7B. WESTINGHOUSE DESIGN APPROACH FOR SOFTWARE DEVELOPMENT FOR THE INTEGRATED PROTECTION SYSTEM

Software instructions will be stored in read-only memory and hence will not be alterable by the hardware system.

This section describes the Westinghouse approach which will be used in the development of the software for use in the Integrated Protection System. The following areas are covered in this section:

Description of the Modularity Concept	(Section 7B-1)
Control Over Nesting and Interrupts	(Section 7B-2)
System Software Organization	(Section 7B-3)
Programming Language	(Section 7B-4)
Software Test and Verification	(Section 7B-5)
Software Documentation	(Section 7B-6)
Applicable Industry Standards	(Section 7B-7)

The Westinghouse approach provides a very structured, disciplined, and systematic program for all phases of the software design from system specification, through design and coding, to testing, verifying, and documenting the final package.

7B.1 DESCRIPTION OF MODULARITY CONCEPT

In order to ensure that the software performs its design function, constraints will be applied to the software structure. The most effective method is to separate the software into sections called modules which have clearly specified interfaces.

A module is defined as the smallest element of software independently tested and stored in the library. A module has well defined interfaces and a limited number of states. Module state in this context means a module function. The

statement expresses the requirement that module complexity should be limited in such a way as to make the module manageable and easily understandable as, for example, in the execution of a decision table. Limiting module state is a principle rather than an absolute number. A module may be considered as a function routine or a subroutine used in a repeated manner. A module represents a mathematical or logical expression. A module will have only one entry and one exit point. A module may name other modules to be called as a function or subroutine procedure, but this "nesting" will be constrained as discussed in Section 7B.2. A collection of several modules can be considered as another level of module, per se, if it also performs a well defined task and has only one entry and exit point. In other words, the modular concept can be generalized by applying the same requirements to varying levels of software.

The modular approach when used in conjunction with structured programming and top-down development offers several advantages (see Section B-3). Top-down development minimizes the writing of any code for which testing is dependent on another code not yet written or one for which data is not yet available. It minimizes the requirement to integrate many pieces of independent code and allows testing to be conducted in parallel with the design phase. Structured programming imposes a strict procedure on writing the module software and on the integration of modules into the system.

The software generation procedure is shown in block-diagram form in Figure 7B-1.

In order to increase reliability, extensive use will be made of the library concept. The library will include collections of verified modules which will be used by higher level modules. Stringent acceptance requirements will be placed on the programs before they are entered in the library. Similarly, procedures will ensure that only authorized modifications are made to the modules in the library. The linking of library modules into a program will be done automatically by a linking leader utility program. The library editor will enable the listing of the contents of a library, deleting, inserting or replacing a module, or merging two library files into a single one.

The use of libraries allows for the orderly development, release, and maintenance of the protection system software. In the development phase, the software module source is written, the module is assembled, and the module is tested. If necessary, it will be modified, reassembled, and retested. When the programmer is satisfied, the module is reviewed by the librarian for proper documentation and coding style. When satisfied, the librarian assigns to the module official identification and files the source listing. The module is now ready for verification. When the verification team is satisfied with the performance of the module, it is ready for release. The librarian assigns it revision number 00 and the relocatable object (the assembler output) of the module is placed on the module library. All further revisions increment the revision number and must be documented in sufficient detail to reproduce the transition from one revision level to the next. All revision levels must be verified by the verification team.

Examples of several modules of different complexity are:

1. Multiply/divide module
2. Lead/lag compensation module
3. DNBR calculation module

The module described in 3 is a functional module, which may consist of several modules of type 2. In turn, modules of type 2 will consist of several modules of type 1. Module 1 is the basic software module.

