need to be answered about the product.

If we had to do it all over again, we would have resolved the architecture-versus-structured-analysis problem earlier, and not tried to do structured analysis without considering the architecture of the system. We did not use the approach outlined by Hatley and Pirbhai,[5] but we did use something related to it. There is a fine line between considering the architecture and including design in the specification. Because this was a redesign and performance was important, we had already analyzed the original driver to find opportunities for shortening the path lengths. These opportunities needed to be incorporated into the design. We didn't know how to do that at the design phase if they weren't included in the specification as well, so we put the architecture in at the top levels of the specification (fast I/O manager, deferred I/O manager, inbound I/O manager).

We should have spent more time keeping the data dictionary entries up to date. All through the project, the lack of attention paid to data definitions was a major failing. The data needs to be defined as the diagrams and process specifications are created. Data dictionary entries that were related to data structures in the original driver were easy. However, we did not always type in the complete definitions. We did not document new data dictionary entries rigorously and this

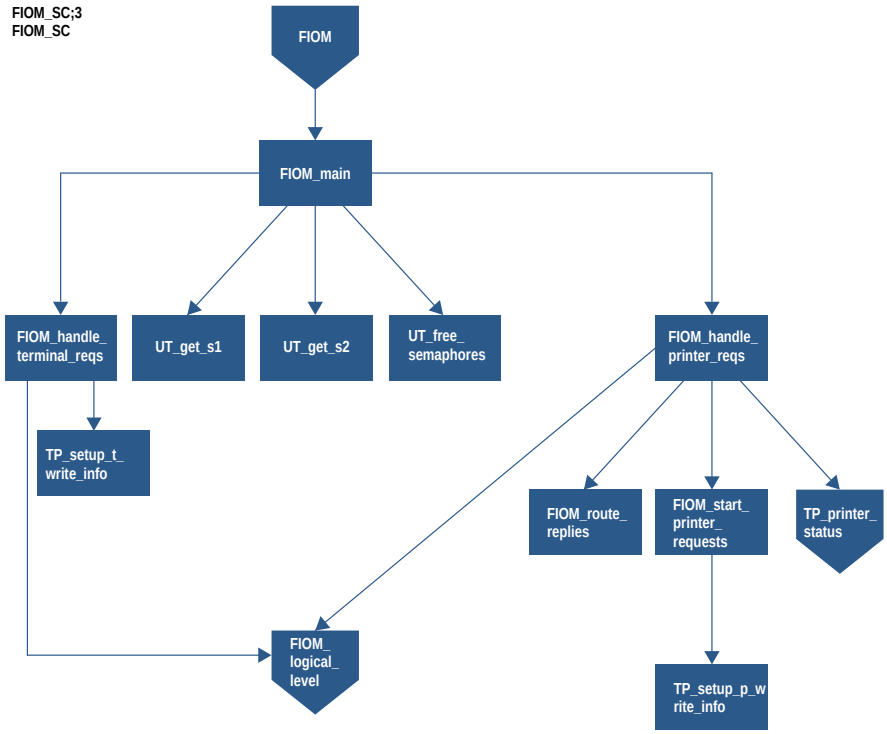


Fig. 9. FIOM structure chart.

weakened the entire data dictionary. We would have also planned time for the internal specification, its inspections, and the rework in the schedule.

Structured Design Overview

Design is the process of transforming the specification of what must be done into the plan of how it will be done. Classical design produces a narrative document with some graphics. It starts with the procedural characteristics. Information is often repeated throughout the design document, and the document is generally not specific enough. This results in some design improvisation during implementation.

Structured design introduces structure and graphics into the design process to cope with the largeness of the system. The goal of structured design is a highly maintainable, comprehensible, and easily tested top-down design. The system is partitioned into components which interact to achieve the functionality of the system.

