Top Down Implementation Plan for System Performance Test Software

G. N. Jacobson and A. Spinak
Operations Sustaining Engineering Section

This article describes the Top Down Implementation Plan used by the Operations Sustaining Engineering Section for the development of System Performance Test software during the Mark IV-A era. The plan is based upon the identification of the hierarchical relationship of the individual elements of the software design, the development of a sequence of functionally oriented demonstrable steps, the allocation of subroutines to the specific step where they are first required, and objective status reporting. The results are meaningful determination of milestones, improved managerial visibility, better project control, and ultimately a successful software development.

I. Introduction

The Mark IV-A era represents significant changes in the operating environment and the hardware configuration within the DSN. System Performance Test (SPT) software will be needed to test and verify the operational integrity of the DSN during the Mark IV-A implementation.

The SPT software package being developed by the Operations Sustaining Engineering Section will consist of a test executive and a set of seven applications tasks. Basically, the executive distributes input data to the applications, provides resource allocation services, and performs common processing functions such as dumping, display generation, and test procedure reading. The application tasks are each designed to test a particular system. Tracking, Telemetry, Command, Monitor and Control, Very Long Baseline Interferometry, Radio Science, and Frequency and Timing will all be supported by SPT software for the Mark IV-A configuration.

The SPT software will reside in the System Performance Test Assembly (SPTA). The SPTA, which also serves as the backup Complex Monitor and Control (CMC) computer, will be a Modcomp Classic 7845. The operating system will be a modified version of the MAX IV Operating System supplied by the computer manufacturer.

The software development effort for the Mark IV-A Project, summarized above, represents the largest project undertaken by the SPT Software Support Group. To achieve the technical, budgetary, and scheduling goals of the project, a top down implementation plan has been created that always develops and maintains the SPT system in a continually cycling, demonstrable fashion. It is the intent of this paper to describe the history, justification, and components of the SPT implementation plan.

II. Background

The success of large software development efforts has improved throughout the industry. These improvements are largely attributable to the application of a technological wave of new approaches that have been loosely referred to as

structured programming. More explicitly, the new technologies include replacement of flowcharts via usage of design languages, elimination of the GOTO by confinement to a small complete set of logical constructs, increased emphasis and formalization of the role of the programming support librarian, increased emphasis on reviews via usage of either structured walk-throughs or inspection teams, and a reorganization of programming personnel into the chief programmer team formation. Unfortunately, despite the considerable progress that has been made, many projects still fail to meet their schedule, have cost overruns, and the end product never quite operates as reliably as intended. In any event, even for supposedly successful projects, the cost of software is still too high.

A major reason for these continuing software difficulties and continued high costs, despite advances in technique, is that the impact of the aforementioned technical advances is limited when constrained by the effects of traditional management techniques. All of the previously mentioned structured programming techniques deal with the programmer and programming. None deal directly with the issues of planning and managing a large-scale software development. The industry is generally using the same planning and managing approaches it has always used, and these have frequently proven to be unsuccessful. The result is that the manager continues to have little visibility and little effective control over the developing system. If the manager had a mechanism that permitted him to arrive at a meaningful implementation plan; permitted him to objectively assess the project's status as it developed; provided him clear visibility of the development activity; considered cost, schedule, manpower and the chosen design, then the manager would be in a position to truly manage the project and lead it to a successful conclusion at minimal cost.

The SPT top down implementation plan fills this gap. The SPT plan is to management what structured programming is to the programmer. As with structured programming, which is complemented by the SPT plan, the SPT plan improves visibility, meaningfulness and orderliness. It allows the manager to start the project off on the right path, closely monitor the software development as it progresses, and ultimately to bring the project to the desired successful conclusion.

III. Implementation Plan Selection Criteria

The top down method appears to have been first espoused by Dr. H. D. Mills of IBM. Though this discussion, and other discussions in the computing literature, advocated top down implementation, little advice was presented on how to plan a top down implementation. Stating that a computer program should be implemented in a top down sequence is insufficient for a large software development. Due to the complexity of its hierarchical structure, literally thousands of top down implementation sequences may be possible. Thus, selecting the proper top down implementation sequence becomes a very significant issue.

