REALTIME TELEMETRY PROCESSING SYSTEM (RTPS) III
A PREVIEW OF SOFTWARE DEVELOPMENT IN THE 1990s

Jerry L. Hill
CSC Network Systems Division
301 North R Street
Lompoc, CA

ABSTRACT

Software development is becoming less an art form and more an engineering discipline. Methods of software development which leave as little as possible to chance are constantly being sought and documented. However, the gap between what is written and what is actually applied is usually quite wide. The only way this gap can be narrowed is through practical application of these very detailed and complex methods. Since it is unlikely that the complexity of these methods will be reduced, automation must be employed wherever possible in the software development process. This paper addresses the successful development of software for the Navy's Realtime Telemetry Processing System III (RTPS III) using practical application of existing methodology in conjunction with a Computer Aided Software Engineering (CASE) tool. Based on this experience, the conclusion presents implications affecting software development the 1990s.

Key words:

CASE
Reuseable Code
RTPS III
Top Down Design
Software Documentation
Unit Folders
Unit Testing
DoD Standards
DoD-STD-2167
Software Development Standards

INTRODUCTION

On February 5, 1987 Computer Sciences Corporation (CSC) was awarded a contract to provide a telemetry processing system for the Naval Air Test Center, Patuxent River, Maryland. The good news was: We had won the contract! The bad news was: We had won the contract! It was a fixed price contract with a very short fuse. In one year we were to integrate two systems which, between them, contained over 75,000 lines of reuseable code. In addition, we were to write another estimated 75,000 lines of new code. Further, the contract had been costed for a one year duration, and past experience had shown that original estimates are usually somewhat ambitious. One function, for example, was originally costed at 250 lines of code. It became apparent within the first two weeks of contract award that this particular function would probably exceed 1000 lines of executable code. Projecting this particular error across the total original estimate provided a very unsettling feeling for the beginning of a new project. As if that weren't enough, long-lead hardware items had yet to be ordered, giving even the most optimistic of planners reason for scepticism. How would we write the interfaces to hardware that wasn't scheduled to be delivered for six months? And that's if it were on time and worked as specified.

It was also difficult to believe that DoD Standard documentation and attendant review cycles would leave us with much time to do the "real work." Finally, design on the front-end of the development cycle and integration and testing on the back-end of the cycle would probably take around half of the total development time. This left no more than six months to integrate two existing systems of significant complexity and prepare what was beginning to look like an optimistic estimate of 150,000 lines of code for delivery to the Navy (the final figure was something over 170,000 lines of executable code). A recognizeable pattern was emerging; one common to most problem software development efforts. Even if everything went well, the budget was in serious jeopardy if techniques weren't found to aggressively streamline the development process. Since the proposal effort we had been investigating Computer Aided Software Engineering (CASE) as one way of improving productivity, but to initiate this approach on an already time-critical project contained risk. We had some hard decisions to make and little leeway for error.

THE GOAL

Our goal, and that of our customer, was to produce a system that would upgrade the Navy's existing capability to a state-of-the-art telemetry processing system. This meant implementation of a loosely coupled system originally consisting of two "streams," but expandable to as many as eight. These streams would be connected to a file system that would provide the files describing telemetry formats, measurement attributes, and

processing rules necessary to support Navy flight test operations. Several of a stream's salient features were to be:

- Process Four 10 Megabit PCM links.
- Process up to 64 FM channels at an aggregate sampling rate of 200 Ksps.
- Perform 200 Ksps EU conversion.
- Record EU samples at 160 Ksps.
- Feature five distinctly different types of display devices to display graphic plots, color graphics, out-of-limit conditions, critical measurements and strip chart plots.
- Perform recall of telemetry history for plotting in real-time.
- Provide real-time calibration of sensor data.

Because of a prototype system built by the Navy, they were able to write a very definitive specification. The basic architecture of a stream was dictated by the specification. This architecture consisted of a telemetry front end, telemetry preprocessor, display processor and a dedicated application-specific processor all linked together through shared memory. All streams were to be interfaced to a single file system consisting of two processors with a common shared memory interface (See Figure 1). The Navy's prototype system consisted of this very architecture, but with only a small subset of the required capabilities. There was considerable overlap between the Navy prototype functions and those in existing CSC built systems, especially in the areas of telemetry preprocessing and inter-processor interface functions. The Navy had specified that Adage vector refresh hardware be used for graphic plots. This particular software was unique to the Navy prototype system and was a good candidate for integration into RTPS III.

