Volume II, Section 5
Table of Contents

5 Software Testing

5.1 Scope

This section contains a description of the testing to be performed by the ITA to confirm the proper functioning of the software components of a voting system submitted for qualification testing. It describes the scope and basis for software testing, the initial review of documentation to support software testing, and the review of the voting system source code. Further testing of the voting system software is addressed in the following sections:

a. Volume II, Section 3, for specific tests of voting system functionality; and

b. Volume II, Section 6, for testing voting system security and for testing the operation of the voting system software together with other voting system components.

5.2 Basis of Software Testing

ITAs shall design and perform procedures that test the voting system software requirements identified in Volume I. All software components designed or modified for election use shall be tested in accordance with the applicable procedures contained in this section.

Unmodified, general purpose COTS non-voting software (e.g., operating systems, programming language compilers, data base management systems, and Web browsers) is not subject to the detailed examinations specified in this section. However, the ITA shall examine such software to confirm the specific version of software being used against the design specification to confirm that the software has not been modified. Portions of COTS software that have been modified by the vendor in any manner are subject to review.

Unmodified COTS software is not subject to code examination. However, source code generated by a COTS package and embedded in software modules for compilation or interpretation shall be provided in human readable form to the ITA. The ITA may

inspect COTS source code units to determine testing requirements or to verify the code is unmodified.

The ITA may inspect the COTS generated software source code in preparation of test plans and to provide some minimal scanning or sampling to check for embedded code or unauthorized changes. Otherwise, the COTS source code is not subject to the full code review and testing. For purposes of code analysis, the COTS units shall be treated as unexpanded macros.

Compatibility of the voting system software components or subsystems with one another, and with other components of the voting system environment, shall be determined through functional tests integrating the voting system software with the remainder of the system.

The specific procedures to be used shall be identified in the Qualification Test Plan prepared by the ITA. These procedures may replicate testing performed by the vendor and documented in the vendor's TDP, but shall not rely on vendor testing as a substitute for software testing performed by the ITA.

Recognizing variations in system design and the technologies employed by different vendors, the ITAs shall design test procedures that account for these variations.

5.3 Initial Review of Documentation

Prior to initiating the software review, the ITA shall verify that the documentation submitted by the vendor in the TDP is sufficient to enable:

a. Review of the source code; and

b. Design and conducting of tests at every level of the software structure to verify that the software meets the vendor's design specifications and the requirements of the performance standards.

5.4 Source Code Review

The ITA shall compare the source code to the vendor's software design documentation to ascertain how completely the software conforms to the vendor's specifications. Source code inspection shall also assess the extent to which the code adheres to the requirements in Volume I, Section 4.

5.4.1 Control Constructs

Voting system software shall use the control constructs identified in this section as follows:

a. If the programming language used does not provide these control constructs, the vendor shall provide them (that is, comparable control structure logic). The constructs shall be used consistently throughout the code. No other constructs shall be used to control program logic and execution;

b. While some programming languages do not create programs as linear processes, stepping from an initial condition, through changes, to a conclusion, the program components nonetheless contain procedures (such as "methods" in object-oriented languages). Even in these programming languages, the procedures must execute through these control constructs (or their equivalents, as defined and provided by the vendor); and

c. Operator intervention or logic that evaluates received or stored data shall not re-direct program control within a program routine. Program control may be re-directed within a routine by calling subroutines, procedures, and functions, and by interrupt service routines and exception handlers (due to abnormal error conditions). Do-While (False) constructs and intentional exceptions (used as GoTos) are prohibited.

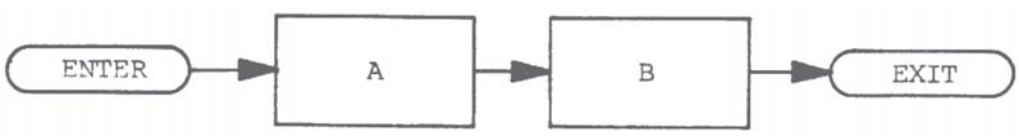
Illustrations of control construct techniques are provided in Figures 4-1 through 4-6.

- Fig. 4-1 Sequence
- Fig. 4-2 If -Then -Else
- Fig. 4-3 Do -While
- Fig. 4-4 Do -Until
- Fig. 4-5 Case
- Fig. 4-6 General loop, including the special case FOR loop

5.4.1.1 Replacement Rule

In the constructs shown, any 'process' may be replaced by a simple statement, a subroutine or function call, or any of the control constructs. In Fig 4-1 for example, "Process A" may be a simple statement and "Process B" another Sequence construct.

5.4.1.2 Figures



Control flows from “Process A” to the next in sequence, “Process B.”

