Automated Incorporation of

Upset Detection Mechanisms in Distributed Ada Systems

by

Elisa K. Heironimus

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

Dr. J.G. Tront, Committee Chairman

Dr. J.R. Armstrong

Dr. C.E. Nunnally

June 6, 1988

Blacksburg, Virginia

Automated Incorporation of

Upset Detection Mechanisms in Distributed Ada Systems

by

Elisa K. Heironimus

Dr. J.G. Tront, Committee Chairman

Electrical Engineering

(ABSTRACT)

This thesis presents an automated approach to developing software that performs single event upset (SEU) detection in distributed Ada systems. Faults considered are those that fall in the single event upset (SEU) category. SEUs may cause information corruption leading to a change in program flow or causing a program to execute an infinite loop. Two techniques that detect the presence of these upsets are described. The implementation of these techniques is discussed in relation to the structure of Ada software systems and exploit the block structure of Ada.

A program has been written to automatically modify Ada application software systems to contain these upset detection mechanisms. The program, Software Modifier for Upset Detection (SMUD), requires little interactive information from a programmer and relies mainly on SMUD directives that are inserted into the application software prior to the modification process. A full description of this automated procedure is included.

The upset detection mechanisms have been incorporated into a distributed computer system model employing the MIL-STD-1553B communications protocol. Ada is used as the simulation environment to exercise and verify the protocol. The model used

as a testbed for the upset detection mechanisms consists of two parts: the hardware model and the software implementation of the 1553B communications protocol. The hardware environment is described in detail, along with a discussion on the 1553B protocol. The detection techniques have been tested and verified at the high level using computer simulations. A testing methodology is also presented.

Acknowledgements

The author is deeply thankful to her major advisor, Dr. J. G. Tront for the continuous support and encouragement provided throughout the course of research. Thanks also go to Dr. J.R. Armstrong and Dr. Charles E. Nunnally for agreeing to be committee members. This research work was supported in part by the Naval Research Laboratory under Contract FE238311.

Table of Contents

List of Illustrations

List of Tables

Chapter 1

Introduction

This thesis presents an automated approach to developing software that performs single event upset (SEU) detection in distributed Ada systems. The heart of the system is a program that processes application software written in Ada, inserting self-checking constructs into the overall software package. The self-checking constructs are designed to detect single event upsets that cause deviations in program flow. This fault detection technique is particularly targeted for microprocessor-based satellite control systems that have weight and power limitations imposed upon them. For this reason, it is important that the technique not incur extensive, if any, hardware overhead. Faults considered are those that fall in the transient upset category, i.e., faults that cause no permanent damage to the circuit, but rather cause a perturbation of data or control information. Transient upsets may cause information corruption, leading to a change in program flow or in the worst-case, causing a program to execute an infinite loop. Two techniques that detect these undesirable events are de-

scribed in this thesis. The implementation of these techniques is discussed in relation to the structure of Ada software systems.

1.1 Background and Previous Research

Satellite computers are highly susceptible to being struck by cosmic particles. These particles may affect a memory cell, causing the state of the cell to change from a 0 to a 1, or vice versa. This change is referred to as a single event upset (SEU). Rasmussen [22], found that the micro computer components affected by SEUs include the program counter, stack pointers, micro-program sequencer registers, and temporary holding registers. The topic of this thesis is concerned with those SEUs that strike control registers, thus leading to a possible deviation in normal program flow.

Due to the weight and power constraints typical of satellite computer systems, the upset detection method employed should not incur extensive memory overhead. A software technique is advantageous since the availability of "marginal" memory allows a software self-checking structure to be incorporated. The term "marginal" memory refers to the memory space not used by the application program but included in the physical memory implementation (e.g., the application program uses 56K and the physical memory is implemented using a 64K chip.)

Previous research in the area of transient upset detection involved both hardware and software techniques. These techniques detect a range of effects arising from transient upsets. One method of detecting upsets involves the concept of abstraction

verification, which is presented by Schmid et al. [5]. Detection mechanisms which stand at a level between simple watchdog timers and complex redundant configurations were employed. Abstraction verification refers to an approach of using these intermediate level mechanisms for external monitoring of upsets. Seven mechanisms that were utilized to detect upsets affecting various system features are listed below:

1. Invalid program flow - detection of an improper sequence of instructions
2. Invalid opcode address - detection of an instruction fetch from a non-instruction address
3. Unused memory - detection of access to existing unused memory area
4. Invalid read address - detection of data access from an instruction area or unused/non existent memory
5. Invalid opcode - detection of an illegal instruction fetch
6. Invalid write address - detection of an attempt to write into unalterable memory area
7. Non-existent memory - detection of an attempt to access a memory location that is non-existent

These seven techniques provided an overall detection rate of 73%; the mechanism that provided the best performance, in terms of upset exposure, was the invalid program flow mechanism (63%). While the faults injected onto the address and control lines yielded high detection rates, the injected faults on the data lines went largely undetected.

Transient upset detection techniques employed by Li [17] were implemented in an external hardware detector design. The techniques were based on illegal operation detection and loss of program control detection and required fewer components than the microprocessor circuit. This method of detection is advantageous for microprocessor-based systems that have weight and power limitations imposed upon

them. Due to the short propagation delay in the detection circuitry, there is a small fault detection latency. However, a disadvantage of this technique is that it requires correct operation of the external hardware at all times.