To create the structured design, the data flow diagram processes are grouped into processes that deal with inputs, deal with outputs, transform inputs into outputs, or handle transactions between inputs and outputs. Modules are then created from these groups. Evaluation and refinement techniques—called cohesion, coupling, and packaging— complete the building of the documents. The structure chart, design dictionary, and module specifications are the documents produced through this process. A structure chart shows the basic components (modules) and their interfaces in a top-down graphical manner. The design dictionary defines the interfaces. It uses similar language to the data dictionary of structured analysis. The module specifications define the procedural part of the design and the sequence of interactions. Each module specification describes what part of the specification is being satisfied, what the module needs to communicate, and how it performs the function. A module specification is written in structured English, as a decision tree or table, or as a state transition diagram.

Fig. 9 shows the first structure chart for the fast I/O manager. There are a main module, some utility modules (e.g., UT_get_s1), and device-specific modules (e.g., FIOM_handle_terminal_reqs) which eventually led to the FIOM_logical_level module. Fig. 10 is the module specification for the fast I/O manager module, FIOM. It shows the sequence in which the other modules are called. Design dictionary entries look similar to data dictionary entries.

The Project and Structured Design

Once we had finished the internal specification, it was unclear how to derive the structured design from it. We decided on the following approach. Each structured analysis process was turned into a module. A hierarchical structure was developed for each level by "promoting a boss" or "hiring a boss" module when necessary. Comparing the fast I/O manager data flow diagram (Fig. 8) with the fast I/O manager structure chart (Fig. 9), one can see that the FIOM_main module is "hired as a boss" (created) to handle semaphores and to call the FIOM_handle_terminal_reqs and FIOM_handle_printer_reqs modules.†

When the rough drafts of the structure charts were ready, they were entered into the Teamwork tool. The team then determined what each module does and knows. The next step was to walk through the design and be sure it works. This involved tracing each I/O through the design to be sure

† Note: Some of the process names are capitalized or spelled differently in Figs. 8, 9, 10, and 11 and in the text of this paper. Some names did change between the structured analysis document and the structured design document. In the module specification, Fig. 10, the engineer chose to capitalize the function codes, request types, and procedures called. In Fig. 11, the names are capitalized because the coding convention that was followed capitalized procedure calls. The MPE/iX operating system is case insensitive, so the differences are insignificant.

```
NAME:
FIOM_main;2

TITLE:
FIOM_main

PARAMETERS:
SM_CB
SM_arg_list
return_status

LOCALS:
request
status

GLOBALS:

BODY:
*************************************************************************

Calling parms:

 SM_CB          --  pointer to SM control block
 SM_arg_list    --  pointer to storage manager arguments

*************************************************************************

TIO_CB := SM_CB^.fwrt_cb;   * used to be fast write control block *

case SM_arg_list^.sm_generic_parms.func_code of

 SM_READ_FN:
  request := READ_REQ;

 SM_WRITE_FN:
  request := WRITE_REQ;

 SM_CONTROL_FN:
  if (SM_arg_list^.sm_control_parms_t.* terminal control * <> TC_QUIESCE_IO)
   request := TC_REQ;
  else
   request := QIO_REQ;

 SM_DEVCONTROL_FN:
  request := TC_REQ;

end;

* since this is the FIOM, and FIOM only handles waited I/O, always block       *
* if necessary to get semaphores.                                              *

status := UT_GET_S1 (FIOM, TRUE);
if (status <> TRUE)
 * error, log and exit *;

status := UT_GET_S2 (FIOM, TRUE);
if (status <> TRUE)
 * error, log and exit *;

if (TIO_CB^.device_type = * terminal *)
 status := FIOM_HANDLE_TERMINAL_REQS (request, SM_arg_list);
else
 status := FIOM_HANDLE_PRINTER_REQS (request, SM_arg_list);

return_status := MAP_TO_OS_STATUS (status);

UT_FREE_SEMAPHORES;
```

Fig. 10. FIOM module specification.

that the right information was available to the modules, and that the modules were doing the right things. During the entire process, modules were collapsed when it was reasonable. We did not do much evaluation of the interfaces (data coupling or cohesion) because of a lack of time. The internal design document contained only Teamwork structure charts and module specifications.