We had to either produce approximately 150,000 lines of executable code from scratch in less than a year or we had to find a way of effectively combining reuseable code from two distinctly different software architectures into a homogeneous system. Providing the two systems would be successfully integrated, there still remained a significantly large coding task to satisfy the remaining requirements.

THE PLAN

Thanks to the Navy's unusually good specification, the problem was well defined. But it was not an easy problem and the time frame was very short. To subject this project to the normal development cycle with the same procedures that had been used in the past was to invite a cost overrun of significant proportions. At the same time, to take short cuts in design and documentation could place the project at substantial risk. We had to find a way of shortening the development cycle without bypassing any procedures that could cause the final outcome to be in doubt.

Staffing

At first glance staffing would appear to be one place to shorten the schedule. If 10 programmers would take two years to write 200,000 lines of code then 40 programmers should logically take six months to accomplish the same task.

To seriously consider this option would be to come face-to-face with the "Mythical Man Month" addressed by Fred Brooks in his book of the same name. Resources simply were not available to support such a staff, to say nothing of the management task. Staffing had to be accomplished top-down, with team leads being brought on first to do the design, followed by programmers to implement that design. Within this context the team leads were considered the most critical single element in the development process. Once the team leads were in place, the design task could begin. Further staffing from then on could be based on the team leads' need for programmers to implement their design. The philosophy was to do as much work up front as possible with as few people as possible. This would preserve the budget until coding tasks had been defined and budget could be most efficiently allocated. Since the code and debug phase of the project was only six to eight months, rapid staffing and down-staffing were necessary. The only reasonable approach in accomplishing this was liberal application of consultants. This we did by staffing with half consultants and half CSC employees.

Tools

We had an adequate set of programming tools for use in software development and they had been used effectively in past projects. What we didn't have were any tools to assist us in the design process. A time consuming problem on past projects was that of meaningfully communicating design to those who needed it, when they needed it. Typically, design starts with the requirements, matures inside one or more system architects' heads, and spills onto engineering pads, notebooks and backs of envelopes. It is then given to Data Entry and Drafting to produce a document. Two weeks after the design is conceptualized it is available for review or for programmers to implement. In the meantime, the design changes to some extent and that update is sent through the same procedure. This is all very time consuming and confusing. Further, costly errors are introduced into the system by each layer of communication. And whatever the errors, there is no real methodology, save for design reviews, for finding and correcting them until they appear as bugs in code. Finally, the emphasis on the resultant design documentation is often influenced more by the DoD-STD format than the necessity to communicate design. Often the end result is that the design and documentation become separate processes and errors are not detected until late in the development cycle.

We decided to use a Computer Aided Software Engineering (CASE) tool to help solve this problem. There was risk involved because CASE represented new technology and there would be a learning curve. But the possible benefits were considered worth the risk. A designer would be able to create and maintain his own design diagrams and process narratives and they could be easily proliferated by giving CASE file access to those who needed the design. The result of using CASE is that the latest design documentation is always in the hands of those who need it. With DesignAid by NASTEC, the CASE tool we selected, information is entered into a design dictionary which is used to "validate and balance" the design flows automatically. Potential design errors (especially interface errors) are brought to the designer's attention so they can be corrected in the design phase. When we showed the Navy the way we were using the CASE tool and how it satisfied the intent of DoD-STD documentation, they encouraged and supported our approach. This represented a radical departure from past DoD contracts with which I had been involved. The resulting documentation was both more practical to use and much less costly to produce and maintain. The top three levels of Yourdan/Demarco standard data flow diagrams printed from DesignAid files are shown in Figures 2, 3 and 4.

Unit Folders

Unit folders were to be the result of the top down design process. The team leads were to decompose each subsystem down to codeable modules or units, each represented by a unit folder providing:

- Name of source module, programmer and charge code.
- Specification requirements to be satisfied by that unit.
- Design path (hardcopy from CASE tool) showing relationship to the system, including semi-detailed interface description.
- Unit test section.
- Design notes.
- Coding standards.