Sosnowski [10] has used a data acquisition system model to illustrate implementation of a set of techniques to detect transient upsets. These techniques are a combination of hardware and software methods and are designed to detect several different effects of transient upsets. The software testing techniques make use of available RAM and ROM and consist of a procedure for determining the validity of a return address stored at the top of a program stack before actually returning from a subroutine. Unused memory space is filled with restart instruction codes and constant parameter tables that are stored in ROM are ended with some restart instruction codes. An advantage of the techniques employed is that the required additional hardware was uncomplicated and existing software was utilized.

A software technique for detecting deviation of program flow is discussed by Oak [12]. The technique consists of modifying software to obtain a self-checking capability. SEUs that cause a disruption in program flow may be detected by incorporating a self-checking mechanism in the software. SEUs that cause program execution to resume in a loop structure may potentially result in an infinite loop if certain loop variables are not initialized. In this situation, an SEU would go undetected by a self-checking software mechanism. A hardware watchdog timer provided a simple and inexpensive solution to this problem. The technique is useful for systems that have a weight limitation since the self tests do not incur extensive memory overhead. The technique was implemented in software models written in the Intel 8086 and TI

9900 assembly languages. The advantage of this technique is that a minimal amount of time and memory overhead was incurred.

1.2 Purpose and Significance of Study

The upset detection mechanisms presented in this thesis are based on the self-checking concept and watchdog timer approach described by Oak [12]. However, his method required that a programmer explicitly incorporate the upset detection mechanisms into the assembly language software at the time of development. The purpose of this thesis was to develop a program that would automatically insert the block-checking constructs into distributed Ada software. By automating the procedure, the programmer is relieved of having to understand self-testing techniques and is relieved of manually re-coding a program to incorporate these techniques. Automating the process also reduces the possibility of introducing errors during the re-coding process. The block-checking technique has already been verified at the assembly level for the Intel 8086 and TI 9900. The significance of this thesis is that the detection techniques are automatically incorporated into a high order language and tested at this level. The automatic insertion of self-checking constructs is performed by a program called Software Modifier for Upset Detection (SMUD).

As a study of how the upset detection techniques function, the checking mechanisms have been incorporated into a distributed computer system that employs the MIL-STD-1553B communications bus protocol. The model used as a testbed for determining the effectiveness of the upset detection mechanisms consists of two parts:

software that simulates the hardware environment and the software implementation of the 1553B communications protocol. For testing purposes, the protocol model was modified slightly to allow for injection of SEUs. The model developed provides a good example of how a generalized simulation environment may be created using the concurrent processing features of Ada.

The distributed system model used in testing the detection mechanisms reflects the current trend towards implementing distributed architectures. The use of distributed architectures in embedded computer systems are being employed in avionic and space systems with the intent of increasing performance and reliability of the system software [18]. The modularity of the systems also provides for easier design implementation and maintenance.

1.3 Organization of Thesis

The chapters that follow include discussions of the system model, the upset detection techniques, the software modifying program (SMUD), and the results from testing these techniques.

Chapter 2 provides a brief background on Ada. This is for the purpose of aiding in understanding how Ada has been employed in developing the upset detection techniques and SMUD. The concept of software partitioning is also presented since the manner in which a software system is mapped onto an underlying hardware system affects the implementation of detection techniques.

Chapter 3 discusses the concept of upset detection by insertion of block-checking constructs and a watchdog timer mechanism into application software. The method of controlling a hardware watchdog timer from the software by using the machine level representation features of Ada is described.

Chapter 4 discusses the development of SMUD and how the detection techniques are incorporated into Ada application software. The modification procedure for a portion of the system model is described in detail.

Chapter 5 describes the system model that employs the MIL-STD-1553B communication bus protocol model and the simulation environment in detail. The simulation environment, based on the characteristics of the system hardware, has been created in order to test the system for upset coverage. This environment was modeled using features of Ada that allow encapsulation of related entities (packages) and concurrent processing (tasks).

Chapter 6 presents the simulation and testing results obtained from incorporating the detection techniques into the system model. The upset detection techniques have been verified through simulation of transient upset effects. The method of modeling these transient upsets in the system is described. Time and memory overhead values have been obtained for each technique implementation. Furthermore, performance measures, such as upset detection latency, overhead, etc., obtained from testing are discussed.

Chapter 7 concludes the thesis with a summary of the upset detection techniques and their effectiveness in distributed Ada software and a description of potential further research and development.

Appendix A contains the source code for the protocol model incorporating the upset detection mechanisms. Appendix B contains the source code for the SMUD pre-processing software.

Chapter 2

Ada Review

This chapter provides a background to Ada to aid in understanding how Ada was employed throughout this thesis. Most of the references to Ada in this thesis will be discussed here; however, certain topics such as machine-dependent features, will be reserved for discussion in later chapters. The information presented here is focused on Ada program units, the exception handling capability of Ada, and software partitioning.

Ada is a block-structured, high-order language. Program units are represented by three basic entities: subprograms (procedures and functions), packages, and tasks. These three entities are described in further detail in the following subsections.

2.1 Subprograms

Ada subprograms are similar to those in other high-order languages in that they consist of a specification and a body. For both procedures and functions, the specification consists of the procedure name along with any formal parameters. For functions, the specification will also consist of a return type. The mode of the formal parameters, which may be specified as ***in***, ***out***, or ***in out***, specifies whether the parameter is an input, output, or both to the procedure. Parameter modes for functions may only be specified as ***in***. An output value for a procedure is specified in a return statement. The body of a subprogram contains the sequence of statements that define the algorithm. Shown below is an example of a procedure and a function.