For example, one could implement all the subroutines at a particular level for the entire system, followed by all the subroutines at the next level across the system, and so on, until finally the bottom level subroutines are implemented. There are those in the industry who advocate this sequence. This would be top down, but in our opinion, represents an inferior implementation sequence. This is because bottom level subroutines usually are required to provide a demonstration of a complete operational system function. Thus, for most of the system's development, very few operational functions would be demonstrable. However, early functional demonstrability is one of the main benefits that should be achieved from top down implementation. Alternatively, many top down sequences might conflict with expected equipment delivery dates.

Thus, selecting the most appropriate top down implementation sequence is an important issue. The SPT plan is using a comprehensive methodology which has been developed for creating and maintaining an optimal top down implementation plan. This technique provides broad benefits to the ensuing software development.

IV. Top Down Concepts

Top down design refers to a method of designing a computer program wherein higher level or calling segments are designed before lower level or called segments. This does not mean that all segments at one level must be designed, or named, before creating the design, or name, of any segments at the next level. It means that if one were to consider the system's subroutine hierarchy as a treelike structure, then along each branch of the tree, subroutines are defined and chosen for design, starting from the top of the hierarchy and working down.

Top down implementation refers to the development of a computer program in a downward hierarchical sequence along each branch of the program's subroutine hierarchy. Design, documentation, coding, integration and testing usually are

concurrently performed on different portions of the developing system. In a top down sequence, these are performed along each branch simultaneously under development.

V. Preparation of the SPT Top Down Implementation Plan

A viable software implementation plan can only be prepared after a sufficient quantity of system analysis activities have occurred and before the detailed implementation has begun. The plan is then used to launch the implementation phase for a large-scale software development. In operation, the SPT approach is based upon the utilization and interplay of three documents. They are the Subroutine Hierarchy, the Network of Demonstrable Functions (NDF), and the Software Status Report.

In a nutshell, the Subroutine Hierarchy represents a design abstract for the computer software. The Network of Demonstrable Functions represents a functional abstract of the operational system. The Software Status Report relates the software design to the NDF system functions for the purpose of scheduling the software and maintaining the status of software development. The main point of this nutshell description is that attentive preparation of these documents results in a meaningful schedule that allows management to have real visibility in areas such as the software's true status, cost to completion, and time to completion.

VI. Subroutine Hierarchy

The Subroutine Hierarchy is a high-level representation of the structure of the hierarchical design of the computer program. It readily conveys a high-level image of the design being represented, showing all of the parts constituting the design, their hierarchical relationship to each other, their categories and, to a degree, their functions. All of the segments must represent small subroutines, perhaps averaging 25 to 50 higher-order language statements.

The Subroutine Hierarchy evolves as the design and the software evolve. Initially, when the implementation plan is first prepared, the Subroutine Hierarchy represents the intended design structure of the computer program. At completion of the software development, it represents the actual structure of the computer program. At all intermediate stages, it is kept current and represents the currently projected structure of the program.

For a large computer program, with perhaps one thousand or more subroutines, the subroutines may be treated in a statistical manner for the purposes of making estimates, schedules and plans. This is one of the principles of the SPT approach. Namely, by partitioning a large computer program into its elemental pieces (subroutines), the effects of isolated misjudgement (e.g., size or complexity) relative to any individual subroutine tend to average out over the total program development, and do not affect the overall outcome. The effect of frequent misjudgement of the same characteristic of many subroutines (e.g., development time) tends to become quickly apparent and serves as a reliable indicator of development trends and ultimate results (if not corrected).

The Subroutine Hierarchy enables the software designers to conveniently conceptualize about the program and its parts, and to visualize the hierarchical organization of the program. It communicates in an overall conceptual manner the structure of the design. It is the essential design element, representing the components of the program's design for the purpose of planning and tracking the implementation of that design. It thus becomes the primary determinant in estimates of cost and memory size for the computer program.

Figure 1 shows a portion of the subroutine hierarchy for the SPT Project. Due to the large number of subroutines, the hierarchical structure is represented in a horizontal rather than vertical (or treelike) manner. Varying hierarchical levels are represented by varying levels of indentation. The hierarchy identifies both the symbolic name and descriptive name of the subroutine. It identifies the particular step in the SPT implementation plan where the subroutine will be first required. (The next section will elaborate on the definition of steps.) Only the first occurrence of a subroutine in the tree is expanded to the bottom level. Subsequent occurrences of any subroutine use a reference number to identify the line number of the first occurrence. If the subroutine itself invokes other subroutines, an asterisk is used to indicate the full expansion can be found at that first occurrence.