After the internal design was inspected and the rework completed, we worked on defining the procedure declarations, data structures, software configuration (file structure), and coding standards. The module specifications were the basis for the Pascal procedures. Coding was mainly a matter of converting pseudocode to Pascal. Fig. 11 is the outer block of the fast I/O manager module. Comparing this to the fast I/O manager module specification, Fig. 10, one can see how closely related they are. The code adds more detail to the module specification framework. (Fig. 10 also illustrates a problem we had with module specifications: many of them

```
begin  { tio_fiom }

try
begin { Try section }

 tio_cb  :=  tio_cb_ptr_type( fiom_cb_ptr );

 I_index := 0;


 {-------------------------------------------------------------}
 { The FIOM only handles blocked IO, always block              }
 { if necessary to obtain semaphores                           }
 {-------------------------------------------------------------}
 if ( CUT_GET_S1 (tio_cb, Fiom) ) then
    if ( CUT_GET_S2 (tio_cb, Fiom) ) then
       if ( not (tio_cb^.device_type.printer) ) then
          { terminal }
          status := FIOM_HANDLE_TERMINAL_REQUESTS
                        ( sm_arg_list_ptr_t(arg_list))
       else
          { printer }
          status := FIOM_HANDLE_PRINTER_REQUESTS
                        ( psm_parm_ptr_t(arg_list))
    else
      begin
       {**** LOG S2 error ****}
       status := Bad_status;
       I_tio_status.int_status.status_code := Internal_err;
       I_tio_status.int_status.layer := Main_layer;
       I_tio_status.int_status.proc_number := Pn_tio_fiom;
       I_tio_status.int_status.location := 0;
       I_tio_status.int_status.Iio_flag := False;
       I_tio_status.ext_status_hpe := status;

        CUT_LOGMSG  ( tio_cb,
                      I_tio_status,
                      Main_layer,
                      Fiom
                      );
      end
 else
  begin
   status := Bad_status;
   {**** LOG S1 error ****}
   I_tio_status.int_status.status_code := Internal_err;
   I_tio_status.int_status.layer := Main_layer;
   I_tio_status.int_status.proc_number := Pn_tio_fiom;
   I_tio_status.int_status.location := 1;
   I_tio_status.int_status.Iio_flag := False;
   I_tio_status.ext_status_hpe := status;

   CUT_LOGMSG ( tio_cb,
                I_tio_status,
                Main_layer,
                Fiom
                );
  end;

 CUT_FREE_S2 (tio_cb);

 CUT_FREE_S1 (tio_cb);

end;  { Try section }

recover
begin { Recover section }


 case ESCAPECODE of
  0 : ;
 otherwise
  begin
   I_tio_status.int_status.status_code := External_err;
   I_tio_status.int_status.layer := Main_layer;
   I_tio_status.int_status.proc_number := Pn_tio_fiom;
   I_tio_status.int_status.location := 2;
   I_tio_status.int_status.Iio_flag := False;
   I_tio_status.ext_status_hpe := hpe_status(ESCAPECODE);

   CUT_LOGMSG ( tio_cb,
                I_tio_status,
                Main_layer,
                Fiom
                );
  end;
 end;
end;  { Recover section }
end;  { tio_fiom }
```

Fig. 11. FIOM code.

were too code-like. The case statement is an unnecessary level of detail, and is actually not carried through into the code since the arg_list is passed to the HANDLE procedures). Analysis of the code shows a 33% reduction in code size compared to the original driver. The reduction comes from the reuse of procedures as a result of structured design, and from a structure that allows common routines to be shared between the DIOM, FIOM, and IIOM modules instead of requiring one copy per module. In the original driver, there was a logical device manager for terminals and one for serial printers. In the redesigned driver, the fast I/O manager and deferred I/O manager are able to handle both terminals and serial printers.