Example 1

```
procedure GET_JOB (time : in integer; job : out string) is
    local declarations
begin
 .
 .  executable statements
 .
end GET_JOB;
```

Example 2

```
function TELL_TIME (clock : in integer) return integer is
 time : integer;

begin
 .
 .
 return time;
end TELL_TIME;
```

2.2 Packages

An Ada package is a program unit that allows logically related entities, such as tasks, subprograms, or variable declarations, to be grouped together. A package may also be used to provide an abstract data type that contains a hidden part and a visible part. This is referred to as information hiding and it protects a programmer by not allowing the programmer access to the information.

A package generally consists of two parts: a specification and a body. The specification declares the interfaces to any program units contained within the package along with any variable, constant, or type declarations. The package body elaborates on the executable portion of each program unit declared in the package specification. If a package specification consists only of variable , type, or constant declarations, no corresponding package body is needed. Shown below is an example of a package that contains both a specification and a body. The specification portion includes the header for a procedure while the package body contains the implementation details of the procedure.

Example 3

```
package DRAW_BOX is
  procedure DRAW_LINES(col,row : integer);
end DRAW_BOX;
package body DRAW_BOX is
  procedure DRAW_LINES(col,row : integer) is
    .
    .
    .
  end DRAW_LINES;
end DRAW_BOX;
```

2.3 Tasks

A task is a program unit that may execute in parallel with other program units. The concurrency may or may not be actual, depending on whether multiple processors are available for execution or whether a multi-programming environment with interleaved execution of tasks is being carried out on a single processor.

To understand what tasks are and how they are used, consider the following examples. The code shown below in Example 4 corresponds to a set of procedures being executed sequentially while the code in Example 5 uses tasks and executes all four chores concurrently.

Example 4

```
procedure GARDENER is
begin
  .
  .
  MOW_LAWN;
  TRIM_TREES;
  TRIM_HEDGES;
  WATER_PLANTS;
  .
  .
end GARDENER;
```

Example 5

```
procedure GARDENER is

task LAWN;
task body LAWN is
begin
  MOW_LAWN;
end LAWN;

task TREES;
task body TREES is
begin
  TRIM_TREES;
end TREES;
```

```
task HEDGES;
task body HEDGES is
begin
  TRIM_HEDGES;
end HEDGES;

task PLANTS;
task body PLANTS is
begin
  WATER_PLANTS;
end PLANTS;

begin  -- procedure GARDENER --
    .
    .
    .
end GARDENER;
```

All tasks that are listed in the declaration section of another program unit (the parent unit) start executing when the unit is called. Since the tasks in Example 5 are in the declaration section of procedure GARDENER, all four task bodies begin execution simultaneously once the procedure is called.

Tasks, like packages, consist of a specification and a body. A task specification defines the interface presented to other program units by using entry statements, similar to subprogram declarations. If no entry statements are used, the task specification states only the name of the task, as in Example 5. Like subprograms, the task entry calls may also have associated parameters. For each entry given in the task specification, there is a corresponding **accept** statement in the task body. For example:

Example 6

```
task BUFFER is
  entry MESSAGE_OUT(data_out : out string);
  entry MESSAGE_IN (data_in  : in  string);
end BUFFER;


task body BUFFER is

   local declarations
```

```
begin
 loop

  select
    accept MESSAGE_OUT(data_out : out string) do
      data_out := data;
    end MESSAGE_OUT;
  or
    accept MESSAGE_IN(data_in : in string) do
      data := data_in;
    end MESSAGE_IN;
  end select;

 end loop;
end BUFFER;
```

The task specification and body shown in Example 6 has two points of entry with which other program units may extract or send data. A rendezvous occurs when one task makes an entry call to another task and the corresponding accept statement of the called task is executed. The **select** statement allows for the option of choosing either entry call. However, if neither branch of the select statement is chosen, the flow of execution is suspended until an entry call has been made. In order to avoid a possible endless wait condition, Ada provides for timed entry calls. Timed entry calls are implemented by two control mechanisms:

- using an **else** clause in the select statement
- using a **delay** statement as one of the branches of the select statement

Example 7

```
select
  accept NOW(.. parameters ..) do
    . sequence of statements associated with
    .      entry call
  end NOW;
else
  -- alternative sequence of statements
end select;
```

Example 8

```
select
  accept NOW(.. parameters ..) do
```

```
    . sequence of statements associated with
    .       entry call
  end NOW;
or
  delay 10.00;
  -- optional sequence of statements
end select;
```

Examples 7 and 8 show examples of a select statement containing an **else** clause and a **delay** statement, respectively. The **else** clause of a select statement is chosen if no other rendezvous occurs immediately. For example, without an **else** clause, if task A calls task B which is currently involved in another rendezvous, the calling task is suspended until its call has been accepted. A select statement that contains an **else** clause allows an optional sequence of statements to be executed if the call is not accepted immediately. Otherwise, execution of task A is suspended until the entry call made to task B has been accepted and completed. The **delay** statement works in the same way as does the **else** clause with the difference being that a specified delay period is allowed to elapse before executing an optional set of statements. For example, using a **delay** statement will cause the statements associated with the delay branch of the select statement to be executed if no rendezvous occurs before the designated time period. If a rendezvous occurs before the delay expires, the expired time value will be reset to the original delay value.

2.4 Exception Handling

Exception handling in Ada permits a programmer to handle exceptional situations which cause normal program execution to be suspended. For each exception, either

user-defined or predefined in Ada, a handler may be used to permit some recovery action such as using a different approach, retrying the operation, or attempting to repair the cause of the error [24]. The predefined exceptions are [2]:

CONSTRAINT_ERROR - raised when a constraint, such as a defined range or index, has been violated

NUMERIC_ERROR - raised when exceptions occur during numeric operations such as overflow, underflow, etc.