VII. Network of Demonstrable Functions

The structure of the software design and the identification of the constituent subroutines have been described as part of preparing the Subroutine Hierarchy. No discussion has yet occurred relative to the development sequence of these subroutines, nor relative to the individual milestones that will be scheduled and tracked during the development. This is where the NDF comes into play.

The NDF is that part of the SPT implementation plan that identifies the individual increments, or steps, and the sequence of development for those steps. The steps are scheduled, developed, tracked, integrated, tested and eventually internally accepted. In other words, on the surface it is a "Pert-like network." Beneath the surface there are a number of aspects of

the NDF that must be explained before its value can be fully grasped.

First of all, the NDF must be created by personnel that have an in-depth functional understanding of the application, its requirements, and the expected operational characteristics of the system. The personnel assigned to the NDF task will have acquired the necessary knowledge as a result of their prior system analysis activities. If they do not have this knowledge, they must first acquire it before they can hope to create a meaningful, detailed, functionally oriented plan of demonstrable steps.

Secondly, the NDF steps are oriented towards functions from the user's standpoint, not from the programmer's standpoint. For example, "output directive/menu index" is a typical step. This is as opposed to "build test configuration table," which would occur internally within the computer and not provide the user direct observation of the step having occurred. On the other hand, a step such as "print test configuration table" could be demonstrated to the user. Successful demonstration of this would imply successful construction of the test configuration table.

This leads us into the third important aspect of the NDF. To the maximum extent possible, steps of the NDF should be readily demonstrable to an observer who is not a programmer. Those few steps that are not readily demonstrable to such an observer must, nevertheless, still be demonstrable. This demonstrability is the only basis upon which an objective determination can be made as to the completion of the step.

The principle of demonstrability leads us to a fourth important aspect of the NDF. The development sequence of demonstrable steps must correspond to a natural functional sequence of increasing functional capability. To put it another way, from the user's operational standpoint, it must be a sequence which demonstrates "first things first."

A fifth important aspect of the NDF is that the steps must each add on to an already cycling system. Each new step must be directly integrated into the cycling system, producing a continuously increasing functional capability that is always demonstrable. Steps required to demonstrate a new step must be implemented and integrated prior to the integration of the new step. In terms of subroutines, this means that for a particular step, those subroutines that are required for invoking a particular segment of that step must be implemented as part of that step or as part of a prior step. In order words, the design must be implemented in a top down sequence along each branch of the subroutine hierarchy. Lower level subroutines that are not required for demonstration of the particular step are to be left as stubs until a step requiring those subroutines is undertaken.

One final aspect of importance is that the steps must fit together to comprise functional paths of the system. In actuality, to create the NDF, the functional paths are defined first, and the paths are then broken down into a sequence of steps. Each path corresponds to a relatively independent (but not necessarily totally independent) major function of the operational system.

As an example, the Telemetry Path of the SPT NDF is shown in Fig. 2. The Telemetry Path itself consists of a main path, a long loop path, and an Automatic Total Recall Subsystem (ATRS) path. For ease of reference, each step is given a number, preceded by a path identifier. Increments of 10 are used to allow insertion of additional steps, should this prove necessary because of changing requirements or priorities. Dashed lines between paths imply dependencies; with respect to Fig. 2, implementation of the ATRS path is dependent upon completion of the capability to accept directives.

VIII. Software Status Report

The Software Status Report ties the Subroutine Hierarchy and the NDF together by relating the design elements to the demonstrable steps. This is the fundamental point from which the value of the Software Status Report, and even the SPT Implementation methodology, is derived. The Software Status Report meaningfully relates the design to demonstrable functions and the corresponding schedules.