Structured Design Recommendations

In general, not enough attention was paid to defining design dictionary entries or data structures. Structured design added the next level of detail to the design from the specification. It was easy to develop the design from the specification once we had our approach. We occasionally created module specifications that are too much like code. Module specifications are not meant to handle the small details that are best suited to coding, but to define how the function is performed. If too much attention is paid to code-like details less is paid to how the specification is to be implemented.

If we had to do it all over again, we would have paid more attention to the interfaces and better defined the data passing between modules. Since we were using a lot of existing interfaces, we did not put the time into the data or design dictionaries. This meant that the data elements were poorly defined, since we tended to assume that everyone knew how the data was defined. We would also have developed a module specification standard to create consistency and avoid variations in the degree of code-like English in the internal design.

Inspections

The internal specification required some basic structured analysis training for the inspections. This was because of the graphical nature of the diagrams and the decision structures. The internal design (structured design document) was easier to understand since it consisted of structure charts, pseudocode, and data descriptions. Inspections were done by a depth walk through the processes and hence concentrated on architectural levels as opposed to interfaces between levels. The inspections did an excellent job of checking for functionality and design flaws, but were weak in the area of interface checking.

The original driver internal design documents were narrative with some state transition diagrams. They were much shorter than the internal specification and internal design created with structured methods. Past reviews of project documentation (external specification, internal specification, internal design) took place at one 2-to-4-hour meeting per document. Since the structured documents were larger and contained more detail, it took about 300 engineer-hours to inspect or review them. The participants felt that the reviews of the internal specification and internal design (structured analysis and structured design documents) were very valuable. The material was much easier to inspect for completeness, correctness, and functionality than the narrative internal specification and internal design documents.

Schedule

There is a concern that using structured analysis and structured design has a large impact on the schedule. The negative impact of structured analysis and structured design on our schedule consisted of training time (two weeks), time spent becoming familiar with structured analysis (one month of intermittent structured analysis), and longer inspections. A positive result of using structured analysis and structured design is that each step is a progression from the last. Each document is built on the previous document, unlike the classical design in which each document is not so closely tied to earlier ones. The effort required at each phase was less than the one before, and was mainly directed at adding more details. The positive impact of structured analysis and structured design on the schedule was shorter test cycles. Nonregression tests were passed ahead of schedule.

Results

The redesign project improved block mode performance and achieved a greater than 30% reduction in the terminal driver path length. The code passed all functional tests including 24-hour and 72-hour nonregression tests. 75% of the testing errors were coding errors. The majority of the remaining defects were found in areas that were known to be weak from the design and inspection phases. The engineers valued learning structured methods and tools both for the quality and completeness of the design and the acquired skill set. The project team believes that the design of the new HP 3000 MPE/iX terminal and serial printer driver would not have been accomplished in the same amount of time with the same amount of thoroughness by the traditional design techniques.

Acknowledgments

I would like to thank the project engineers—Jeff Bandle, Jon Buck, Larry White, and Erik Ness—for their hard work and team spirit, with special thanks to Jeff for his help with the figures for this paper. It was a pleasure to work with them as project lead engineer. Without them, this paper on our success with structured analysis and structured design would not have been possible. I would also like to thank the project manager, Celeste Ross, for her willingness to try something new and for her support, and the inspectors, reviewers, and moderators for their much appreciated time and effort.

References

1. T. DeMarco, *Structured Analysis and System Specification,* Prentice-Hall, Inc., 1979.

2. M. Page-Jones, *The Practical Guide to Structured Systems Design,* Yourdon Inc., 1980.

3. *Grenoble Networks Division Product Life Cycle Guidelines, Edition 2,* Hewlett-Packard, April 1991.

4. R. Merckling, *Structured Analysis/Real-Time and Structured Design/Real-Time Class Notes,* 1991.

5. D. Hatley and I.A. Pirbhai, *Strategies for Real-Time System Specification,* Dorset House, 1988.