SELECT_ERROR - raised when all branches of a select statement (not containing an else clause) are closed

STORAGE_ERROR - raised when dynamic storage allocated to an entity has been exceeded

TASKING_ERROR - raised when exceptions during intertask communications occur

User defined exceptions are declared in the declarative section of a program unit such as the following:

INCORRECT_TAG : exception;

While predefined exceptions may be raised either explicitly or implicitly (as the result of a run-time error), user-defined exceptions must be raised explicitly. This is accomplished through the use of the **riase** statement.

raise INCORRECT_TAG;

This has the effect of suspending execution flow and transferring control to an exception handler.

An exception handler is always placed at the end of a block. An exception handler frame begins with the keyword **exception** and may consist of several exception handlers. Each handler is introduced by a **when** clause followed by a sequence of recovery action statements. The recovery procedure indicated in the exception handler may either re-raise the exception and propagate it to the next enclosing program unit or provide a sequence of actions to be performed. If a handler is not provided for a

particular exception in the current block, the exception is re-raised and propagated through to the enclosing program unit. The format of an exception handler that handles both user-defined and predefined exceptions is given below:

Example 9

```
procedure SHOW is

  INCORRECT_TAG : exception
    .
    .
    .
begin    -- start of program unit
    .
    .
    .
 exception
   when INCORRECT_TAG = >      -- user-defined
          DO_RECOVERY;
   when TASKING_ERROR = >      -- predefined
          DO_NULL;
   when CONSTRAINT_ERROR = >   -- predefined
          DO_OVER;
end;
```

The procedure in example 9 contains exception handlers for one user-defined exception, INCORRECT_TAG and for two predefined exceptions, TASKING_ERROR and CONSTRAINT_ERROR. The handler for the user-defined exception invokes a procedure DO_RECOVERY while the other two handlers specify other procedures to be invoked if the exceptions, TASKING_ERROR and CONSTRAINT_ERROR, are raised.

2.5 *Partitioning Ada Software for Distributed Systems*

Most modern real-time autonomous systems are being designed as distributed computer systems rather than as uni-processor systems. Distributed systems may be divided into three components [16]:

- application-dependent software
- distributed (mainly application-independent) underlying hardware components
- a partitioning specification that maps the application software onto the hardware

Partitioning specification [16] has been the topic of much research recently since this aspect of distributed system design will ultimately affect other system resources such as real-time response, communication between processors, and memory. Software partitioning identifies which processor each software functional unit will execute on. The method used in partitioning Ada software for execution on a distributed system affects the design of the software system, thus affecting the software methods employed for upset detection. Four distinct methods for partitioning Ada software systems have been discussed by Cornhill [16]:

1. writing an Ada program for each processor
2. writing a single system program and partitioning any functional units
3. writing a single system program and partitioning the task units
4. extending Ada to include additional constructs

The first option of writing a separate program for each processor does not allow cross checking of variables between program interfaces. In addition, software portability is decreased, thus requiring software redesign in the event of any hardware change.

Writing one program for the entire application program allows data types to be represented consistently and for correct parameter types used in inter-process communication to be checked. The second option listed involves partitioning any source level Ada construct such as tasks, functions, and procedures. This approach allows several different partitioning methods without needing to redesign. The third option of partitioning only tasks allows a correlation of task concurrency to processor concurrency. A disadvantage of this method is that all code must be partitioned early in the design stage of software development. The fourth approach to partitioning systems adds new grammatical constructions to Ada specifically designed for distributed programming. This implies an inadequacy with Ada i.e., not a strong enough exception handling capability.

The partitioning approach selected for this thesis in developing the distributed software system was to develop a single program and partition the task units. The entities resulting from the partitioning are referred to as ***virtual nodes***, where a ***virtual node*** is defined as a group of program units executing on the same processor. Writing a single program allowed cross-checking of variables and data types across separately compiled units. Inter-task communication was limited to the rendezvous mechanism.

The information presented in this chapter has served to provide a brief background to the references made of the Ada language (i.e., program units, exception handling, software partitioning). Discussion of some features of Ada has been reserved for later chapters where a discussion on their implementation and use is presented. For a more complete introduction to Ada, see [1,2,23,24].

Chapter 3

Incorporation of Upset Detection Techniques

During execution of real-time software, a processor continuously executes a defined software block. A transient fault in a control register of a processor may cause the program flow to deviate from the normal path and resume execution in another block of software (Figure 1). An effective strategy for detecting this type of deviation has been described by Oak [12]. The differences in the technique proposed here, and that of Oak are:

1. Oak's technique operated on software implemented in assembly language while the technique developed here operates on a high order language.
2. In this model, a generalized distributed system is employed as opposed to a uniprocessor system used by Oak.
3. Oak's technique is based on sequential processes while Ada systems may have several concurrent processes executing on a single processor, or multiple processors.
4. A program has been written to automatically insert block-checking constructs into Ada application software.