The Unit Folder provides the medium for the designer to communicate definitive requirements and design to a programmer. It, in turn, provides the medium for a programmer to communicate his solution to the designer. These unit folders were to be kept simple and informal. They were to be produced using clerical assistance to lessen the burden on the technical staff. In practice the support staff built the original folder for the team leader who, in turn, presented it to the programmer as an assignment. It was hoped that through use of the unit folder, programmers would be able to become productive much more quickly, and that they would be able to adapt more easily to a predetermined structure.

Approach

We really weren't planning to do anything that hadn't been tried before or at least anything to which extensive "lip service" hadn't been given. Top down design wasn't a new concept, though application of CASE technology to this concept was relatively untried. Unit folders were not a new idea, but they had to be made more practical by being made the repository for useful information. Administrative drudgery had to be off-loaded to the support staff so that the technical staff could concentrate on solving the technical problems. Documentation had to reflect the actual design as it occurred and it had, ultimately, to reflect the end product. This meant probable deviation from DoD standards which couldn't be done without the confidence and cooperation of the Navy.

Peer reviews would be conducted informally by the team leads. This wasn't the original intent for these reviews, but it was decided that there wouldn't be enough time to have programmers review each other's code. Units were generally small enough (60-80 man-hours) that it would literally be cheaper to rewrite a module if necessary than to expend significnt time performing peer review on all modules across the board.

Within the constraints inherent in this plan, team leads were given the authority to progress with the development of their respective subsystem as they deemed necessary. Administrative tasks were off-loaded from them to the software manager and support staff as much as possible so that team leads could concentrate on technical issues.

DEVELOPMENT

Development on RTPS III consisted of allocating unit folders to the programming staff so that programming could begin. In conjunction with this, logical milestones were defined to mark the progress of the development. On RTPS III the milestones consisted of builds which represented stages in the system development that must be reached before the next stage could reasonably be accomplished. The first build consisted of the integration of two existing systems: The Navy's "Adage-peculiar" display software and Computer Sciences' telemetry acquisition and processing software. While three staff members were performing this task, others were coding Build 2 software. Build 2 contained most of the functions that were considered necessary to support an operational system. Build 3, the last build, contained functions which were less critical to system operation, or functions which were dependent upon completion of Build 2 functions.

Unit folders were used to provide the framework for software development. These folders were fairly informal and provided a repository for everything a developer or maintenance programmer could ever want to know about the program(s) represented by that folder. There were 107 unit folders divided among 20 programmers (including the team leads).

Team leads controlled the unit folders. Related unit folders were assigned to a programmer to be completed on a sequential basis. The scheduling of a module was dependent upon when the hardware or software it required were to be available and/or when that module would be required by another software element.

Predictably this didn't work as well as we would have liked due to unforseeable problems. Like it or not, we were in a dynamically changing environment. It made following any plan very difficult. To help alleviate this situation, we created a dynamically changeable status board containing a time-relative representation of all modules to be coded. These modules were rescheduled (moved around on the board) on a weekly basis subject to unforseen problems. This board provided management visibility into areas which were the most critical so that action could be taken immediately when something didn't happen as planned (such as delivery of a hardware component). This area is a good candidate for automation in the 1990s.

When fully staffed, we found that we had a full range of programming ability represented. The fastest coders were an order of magnitude faster than the slowest coders. As the faster programmers finished their "line" of modules, they were assigned modules from other programmers' area of responsibility. We reassigned slower programmers to other areas where they were still able to make a valuable contribution to the project.

INTEGRATION/TEST

There was a separate group assigned to perform integration and test of the RTPS III system. This group was responsible for the writing of the test plans and procedures for each build as well as providing the focal point for system integration. Test procedures were written to the specification paragraph level as opposed to addressing each specific requirement. The philosophy was that the programmer informally perform and document the detailed "shall-level" testing at the unit level and that the system testing be formally documented and performed by the Test and Integration group. This was fine as long as the programmer held up his end of the bargain. As might be expected, not all programmers were equally motivated to perform (or at least document) unit testing for their assigned unit folder. The test group, in cases where the unit testing was deficient, performed and documented the unit test themselves, with some "hand holding" by the responsible programmer.

When a defineable capability (consisting of one or more related modules) was complete, it was held by the software development staff until the Test and Integration staff was ready for it, then it was officially turned over. These mini-builds or "bundles" could then be integrated into the existing system regardless of whether or not it was time for another build. This provided an even flow with no "big bang" build test at the end of a build. The

build concept was still used, but more for purposes of preparing test procedures and configuration management than for control of system development.