In concept, the document is very simple. For each step from the NDF, the corresponding required subroutines from the hierarchy are listed. Each subroutine is allocated to a single step, the first step from the NDF that requires the particular subroutine. Consequently, the subroutines listed under a particular step are just those subroutines still required for demonstration of the particular step's function. Other subroutines may also be required for the step, but they would not be listed with the step if some prior step already required the subroutines. For status tracking purposes, columns are provided where design, code, documentation, test size and other status fields can be checked off for each subroutine. These fields will be recorded as complete or will contain the date set for completion. No attempt is made to allow percentage estimates of completion by the programmer. Statistical data provided in the Software Status Report is based on treatment of individual segments as statistical equivalents. In addition, the Software Status Report includes a description of each step, i.e., the function that is being demonstrated by

the particular step. The report also identifies the qualifications or limitations, if any, that apply to the step's demonstration and the requirements that the step fulfills.

Whereas in concept the Software Status Report is very straightforward, creation of the report requires a thorough functional understanding of the system and of the corresponding design as represented by the Subroutine Hierarchy. Only with such knowledge as a base could the programming staff hope to allocate specific subroutines to each NDF step.

Thus, the Software Status Report contains all of the steps from the NDF and all of the assigned subroutines from the hierarchy, along with the development status for each subroutine and step. With automated support, highly objective status reports are easily generated from this data base. Technical and administrative management are provided accurate visibility into the status of the total software development.

Figure 3 shows the Software Status Report for a typical step from the SPT Implementation Plan. Fig. 4 provides a brief description of each of the fields on the report. Fig. 5 contains a Management Summary for one of the paths on the SPT NDF.

IX. Summary

The SPT approach to top down implementation planning is based on certain premises. One of these premises is that by minimizing or eliminating large unknowns, management has the best chance of accomplishing the project's goals. If there is some large functional area for which management has little basis, other than someone's intuition, for expecting the implementation to take say six months, with a particular size staff, as opposed to say three years, then the project is in a precarious position. A large nebulous function which has only been quantified at its total level by intuition, even though based on experience, is a dangerous unknown. The obvious way to get better control of a big unknown is by reducing it to many small pieces, some of which may be small unknowns. To put it in other terms, analyzing the task and breaking it down into many smaller pieces eliminates the risk of the large unknowns. There may still be some unknowns or surprises, but the potential absolute effect of a misjudgement relative to a small task is going to be inherently smaller than for a misjudgement associated with the much larger original task. An important additional aspect is that in the process of decomposing the original function, understanding occurs, and for the most part, comprehension replaces intuition.

Also, by decomposing a system into a large number of small pieces, a point is reached where the individual pieces can be treated for planning purposes as statistically equivalent. At the management level, the differences in size or complexity of individual small subroutines is of minimal importance. As subroutines are implemented, actual data should be used to update the estimated statistical characteristics of the average subroutine. For example, suppose an original memory allocation of 128K is made for 2000 subroutines. This averages 64 memory cells per subroutine. Suppose, after 200 subroutines have been implemented, 15000 cells have been used. This would show an actual average of 75 cells per subroutine with the trend total being 150K for all 2000 subroutines. Thus, with only 10% of the segments implemented, a reliable danger signal has been raised, and the signal includes the magnitude of the forecasted overrun. With such an early warning, management still has time to take some appropriate effective action to act upon the issue before it becomes an actual problem.

The key basis for planning and tracking the implementation is assigning the implementation of each subroutine to a single step. This is where it all comes together. But it must be done with a great deal of care and precision. The correlation between the Subroutine Hierarchy and the Software Status Report must be accurate. When design or plan changes occur, and they will, changes must be made to both documents. Both of these documents should be looked upon as evolving documents, but they must evolve concurrently and in parallel.

Why does this "single step" premise form the basis to the SPT approach to implementation planning? Because everything is accounted for. Each subroutine appears for implementation on only one step – the first step that requires the subroutine. The effort required for each step can be considered to be a function of the number of subroutines in the step. The programming implementation budget can be spread over the steps in proportion to the number of segments in each step. Then, if you are on schedule, you are on budget. Subroutines don't appear redundantly (on more than one step) to confuse the bookkeeping. Everything balances and all subroutines are able to be tracked. A full decomposition of the system into subroutines and a careful and complete assignment of those subroutines to a series of well-defined, demonstrable steps is of fundamental importance to successful usage of the SPT methodology.