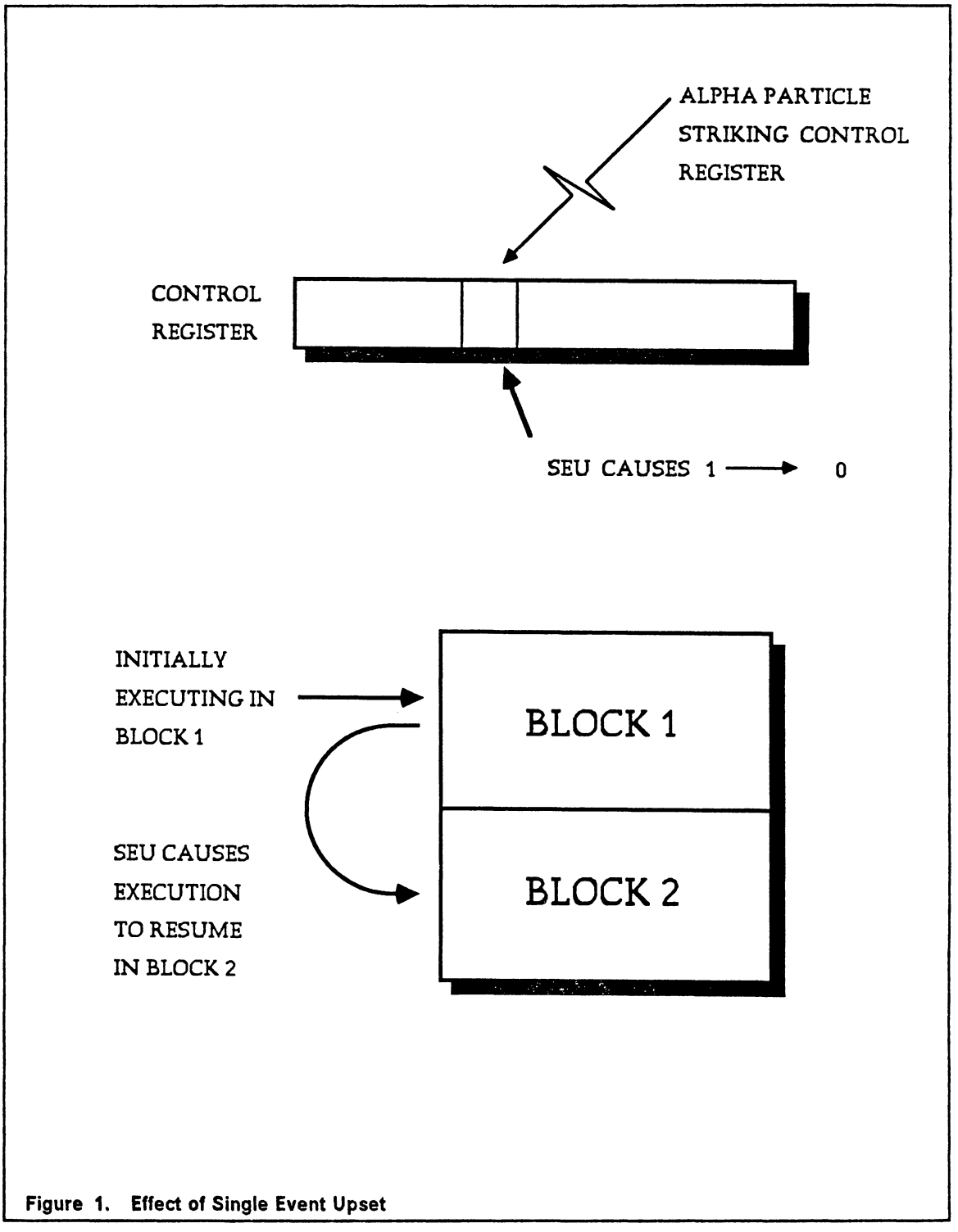

Figure 1. Effect of Single Event Upset

To detect a deviation in program flow, block-checking constructs are inserted into the application software. This requires the application software to be defined in terms of blocks, with a block constituting a logically bound set of instructions around which the block-checking modifications are to be inserted. The constructs consist of TAG initializing, TAG checking, and TAG resetting. The placement of the block-checking constructs is shown in Figure 2.

To insure detection of deviated program flow, a hardware watchdog timer mechanism is also employed. The timer is set upon correct entry into a block via a software instruction. A watchdog timer that is not reset before its terminal count is reached will signal an error. This additional detection mechanism will detect infinite loops and reduce error detection latency. Figure 3 illustrates the flow diagrams for both an unmodified software block, and one that has been modified by the incorporation of the block-checking software.

3.1 Incorporation of the Block-Checking Structure

The approach to inserting upset detection mechanisms in pre-existing Ada software requires block boundaries to be defined. Definition of these boundaries is not too difficult since Ada is a block structured language. The entities that are defined to be blocks are procedures, functions, and tasks. Each block is assigned a unique identifying value. Upon correct entry into a block, the ID of the block is assigned to a TAG variable. The TAG retains this ID value until the end of the block is reached. The TAG is then checked for the block ID value and, if correct, is reset to the value zero. If,