Approximately 10% of the team leads' time was originally allocated to assist in system integration. The team leads, who in effect, became members of the integration team spent at least the originally allocated percentage of their total time in this endeavor.

SUMMARY AND CONCLUSION

Computer Sciences Corporation (CSC) was awarded the RTPS III contract the first week in February, 1987. The system was shipped the second week of March, 1988, 13 months after contract award. Software delivery consisted of over 170,000 lines of executable code, over half of which was new code. Including design, code and debug, test and integration and design documentation, the total cost for this software development was less than $10 per line of code. This unheard of (by this author) accomplishment was due to:

- A technically astute customer who knew what he wanted and was willing to work in cooperation with the contractor to get it.
- Use of CASE technology in design and documentation of the software.
- Extremely talented team leaders in each of the three major subsystems.
- Dedicated software staff doing whatever it took to meet reasonable schedules.
- Timely inclusion of reuseable code.

In the decade of the 90s, as in the past, successful software projects will be predicated on hard work and intelligent application of accepted principles of system development. This author predicts that CASE will dominate software development in the next decade. This will go hand-in-hand with a modification of DoD Standards to accomodate the automated design process. If current standards don't change fast enough to keep up with new methods, DoD customers will look favorably upon contractors presenting them with reasonable alternatives to DoD standard documentation.