Figure 4-1, “SEQUENCE”

Using the replacement rule to replace one or both of the processes in the Sequence construct with other Sequence constructs, a large block of sequential code may be formed. The entire chain is recognized as a Sequence construct and is sometimes called a BLOCK construct. In many languages, a Sequence may need to be marked with special symbols or punctuation to delimit where it starts and where it ends. For example, a “BEGIN” and “END” may be used. This allows the scope of a Sequence used as “Process C” in the IF-THEN-ELSE (Fig 4-2) to be recognized as completing the IF-THEN-ELSE rather than part of a higher level Sequence that included the IF-THEN-ELSE as a component.

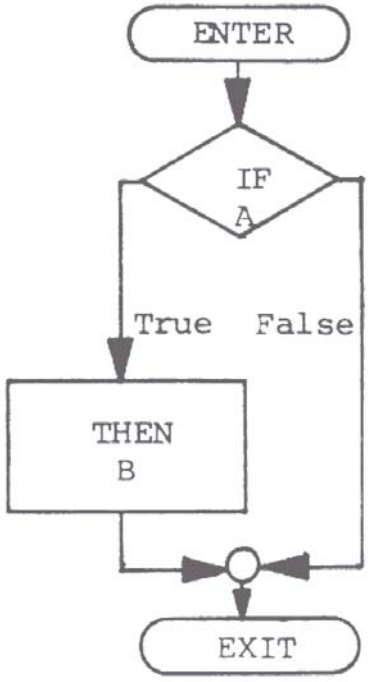


Figure 4-2, “IF-THEN-ELSE”

*****In Figure 4-2, Flow of control will skip a process pending the condition of “A."

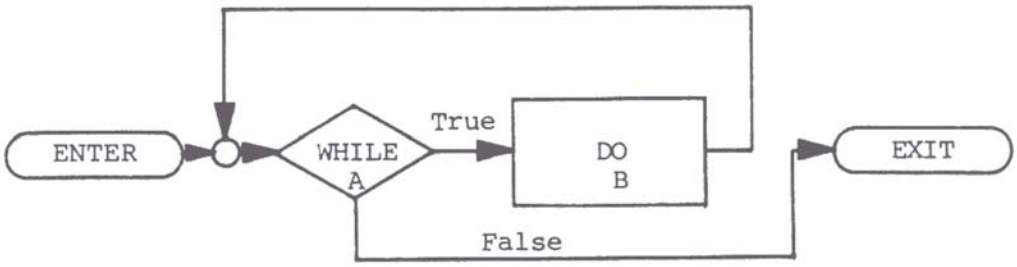


Figure 4-3, "DO-WHILE"

In Figure 4-3,condition "A" is evaluated. If found to be true, then control is passed to Process "B" and condition "A" is reevaluated. If condition "A" is found to be false, then control is passed out of the loop. Note that, if B is a BLOCK, the "DO" may be recognized as the opening symbol. A terminating symbol is needed from the language used.

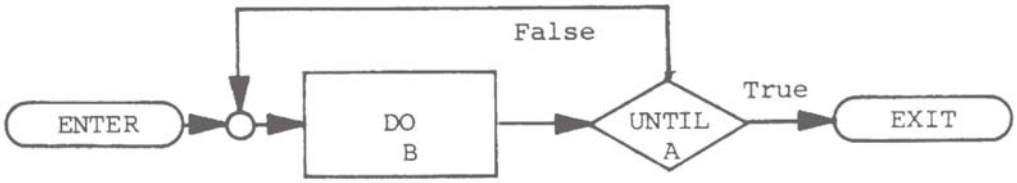


Figure 4-4, "DO-UNTIL"

Figure 4-4 is similar to a DO-WHILE, except that the test of condition A is performed after "Process B" has executed and the DO is performed upon a false "A" condition.. If condition "A" is true, control is passed out of the loop.