SYMBOLIC NAME 1 2 3 4 5 6 7 8 9 *	DESCRIPTIVE NAME	CLASS /STEP	LINE NUM	REFER NUM
. . . . . CSTMAC	ACTIVE MODE PREDICTOR	C90	287	
. . . . . . CDBCOM	DOUBLE INTEGER COMPARE		288	197
. . . . . . CSHDGB	HSD BIT ACQUIRE		289	195
. . . . . . CDBADD	DOUBLE INTEGER ADD		290	277
. . . . . . CSTMEV	EVENT PREDICTOR	C50	291	
. . . . CDBADD	DOUBLE INTEGER ADD		292	277
. . . . CSHDGB	HSD BIT ACQUIRE		293	195
. . . . CDBSUB	DOUBLE INTEGER SUBTRACT		294	284
. . . . CDBCOM	DOUBLE INTEGER COMPARE		295	197
. . . . CSRCRP	RECALL QUEUE RESPONSE PROC.	C90	296	
. . . . . CSCHEK	STATUS CHECK		297	258*
. . . . CSSUPD	SUSPEND RESPONSE PROC.	C130	298	
. . . . . CSTMSU	MODEL RE-ADJUST	C130	299	
. . . . . . CDBCOM	DOUBLE INTEGER COMPARE		300	197
. . . . . . CDBADD	DOUBLE INTEGER ADD		301	277
. . . . . . CDBSUB	DOUBLE INTEGER SUBTRACT		302	284
. . . . CSCCLO	CONTROL CENTER RESPONSE PROC.	C60	303	
. . . . . CSHDGB	HSD BIT ACQUIRE		304	195
. . . . . CSCHEK	STATUS CHECK		305	258*
. . . . CSMOUP	MODE CHANGE RESPONSE PROC.	C100	306	
. . . . . CSHDGB	HSD BIT ACQUIRE		307	195
. . . . . CHXASC	HEX TO ASCII CONV.		308	217
. . . . . GEFMSG	DISPLAY EVENT MESSAGE		309	31
. . . . . CSCHEK	STATUS CHECK		310	258*
. . . . . CSTIRM	RADIATION TIME TEST		311	281*
. . . . CSTWOV	WINDOW OVERRIDE RESPONSE PROC.	C120	312	