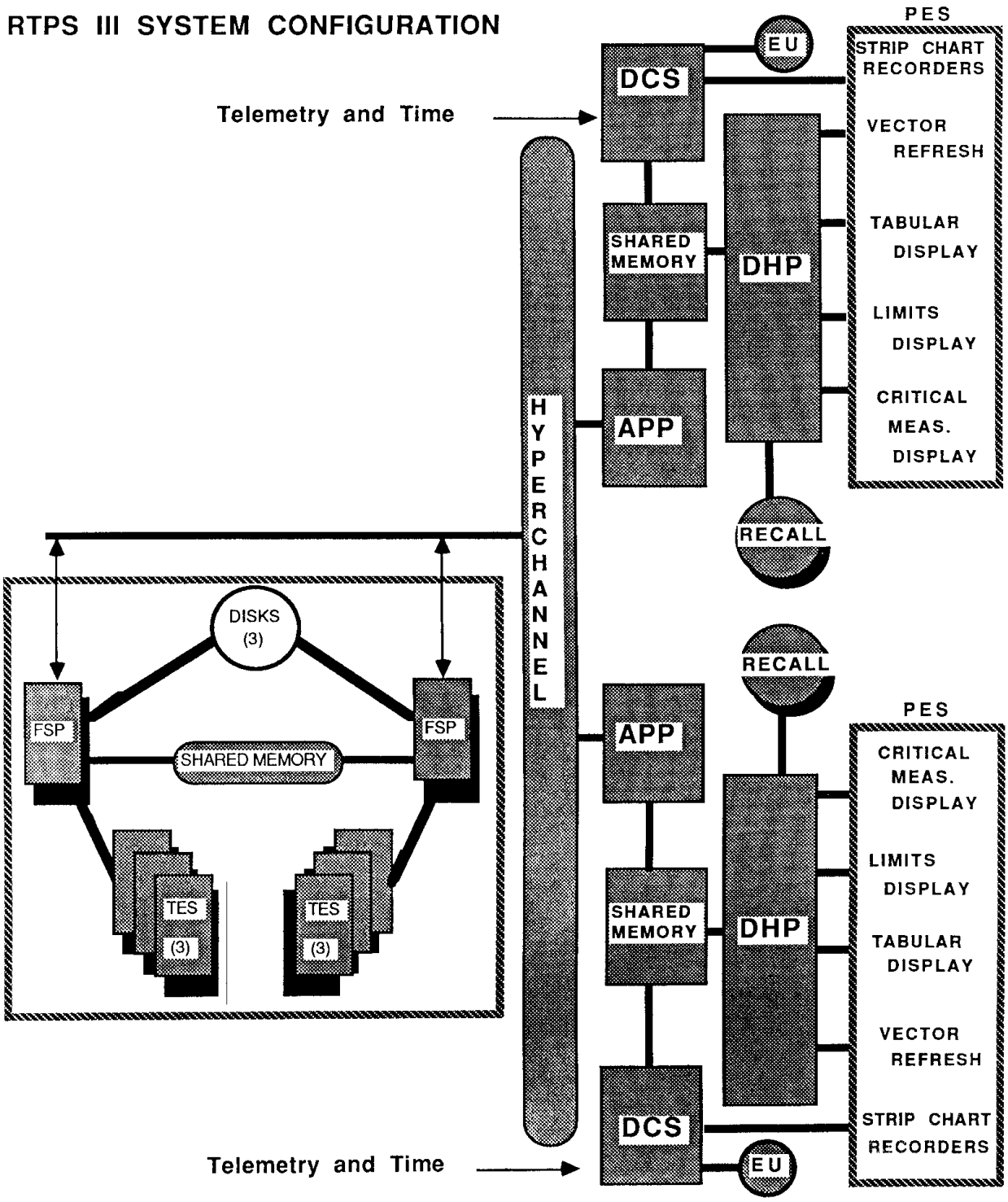


FIGURE 1. RTPS III SYSTEM CONFIGURATION