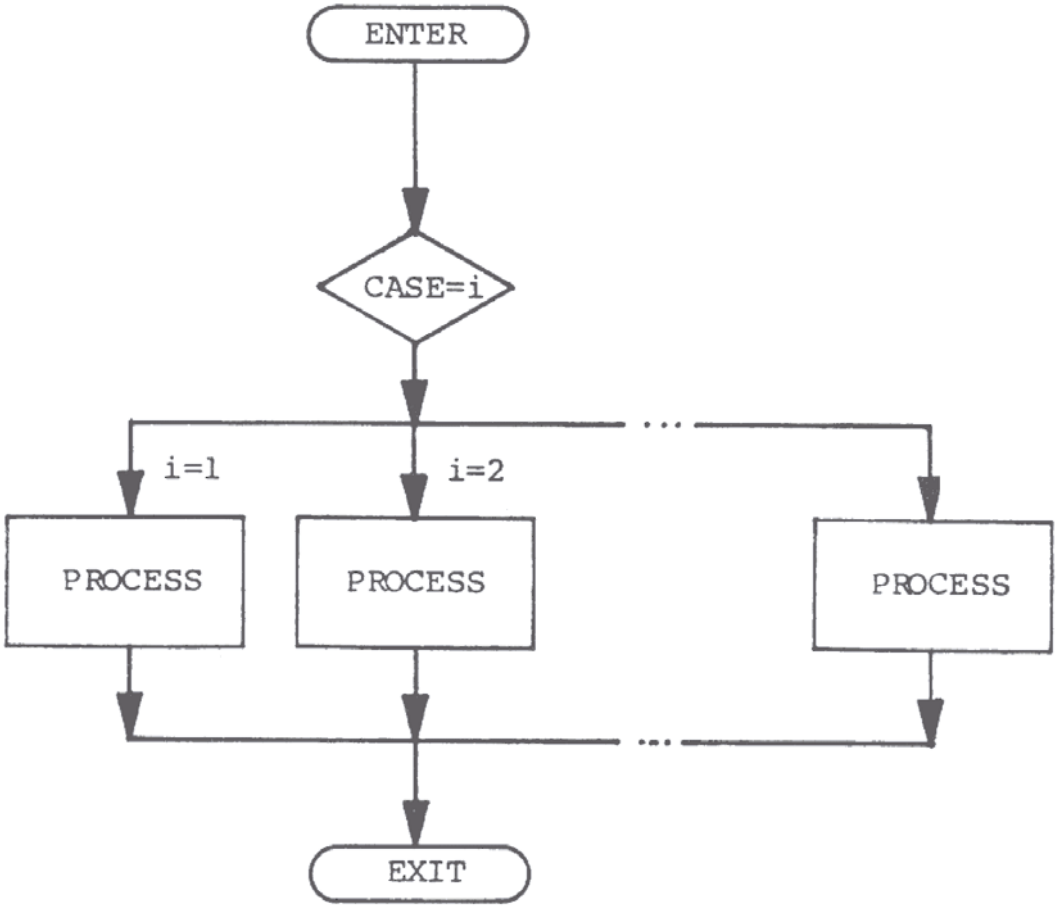


Figure 4-5, "CASE"

Control is passed to a Process based on the value of i.

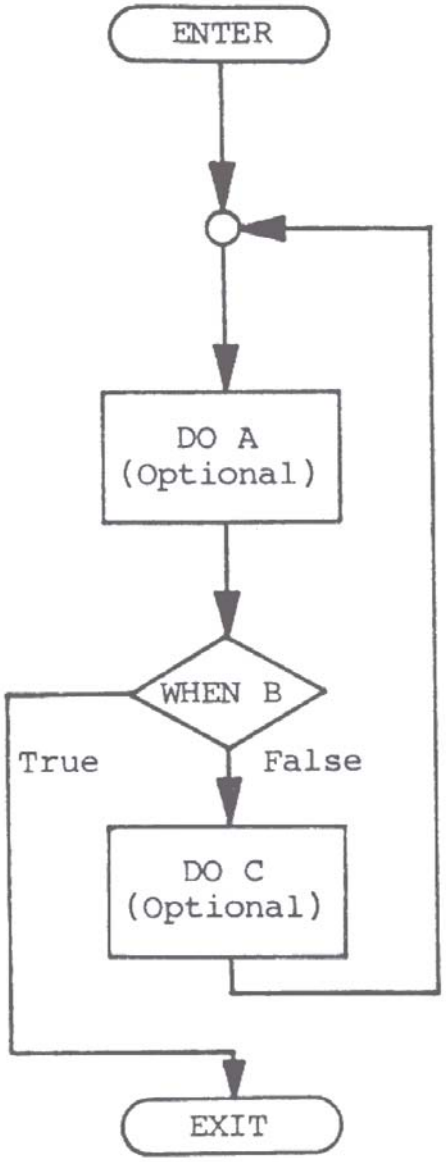


Figure 4-6, "General LOOP"

Optional process A is executed. Condition B is then evaluated. If found to be false, optional process C is executed and control is passed to process A. Condition B is then evaluated again. If condition B is true, then control is passed out of the loop.

A special case of the GENERAL LOOP is the FOR loop. The FOR is not strictly essential as it can be programmed as a DO-WHILE loop. The FOR loop executes on a counter. The control FOR statement defines a counter variable or variables, a test for ending the loop, and a standard method of changing the variable(s) on each pass such as incrementing or decrementing. For example,

"FOR c = 0; c < 10; c + 1

DO Process A;"

The counter is initialized to zero, if the counter test is false, the DO process is executed and the counter is incremented (or decremented). Once the counter test is true, control exits from the loop without incrementing the counter. The implementation of the FOR loop in many languages, however, can be error prone. The use of the FOR loop shall include strictly enforced coding conventions to avoid the common errors such as a loop that never ends.