7B.2 CONTROL OVER NESTING AND INTERRUPTS

In real-time software systems, such as for the protection system, the number of allowable interrupts has to be limited. This constraint is introduced since it is extremely difficult to test and verify software in the presence of interrupts. Interrupts in general have a random nature. By limiting the number of interrupt levels, the cases when the interrupt actually occurs can be accounted for during the test and verification process. The Westinghouse integrated protection system software, will use no software interrupts. Software interrupt, in this context, refers to an interrupt that is always

made by hardware. In conventional systems, the software manages interrupt processing by saving the program counter, comparing priorities, and returning from interrupt. The IPS will limit interrupt in several ways. Firstly, the number of interrupts will be limited to the minimum number necessary to halt the program for such conditions as power failure, operator reset, or memory priority error. Secondly, there will be only a single level of interrupt. Thirdly, the interrupt will cause the program to branch into a specific location where execution continues without returning to the location where the interrupt occurred.

A power up sequence will be accomplished through hardware implementation and will serve to initialize the program upon return of power.

To enhance the readability of the program and to fully utilize the structured programming approach, the nesting of subroutines will be controlled. By leaving the number of nesting subroutines limited, testing and verification are more easily planned and executed, both by comparison to the functional requirements and by execution on the computer.

The single entry/single exit requirement on modules will be met by requiring called subroutines to always return to the calling module. Nesting subroutines will be controlled by this rule. Nesting refers to a subroutine which is being executed and in turn, calls another subroutine. Return from a subroutine will be made to the calling module at the first executable statement after the subroutine call. Figure 7B-2 gives a pictorial representation of the software nesting requirements. Note that each called subroutine always returns to the module which called it.

Re-entering terms refers to a subroutine which is being executed, and before its execution is completed, the same subroutine is entered again. Typically, this situation can occur in two cases. If the program is recursive, then a subroutine can call itself, either directly or through a second time before the first execution was completed. Both recursive programming and interrupt nesting will be forbidden.

7B.3 SYSTEM SOFTWARE ORGANIZATION

The protection system software will be based on the principles of top-down development, structured programming, and modularity. Basically, top-down programming means that the entire program is first layed out in one top segment. At the "top" the software will be specified in general terms. Once this is done, the next level of module will be outlined. The process is repeated as levels of program details are relegated to various levels of modules. The process is repeated until the software specification is satisfied.

Structured programming is both a design and coding technique. It encourages code readability, enforces modularity, encourages changeability, enhances maintainability, simplifies testing and permits improved manageability and accountability.

The protection system software will be organized in a simple chain structure. Figure 7B-3 depicts this structure.

The modules will be cyclically executed asynchronously. The program sequences through the chain advancing from module to module. No traceback external to modules will be permitted and there will be no nesting of interrupts as was noted in Section 7B.2. The operating system, as it is known in minicomputer-based systems, will not exist in the protection system. The chain program, as described, replaces the operating system. To further elaborate on this point, the data bus and its controller processor are not performing functions usually attributed to an operating system. This can be assured because the trip function processors do not require the operation of the data bus controller to perform the trip functions. The data bus is several two ported random access memories (RAM) connected in series. The data bus is controlled by a dedicated microprocessor which accesses information from one of the two memory ports. The function processors access information from the other port as shown in Figure 7B-4. All processors have a single loop execution sequence. There is no synchronization of the data bus

controller processor and the function processors. Information is accessed by all processors with normal memory references. Contention between processors for the shared memory is resolved by the two ported memory card.

7B.4 PROGRAMMING LANGUAGE

The most important software tool available to the programmer is the programming language. Most of the programming will be done using a high level language. Sections of the program which operate directly with external hardware or which require a very high speed will be programmed in assembly language. The compiler will output a comprehensive listing which includes the source code together with the assembly language code that was generated for each statement. An option to print an alphabetized list of variables and procedures with reference to their location in the program text is available.

Readability is the key feature of a good programming language. The language will include control structures (IF THEN ELSE, DO WHILE, etc.) to encourage structured programming and to discourage excessive use of Go To statements. The following features of the high level language will aid in the development and maintenance of reliable software:

1. Structured programming support and software modularity.

2. Explicit module interfaces with single-entry and single-exit.

3. Easy to learn and use.

4. Self-documenting.

5. Support of noncontiguous memory addressed such as read-only memory, read-write memory, and memory mapped input/output.