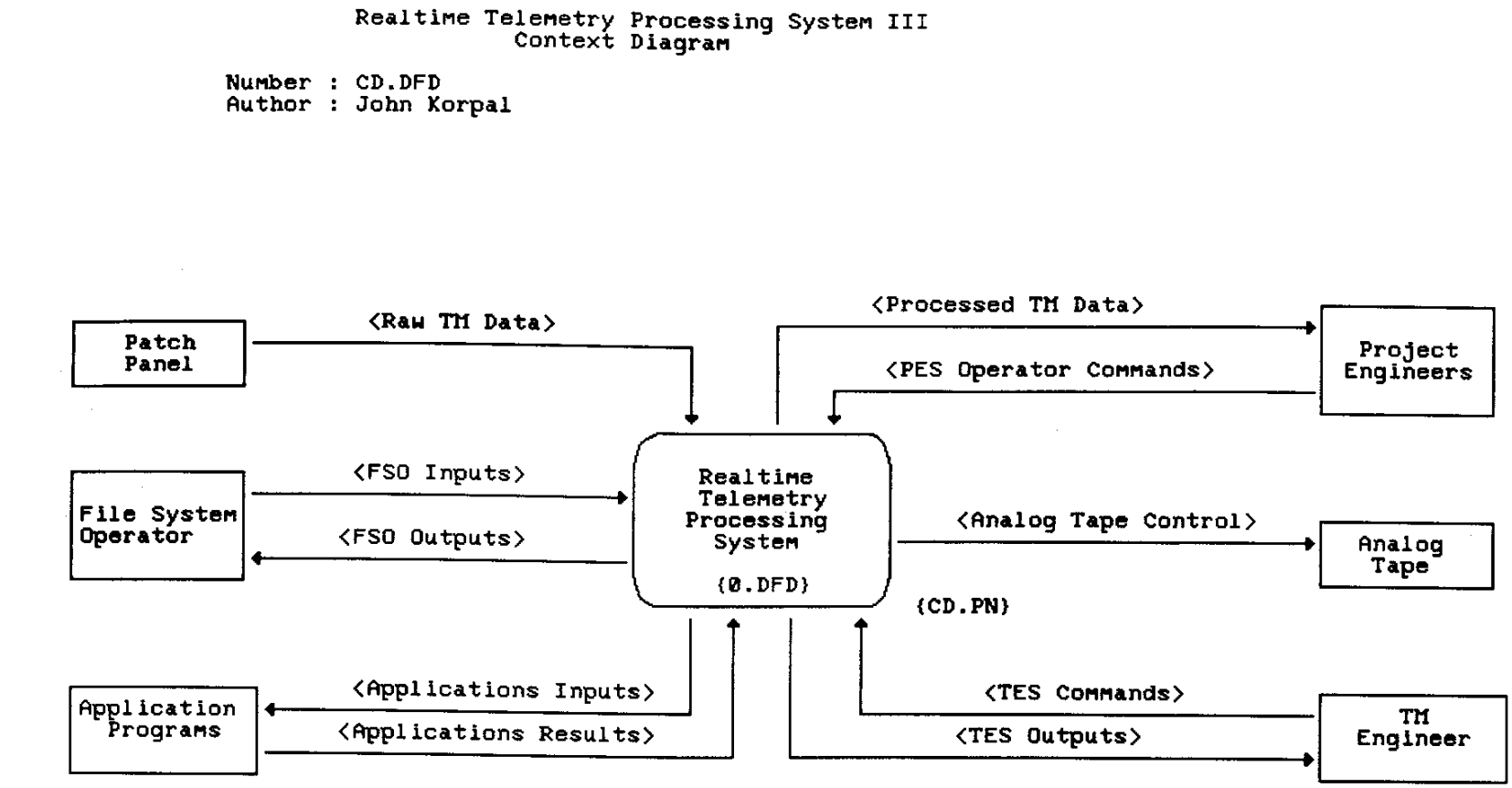
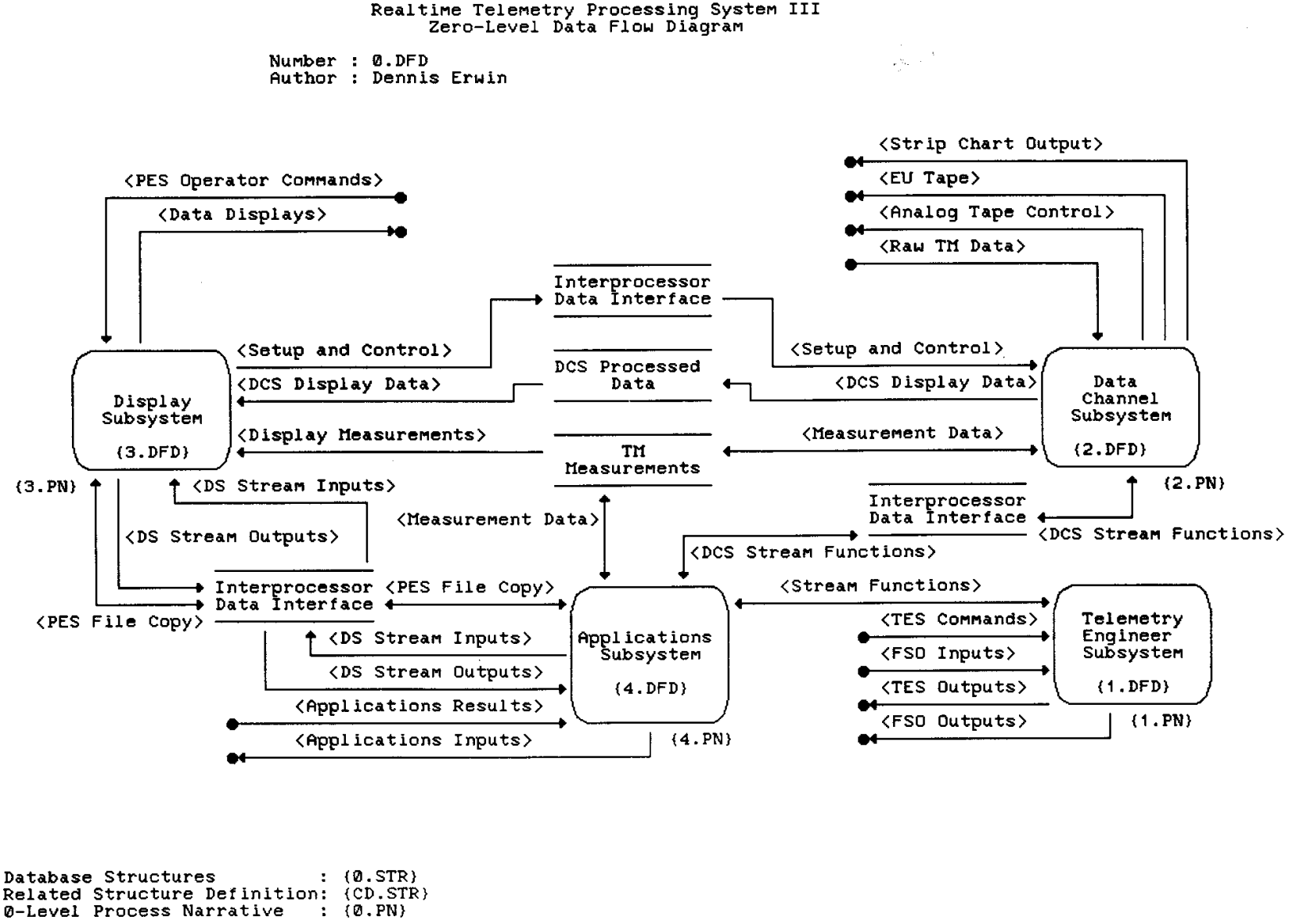