The GENERAL LOOP should not be used where one of the other loop structures will serve. It too is error prone and may not be supported in many languages without using GOTOs type redirections. However, if defined in the language, it may be useful in defining some loops where the exit needs to occur in the middle. Also, in other languages the GENERAL LOOP logic can be used to simulate the other control constructs. Like the special case, the use of the GENERAL LOOP shall require the strict enforcement of coding conventions to avoid problems.

5.4.2 Assessment of Coding Conventions

The ITA shall test for compliance with the coding conventions specified by the vendor. If the vendor does not identify an appropriate set of coding conventions in accordance with the provisions of Volume I, section 4.2.6.a, the ITA shall review the code to ensure that it:

a. Uses uniform calling sequences. All parameters shall either be validated for type and range on entry into each unit or the unit comments shall explicitly identify the type and range for the reference of the programmer and tester. Validation may be performed implicitly by the compiler or explicitly by the programmer;

b. For C based language and others to which this applies, has the return explicitly defined for callable units such as functions or procedures (do not drop through by default) and, in the case of functions, have the return value explicitly assigned. Where the return is only expected to return a successful value, the C convention of returning zero shall be used or the use of another code justified in the comments. If an uncorrected error occurs so the unit must return without correctly completing its objective, a non-zero return value shall be given even if there is no expectation of testing the return. An exception may be made where the return value of the function has a data range including zero;

c. Does not use macros that contain returns or pass control beyond the next statement;

d. For those languages with unbound arrays, provides controls to prevent writing beyond the array, string, or buffer boundaries;

e. For those languages with pointers or which provide for specifying absolute memory locations, provides controls that prevent the pointer or address from being used to overwrite executable instructions or to access inappropriate areas where vote counts or audit records are stored;

f. For those languages supporting case statements, has a default choice explicitly defined to catch values not included in the case list;

g. Provides controls to prevent any vote counter from overflowing. Assuming the counter size is large enough such that the value will never be reached is not adequate;

h. Is indented consistently and clearly to indicate logical levels;

i. Excluding code generated by commercial code generators, is written in small and easily identifiable modules, with no more than 50% of all modules exceeding 60 lines in length, no more than 5% of all modules exceeding 120 lines in length, and no modules exceeding 240 lines in length. "Lines" in this context, are defined as executable statements or flow control statements with suitable formatting and comments. The reviewer should consider the use of formatting, such as blocking into readable units, which supports the intent of this requirement where the module itself exceeds the limits. The vendor shall justify any module lengths exceeding this standard;

j. Where code generators are used, the source file segments provided by the code generators should be marked as such with comments defining the logic invoked and, if possible, a copy of the source code provided to the ITA with the generated source code replaced with an unexpanded macro call or its equivalent;

k. Has no line of code exceeding 80 columns in width (including comments and tab expansions) without justification;

l. Contains no more than one executable statement and no more than one flow control statement for each line of source code;

m. In languages where embedded executable statements are permitted in conditional expressions, the single embedded statement may be considered a part of the conditional expression. Any additional executable statements should be split out to other lines;

n. Avoids mixed-mode operations. If mixed mode usage is necessary, then all uses shall be identified and clearly explained by comments;

o. Upon exit() at any point, presents a message to the user indicating the reason for the exit().

p. Uses separate and consistent formats to distinguish between normal status and error or exception messages. All messages shall be self-explanatory and shall not require the operator to perform any look-up to interpret them, except for error messages that require resolution by a trained technician.

q. References variables by fewer than five levels of indirection (i.e. a.b.c.d or a[b].c->d).

r. Has functions with fewer than six levels of indented scope, counted as follows:

```
int function()
```

```
{
        if (a = true)
1       {
                if ( b = true )
2               {
                if ( c = true )
3                       {
                        if ( d = true )
4                               {
                                        while(e > 0 )
5                                       {
                                        code
                                        }
                        }
                }
        }
}
}
```

s. Initializes every variable upon declaration where permitted

t. Specifies explicit comparisons in all if() and while() conditions. For instance,

i. if(flag)

is prohibited, and shall be written in the format

ii. if (flag == TRUE)

for both single and multiple conditions.

u. Has all constants other than 0 and 1 defined or enumerated, or shall have a comment which clearly explains what each constant means in the context of its use. Where “0” and “1” have multiple meanings in the code unit, even they should be identified. Example: “0” may be used as FALSE, initializing a counter to zero, or as a special flag in a non-binary category.

v. Only contains the minimum implementation of the “a = b ? c : d” syntax. Expansions such as “j=a?(b?c:d):e;” are prohibited.

w. Has all assert() statements coded such that they are absent from a production compilation. Such coding may be implemented by ifdef()s that remove them from or include them in the compilation. If implemented, the initial program identification in setup should identify that assert() is enable and active as a test version.