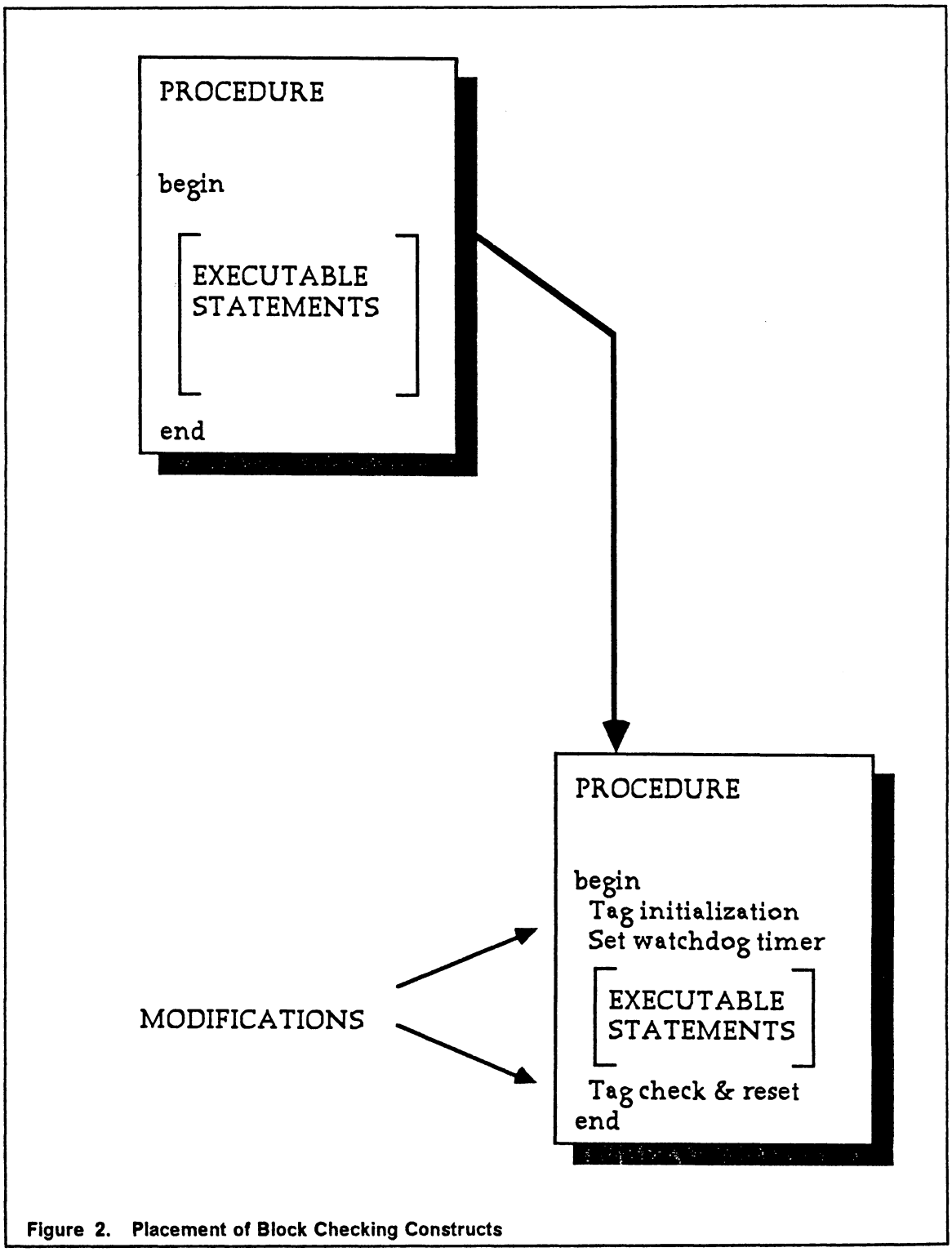

Figure 2. Placement of Block Checking Constructs

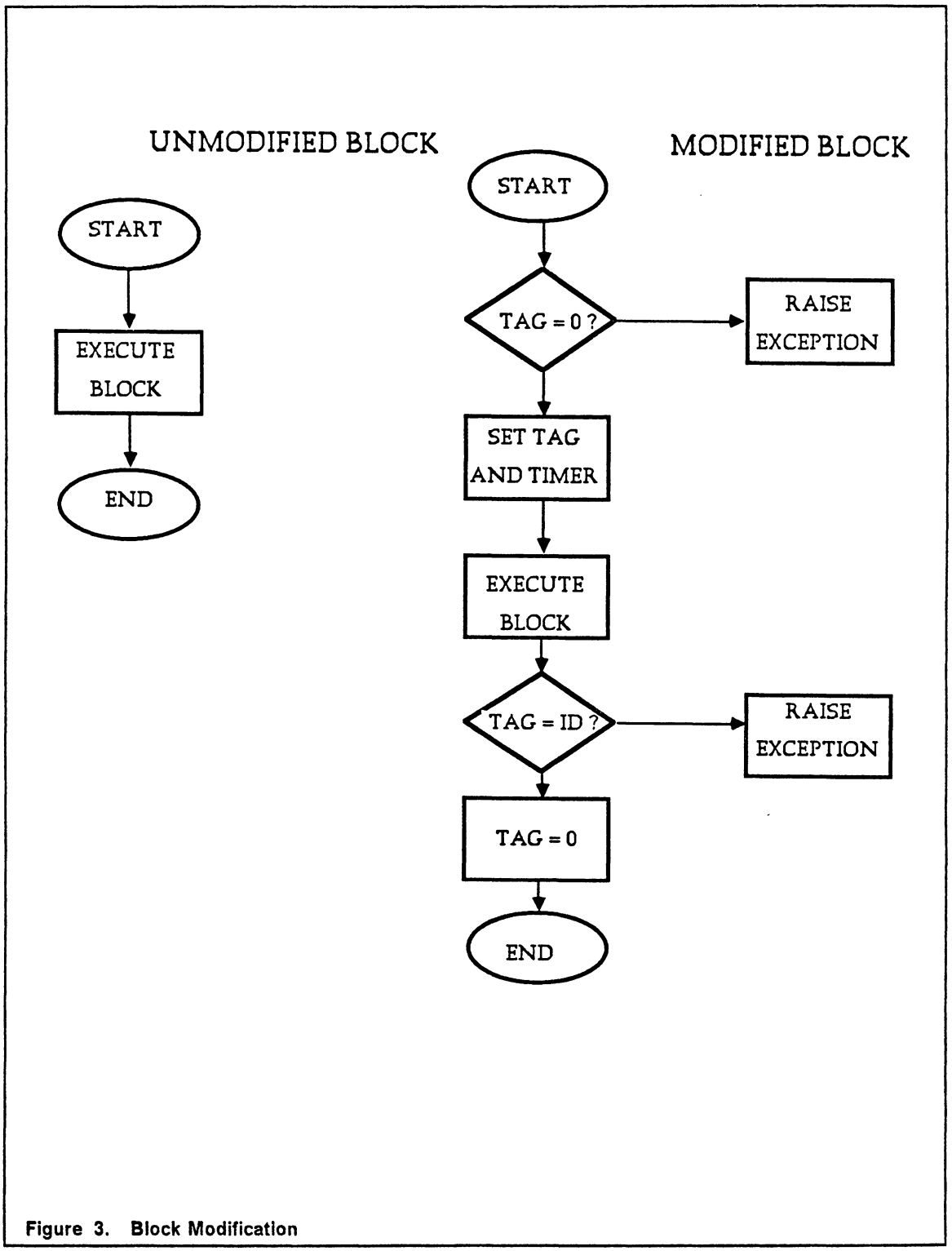

Figure 3. Block Modification

at any time, an error is encountered in the value of a TAG, a procedure is called to raise an exception. This suspends further execution of the currently executing block and allows a recovery mechanism to take place in the exception handler.

The insertion of the block-checking constructs into software is divided into two parts. The first part is the initializing of the TAG; the second part is the resetting of the TAG. Additional checks are inserted if the body of the block is lengthy and/or loop structures are present.

The TAG variable is declared as an integer and initialized to zero in the declaration section of each program unit. Initializing the variable in the declaration section will insure that the TAG value is zero when the program unit is properly invoked. Shown below is the format of a variable declaration and initialization. The name TAG1 denotes the name of the variable while the reserved word **integer** indicates the variable type. To initialize the variable, the assignment operator := is provided along with an initialization value.

```
TAG1 : integer := 0;
```

The value of the TAG variable, which is unique to a block, identifies the currently executing block of the system. The block-checking construct that checks for a correct block entry and initializes the TAG variable to the ID value of the block is given below:

```
if TAG1 = 0 then
  TAG1 := block_id;
else
  HANDLER;
end if;
```

The second part of inserting a block-checking construct takes place at the end of the block. The code below shows the resetting of the TAG before exiting the block:

```
if TAG1 /= block_id then
```

```
    HANDLER;
else
    TAG1 := 0;
end if;
```

This construct is inserted at the end of a program unit, either before an exception handler or before the last end statement. The TAG is checked for the value that uniquely identifies that block. If incorrect, the procedure HANDLER is called and an exception is raised; if correct, the TAG is reset to zero. A TAG is defined to be invalid if it does not contain the value it is checked for. An invalid TAG indicates that the block has incorrectly been entered into i.e., other than through the correct entry point, which is the beginning of a block.

The internal structure of a block may contain code that is either executed once or many times, as in a loop structure. For long code blocks that do not contain loop structures, additional block checks are inserted after a certain number of statements to minimize the latency in detecting faults following an incorrect block entry. For example, consider a program unit in which the TAG value is checked only at the entry and exit points. If an SEU causes a program flow deviation into this block at a point beyond the TAG initialization at the start, upset detection will not occur until the TAG check at the end of the block. Depending on the block length and point of entry, a long latency period may occur. The insertion of a block-checking construct at some point within that block would minimize the detection latency period. This construct does not check for a zero value or reset the TAG to any value. It merely checks for the value of the currently executing block:

```
If TAG1 /= block_id then
    HANDLER;
end if;
```

Code blocks that contain loop structures present another possibility of a long detection latency period. In Ada, loop structures occur in three forms: basic loops, while loops, and for loops. The ***basic loop*** allows a sequence of statements to be executed until a certain condition is met. The statements of a ***while loop*** are executed while a certain condition remains TRUE, and a ***for loop*** is executed once for each value in the discrete range of the control variable. Both of the latter loop structures may contain an exit statement to allow a premature exit.

Block checks may be included within the loop structure to minimize upset detection latency. If an SEU causes execution to resume within a loop structure not containing any TAG checks, a long detection latency period may follow, depending upon the length of the loop structure and the number of iterations to be performed. Inserting a block-checking construct within the loop structure will prevent the statements from being executed more than once before the program flow deviation is detected. The additional memory and execution time overhead will be of greater proportion for loops that contain a few lines than for loops that contain a much larger number of statements. Since there is no way of always pre-determining the value of a loop control variable, a TAG check should always be incorporated into a loop structure when possible.

3.1.1 Nested Program Units

The block-checking approach for the case of nested program units may differ from that of top level units where the calling hierarchy is precisely known. If the hierarchy is known, then upon entering a nested routine, the TAG can be checked for the value

of the calling program unit, rather than checking for a value of zero. If the value is correct, the TAG is set to a new value that uniquely identifies the nested block. Before exiting the nested block, the TAG is checked for the value corresponding to this block and is reset to the value of the calling program unit. The other principles of the technique used for nested statement structures, such as introducing block checks within loop structures and adding extra TAG checks in long blocks of code, remain the same for nested routines.

This approach is adequate if each nested program unit is to be called by only one program unit; however, it is very unlikely that this will be the case for all units, thus making the previously described TAG manipulation unsuitable. To remedy this problem, a dedicated TAG variable is used for every procedure, function, and task, no matter what its location is in the nesting levels. Although this increases time and memory overhead, this approach allows the same self-testing technique to be used for any level of nested program units. Using this method eliminates restrictions on which unit may call another unit.