FIGURE 2. RTPS III CONTEXT DIAGRAM



Database Structures : {0.STR}
Related Structure Definition: {CD.STR}
0-Level Process Narrative : {0.PN}

FIGURE 3. RTPS III ZERO-LEVEL DATA FLOW DIAGRAM

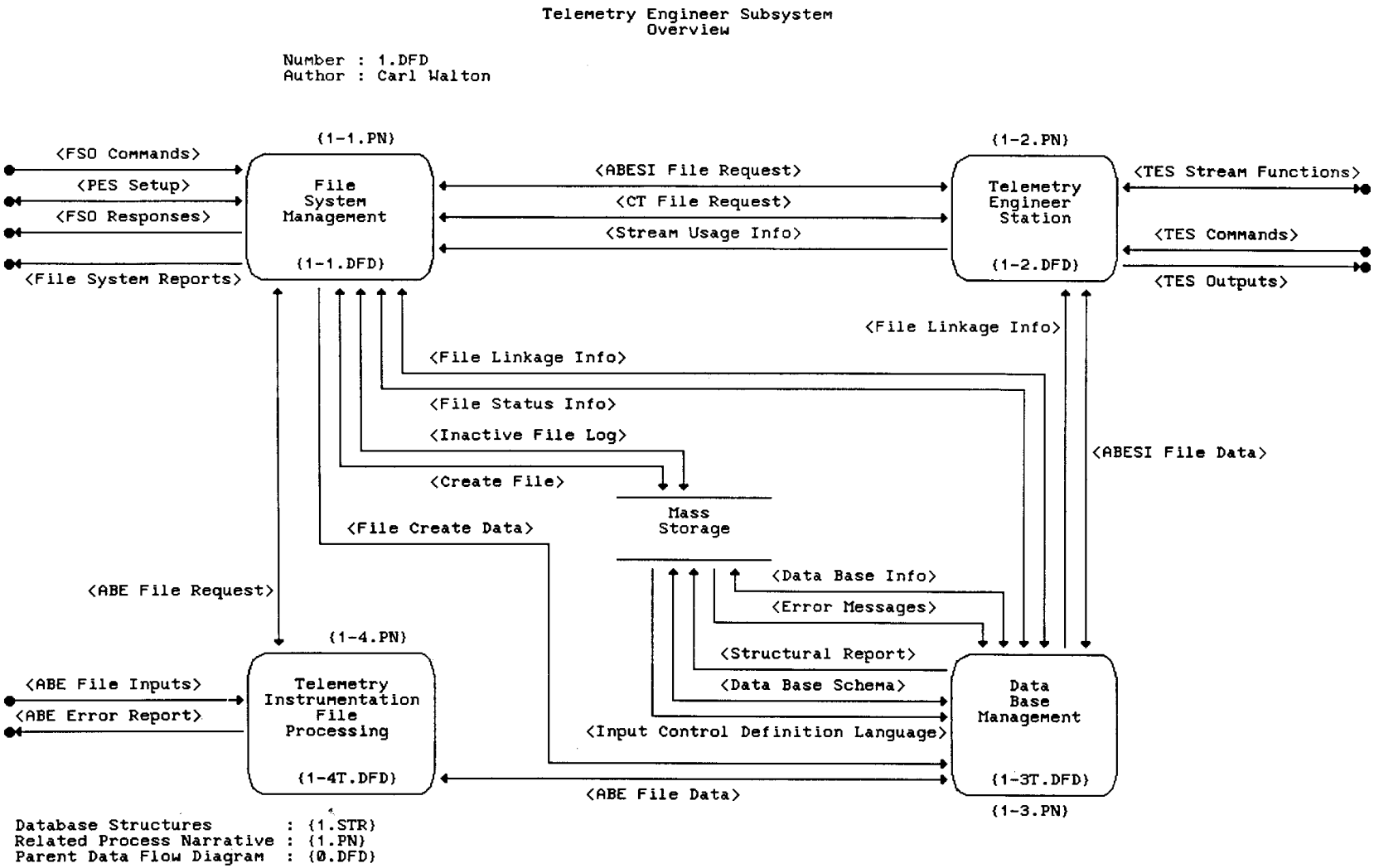


FIGURE 4. TELEMETRY ENGINEER SUBSYSTEM OVERVIEW