Fig. 1. Subroutine Hierarchy

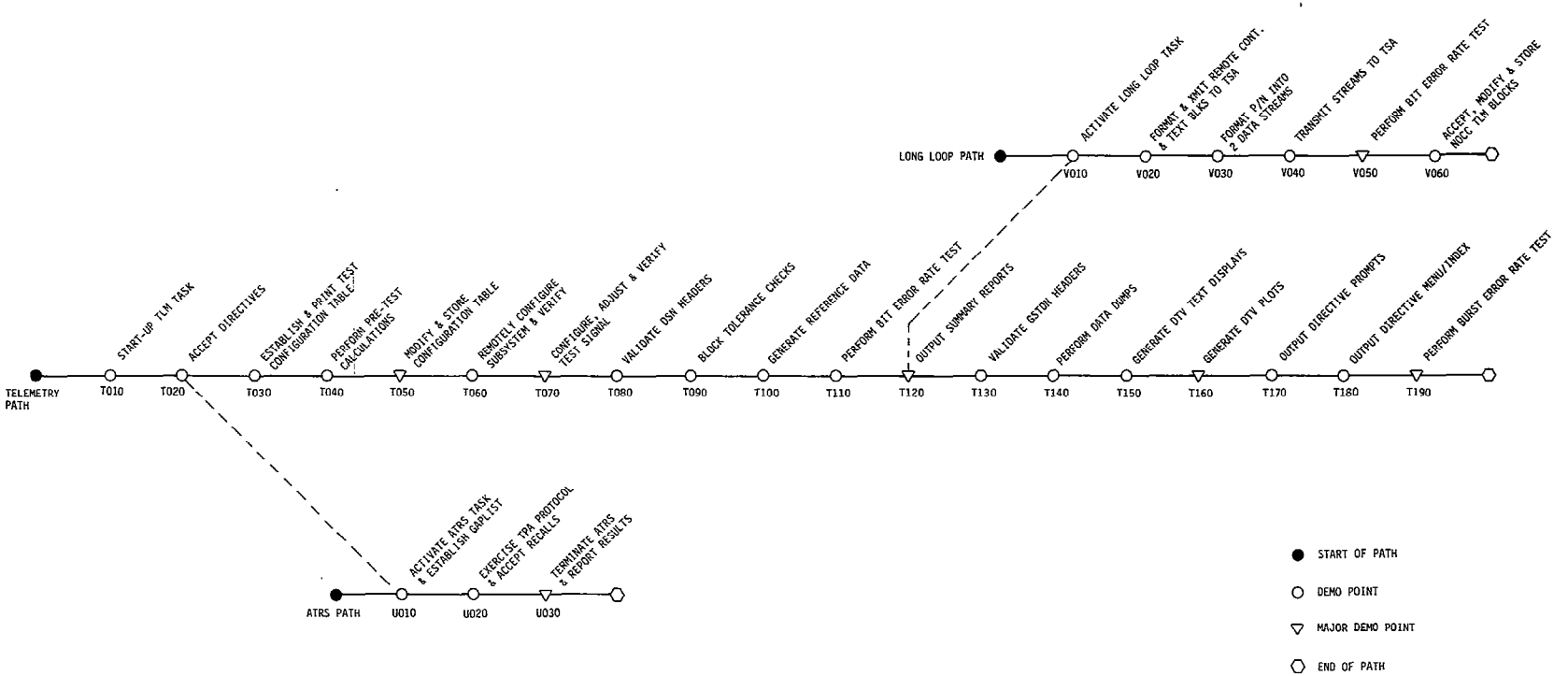

Fig. 2. Network of demonstrable functions—telemetry path

STEP: 010 Transfer Operator Entry to SYMBIONT

DESCRIPTION:

This step provides an operator entered directive to the Symbiont.

INTENT:

This step demonstrates that an operator entered directive is placed in an SSB and entered in the Symbiont Queue. This step simulates the eventual LMC, LAN Handler, FIDM to Symbiont interface. This also provides a basic buffer and queue management mechanism.

SOURCE:

AUTHOR:	A. Spinak	
DUE DATE:	03/02/82	
COORDINATOR:	T. GREER	
STATUS:	DATE	TYPE
DSGN RVU:	02/09/82	TEAM
TEST:	03/11/82	ACCEPTED
ANOMALIES:	0	

NOTES:

Most or all of this step's software is in the nature of a temporary workaround or Environmental Interface Module (EIM). To prove, the SSB and the Symbiont Queue should be checked. No queue boundary conditions will be demonstrated.

SEGMENTS	DESCRIPTION	CL	DESIGN DATE	DESIGN PERSON	ST	CODE DATE	CODE PERSON	ST	QA	CMP	LINES
SIOINP	OPERATOR 2 SPT SIMULATION	I	02/01/82	MJB	*	02/11/82	MJB	*		*	21
SIOPOM	PARSE OPERATOR MESSAGE	I	02/02/82	MJB	*	02/11/82	MJB	*		*	21
SIOPAK	PACK CD SSB	I	02/03/82	TO	*	02/11/82	TO	*		*	30
SIOCBB	LOAD TEST COMM BUFF BLOCK	S									0
SIOSSB	LOAD TEST SSB BLOCK	S									0
SIOLNK	CHANGE LINK ID	S									0
ORQENQ	PLACE NODE ON QUEUE	I	02/02/82	TCG	*	02/16/82	MJB	*		*	51
ORQINI	INIT FREE Q NODE POOL	I	02/05/82	TCG	*	02/16/82	TCG	*		*	14
OPOPQU	POP FREE QUEUE NODE	I	02/05/82	MJB	*	02/16/82	TO	*		*	15
GBFQUE	I/F TO QUEUE ENTRY	I	02/02/82	TO	*	02/12/82	TO	*		*	35
GQINIT	I/F TO INIT QUEUES	I	02/04/82	TO	*	02/12/82	TO	*		*	14
OPSHQU	PUSH FREE QUEUE NODE	I	02/05/82	MJB	*	02/12/82	MJB	*		*	16
SPINIT	SPT INITIALIZATION	I	02/10/82	TCG	*	02/15/82	TCG	*		*	42
GBINIT	I/F TO INITIALIZE BUFFERS	I	02/22/82	MJB	*	03/05/82	TO	*		*	58
GBPOOL	I/F TO RELEASE BUFFER	I	02/22/82	MJB	*	03/05/82	MJB	*		*	55
GETBUF	I/F TO GET BUFFER	I	02/22/82	MJB	*	03/05/82	MJB	*		*	56