6. Convenient access to system software and services.

7. Provision for syntax and context checking.

8. Support of numerous data structures and variable types such as bits, bytes (8 bits), integers (16 bits), and real numbers (32 bits).

9. Support multidimensional arrays.

10. Language extensions.

The programming language can encourage good programming practices but cannot enforce them. The additional programming style guidelines will be used to improve the readability and reliability of the program.

1. The size of each software module will be limited to a few pages of code.

2. Simple control structures will be used and Go To statements will be avoided.

3. All subroutines will return to the caller directly following the call.

4. All subroutines will have single-entry and single-exit.

5. Interrupts will not be permitted.

7B.5 SOFTWARE TEST AND VERIFICATION

Verification is intended to provide an adequate level of assurance that the software actually performs the functions which are specified by the functional requirements. It gives assurance that the protection system software meets its specification and will perform its protection function.

Refer to ANS Standard on QA.

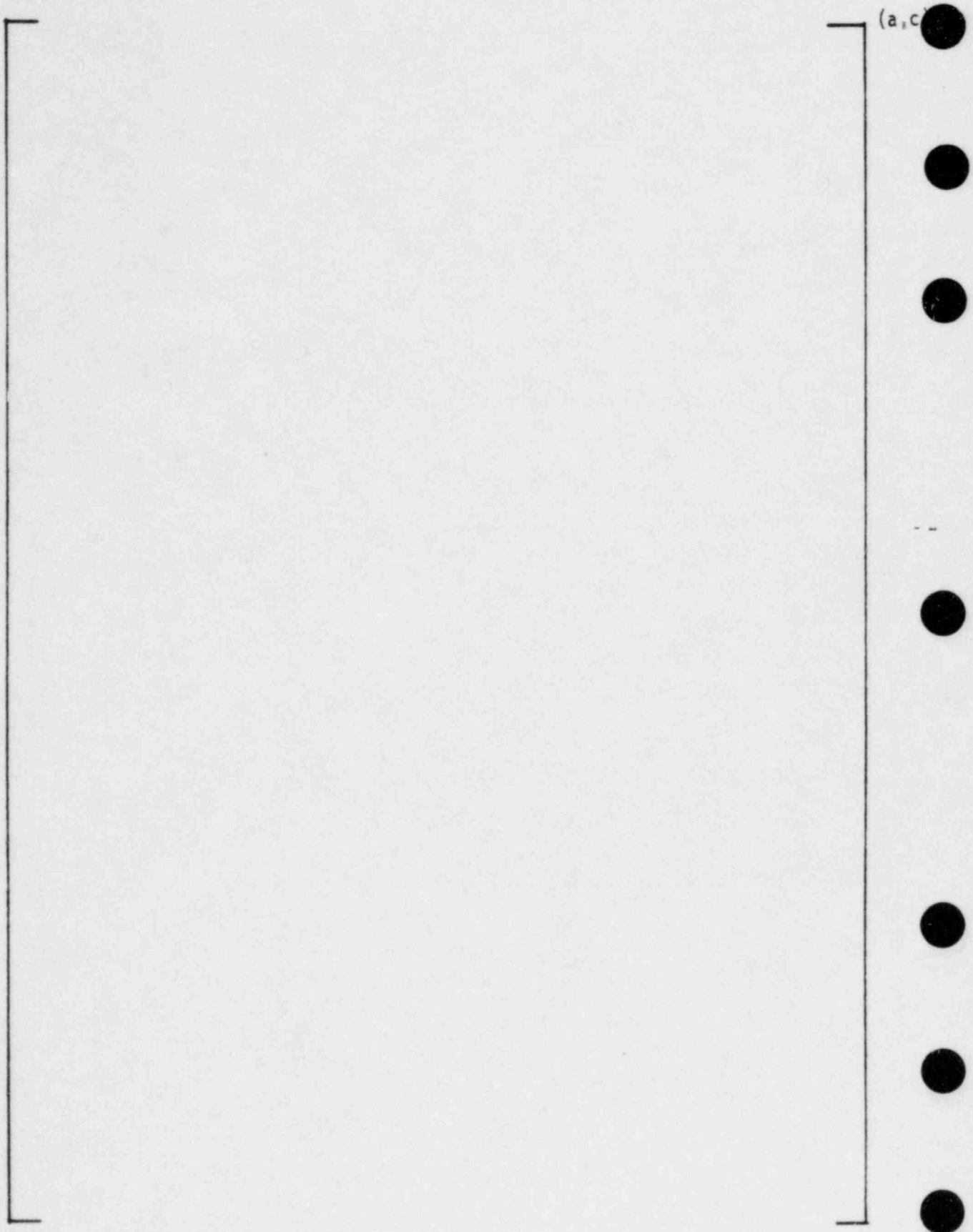


FIGURE 7B-1 SOFTWARE GENERATOR BLOCK DIAGRAM

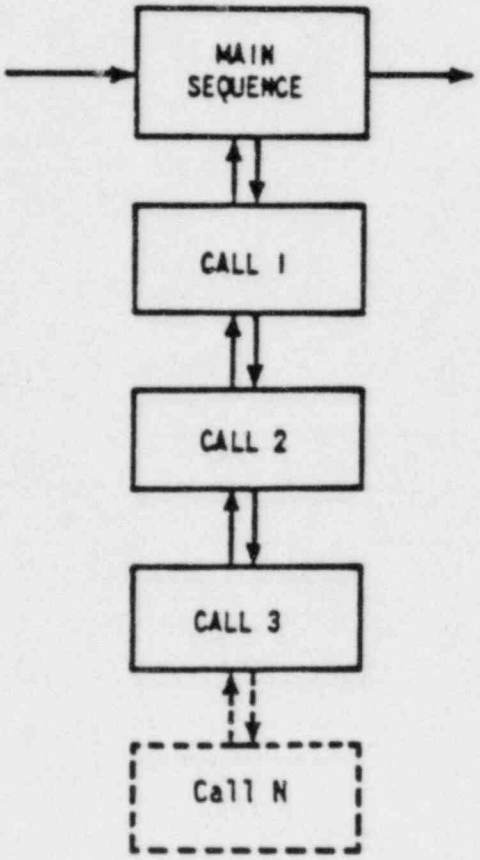


FIGURE 7B-2 SOFTWARE NESTING DIAGRAM

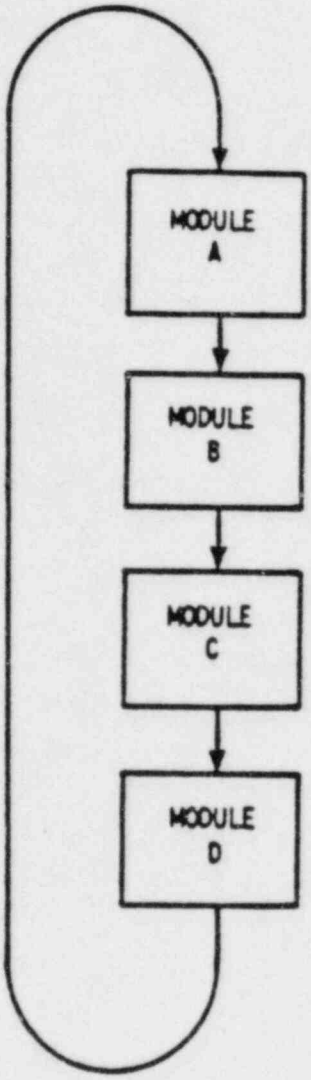
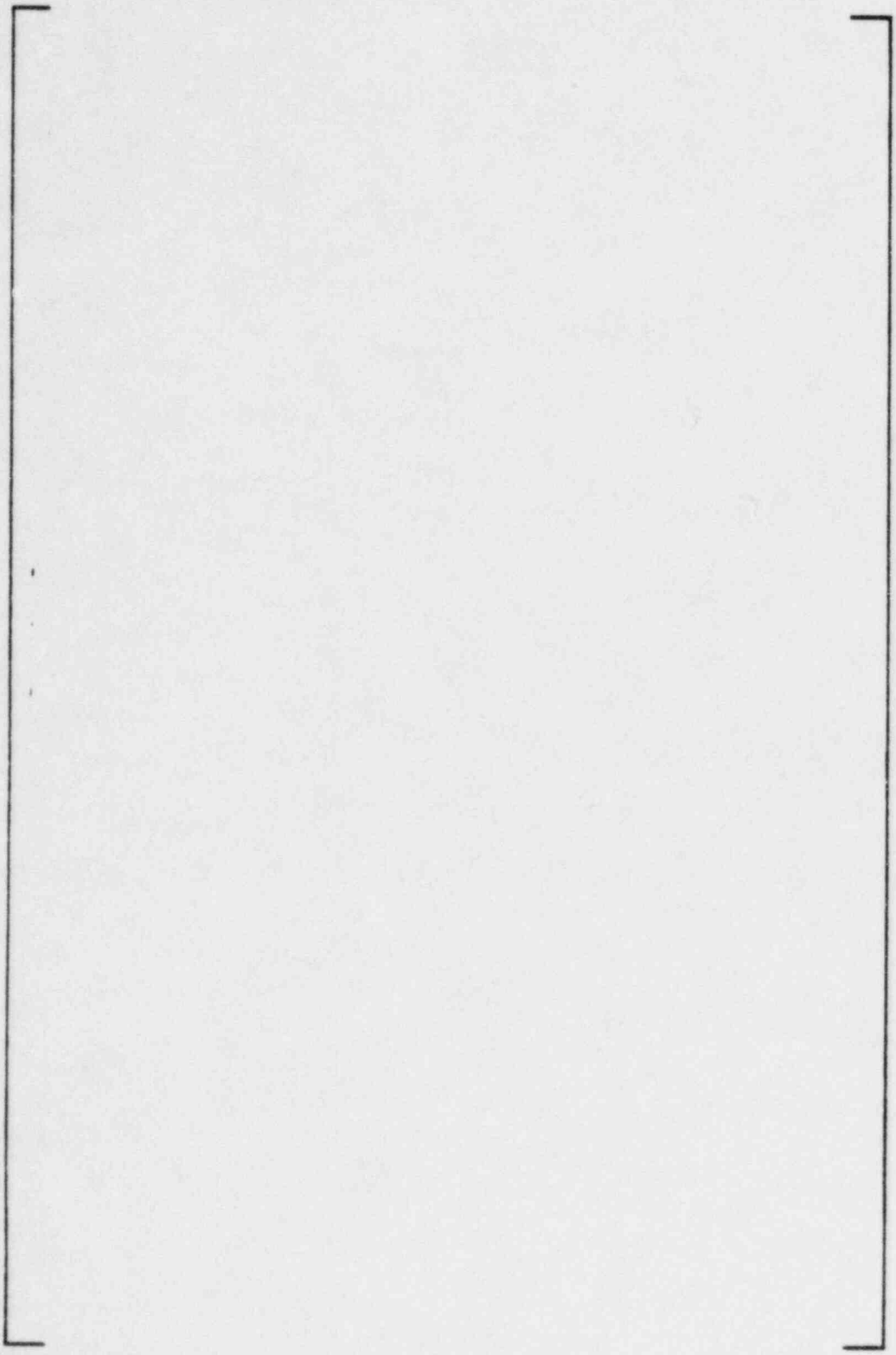


FIGURE 7B-3 SYSTEM SOFTWARE ORGANIZATION



(a,c)

FIGURE 7B-4 IPS DATA BUS