Fig. 3. Software status report—step 010

STEP: NUMBER AND NAME OF THE STEP

DESCRIPTION:

DESCRIPTION OF THE SYSTEM FUNCTIONS AND CAPABILITIES IMPLEMENTED IN THIS STEP AND A CONCISE SUMMARIZATION OF THE DEMONSTRATIONS TO BE TESTED' (UP TO 3 LINES)

INTENT:

EXPLANATORY INFORMATION THAT IS USEFUL IN INTERPRETATION AND COMPREHENSION OF THE STEP. (UP TO 5 LINES)

SOURCE:	REFERENCE TO THE REQUIREMENTS MET BY THIS STEP.	
AUTHOR:	ORIGINATOR OF THIS STEP INPUT.	
DUE DATE:	DATE WHEN THE STEP WILL BE COMPLETED AND READY FOR ACCEPTANCE.	
COORDINATOR:	COORDINATOR OF THE IMPLEMENTATION OF THE STEP AT THE DETAIL LEVEL.	
STATUS:	DATE	TYPE
DSGN RVU:	DESIGN REVIEW DATE	TYPE OF REVIEW (TEAM, CDE OR PROG)
	TEST STATUS DATE	TYPE OF TESTING (CHECKOUT OR ACCEPTED)
		(IF TYPE IS BLANK, DATE IS PLANNED DATE)
		(ELSE DATE IS ACTUAL DATE)
ANOMALIES:	NUMBER OF OUTSTANDING ANOMALIES	

NOTES:

NOTES WHICH WOULD BE HELPFUL IN EXPLAINING THE STEP AND ITS IMPLEMENTATION. (UP TO 5 LINES)

THE SEGMENT COLUMNS ARE AS FOLLOWS:

STEP NO	– STEP NUMBER
SEGMENTS	– SEGMENT NAMES
DESCRIPTION	– SEGMENT DESCRIPTION
CL	– CLASSIFICATION CODE (I-IMPLEMENT, S-STUB, U-UNDEFINED)
DESIGN DATE	– PLANNED DATE OF DESIGN COMPLETION
DESIGN PERSON	– INITIALS OF PERSON RESPONSIBLE FOR THE SEGMENT'S DESIGN
DESIGN ST	– DESIGN STATUS (* IF COMPLETED)
CODE DATE	– PLANNED DATE OF CODE COMPLETION
CODE PERSON	– INITIALS OF PERSON RESPONSIBLE FOR CODING OF THE SEGMENT
CODE ST	– CODE STATUS (* IF COMPLETED OR IF REJECTED)
QA	– QA STATUS CODE (* IF ACCEPTED)
CMP	– COMPILATION CODE (* FOR CLEAN COMPILE)
LINES	– NUMBER OF LINES IN ACCEPTED SEGMENT

Fig. 4. Software status report description

STEP NUM	STEP NAME	C	UNIQ	CUM	DESGN	CODE	CMP	TEST	ACEPT	DUE DATE	LINES
C01	Initialize Command Task		16	16	0	0	0	0	0		0
C02	Generate Command File		9	25	0	0	0	0	0		0
C03	Accept & Check Incoming Block		12	37	0	0	0	0	0		0
C04	Process Control & Status Block		8	45	0	0	0	0	0		0
C05	Process Event Block		9	54	0	0	0	0	0		0
C06	Transmit Block to CPA		12	66	0	0	0	0	0		0
C07	Transmit File to CPA		16	82	0	0	0	0	0		0
C08	Recall File Directory		2	84	0	0	0	0	0		0
C09	Attach File to Queue		7	91	0	0	0	0	0		0
C10	Initiate Command Radiation		2	93	0	0	0	0	0		0
C11	Modify Standards-and-Limits Table		18	111	0	0	0	0	0		0
C12	Transmit Additional Directives		15	126	0	0	0	0	0		0
C13	Suspend, Abort, Resume Radiation		9	135	0	0	0	0	0		0
C14	Verify Command Bits		11	146	0	0	0	0	0		0
TOTALS			146	146	0	0	0	0	0		0
PERCENTS				100	0	0	0	0	0		0
PROJECTION											

Fig. 5. Software status report management summary