3.2 *Watchdog Timer Mechanism*

Although the insertion of block-checking constructs into application software is an effective means of detecting deviations from normal program flow, some deviations will go undetected by this method. These deviations cause program flow to resume in the same block while skipping statements. If program flow resumes within a loop structure, certain loop variables may fail to be initialized, possibly resulting in an in-

finite loop. A hardware watchdog timer is a simple and inexpensive means of detecting these infinite loops and reducing upset detection latency. The timers are loaded with values at pre-established checkpoints in the software. The timer counts down and is reset before the next checkpoint is reached. If the timer is not reset before the count expires, the timer signals the processor of the error. As an example of how a simple loop structure may be turned into an infinite loop, consider the following section of code:

```
start := 1;
loop
  if TAG /= TAG_id then
    HANDLER;
  end if;
  exit when start = 100;
  start := start + 1;
end loop;
```

Although the above loop structure contains a TAG check, a deviation in program flow may not be detected if the execution resumes in the middle of the loop or after initialization of the variable **start**. The failure of the variable **start** to be initialized will cause this value to be undefined, and the loop may be executed indefinitely.

3.2.1 Watchdog Timer Initialization Strategies

Two strategies for initializing a watchdog timer are discussed. These initialization strategies are referred to as Version I and Version II. The first initialization strategy (Version I) used in this research consists of resetting a simple counter. There are two places where software instructions for resetting the watchdog timer are inserted into the application software: upon correct entry into a block and after a procedure call. The instruction to reset the watchdog timer is inserted into the code after the block-checking construct for the initialization of the TAG check has been inserted. The in-

struction that resets the hardware timer explicitly refers to the low level features associated with the watchdog timer. This is explained in further detail in the next section.

The second location for resetting the watchdog timer is after a procedure call. Since each program unit is uniformly modified, each unit will include an instruction at the start of a block to reset a timer. A unit that has been called by another program unit will destroy the current value of the watchdog timer. Upon termination of the called unit, the count remaining in the timer may not be sufficient to allow completion of the calling program unit. Unless the timer is reset after the subprogram call, the timer may improperly signal an error. Therefore, by default, an instruction to reset a watchdog timer is inserted after each procedure call and task rendezvous to reset the timer.

The second initialization strategy (Version II) is based on one described by Oak [12]. This method uses the following initialization methods:

1. Initialization of the watchdog timer at the start of a program unit
2. Adding an execution time count to the existing watchdog timer for nested program units.

This strategy does not require an instruction to reset the watchdog timer after each nested program unit call. This method is easier to implement in software since each program unit only requires one instruction at the start of the unit to add a count to the existing timer value. However, this method suffers from the need to have a prior information on the length of execution time of a program unit.

In both methods, the watchdog timer is initialized at the start of the program unit. Upon entering a program unit, a value corresponding to the length of time needed to

execute the unit is moved directly to a counter. A problem of implementing a watchdog timer in a task program units arises when an inter-task communication has no defined limit. Since a task body may have its execution suspended until a rendezvous occurs, the time to execute the task unit may not be consistent. In cases such as this, it is likely that a watchdog timer would not be reset in time and would incorrectly signal that an error has occurred. In automating the procedure of inserting instructions to reset watchdog timers, directives are provided to allow a programmer to tailor a software system by indicating whether a watchdog timer operation should be included or omitted for each program unit.

3.2.2 Implementation of Watchdog Timer

The representation and implementation of the hardware watchdog timer has been designed at the high level in Ada. Designing at the high level without considering the actual hardware representation allows the high level representation to be applied to different machines. There are four representation clauses available in Ada that a programmer may use in specifying data representation at a low level:

1. length clause
2. address clause
3. record representation
4. enumeration clause

The Ada declaration of the watchdog timer implemented in this thesis uses the length and address clauses. These two clauses allow the watchdog timer to be referred to explicitly and are discussed further.

The length specification controls the amount of storage to be used for the object. For example, consider the following declaration of an 8-bit watchdog timer:

```
bits : constant := 1;
type watchdog_timer is range 0..255;
for watchdog_timer'SIZE use 8*bits;
```

The range given indicates the possible values (0 - 255) that may be assigned to the watchdog timer. The length clause **for watchdog_timer'SIZE use 8*bits** specifies that the representation of range values can only be stored in 8 bits. For a 16-bit watchdog timer, the values may range from 0 to 65536 and can be represented in 16 bits. The SIZE attribute specifies an upper limit to the number of bits to use when representing objects of the type **watchdog timer.** For example, if **WDT1** and **WDT2** are variables of the type **watchdog_timer,** the compiler is instructed to represent these variable values in 8 bits.

The address clause is used to specify the absolute memory address of the watchdog timer. By specifying an address clause, an object is aligned with a specific machine address. For example, consider the two objects, **WDT1** and **WDT2,** declared as type **watchdog_timer**:

```
WDT1 : watchdog_timer;
for WDT1 use at 16#2000#;

WDT2 : watchdog_timer;
for WDT2 use at 16#2004#;
```

The address clause **for WDT1 use at 16#2000#** indicates that the object **WDT1** is mapped to the hexadecimal memory address 2000. The object **WDT2** is mapped to the physical address 2004. Figure 4 on page 34 shows how a software instruction sets a hardware watchdog timer. To physically access the hardware watchdog timer, the statement **WDT1 := 255** might be executed. This causes the binary value

11111111 to be loaded into the watchdog timer. The value 255 is selected as the default value to which the timer is reset in this research.

This chapter has provided a description of the upset detection mechanisms proposed in this thesis for detection of SEUs in program counters. A hardware watchdog timer mechanism has been incorporated to reduce the error detection latency and detect SEUs not detected by the block checking structure. Appendix B lists the source for SMUD incorporating the Version I watchdog timer strategy.

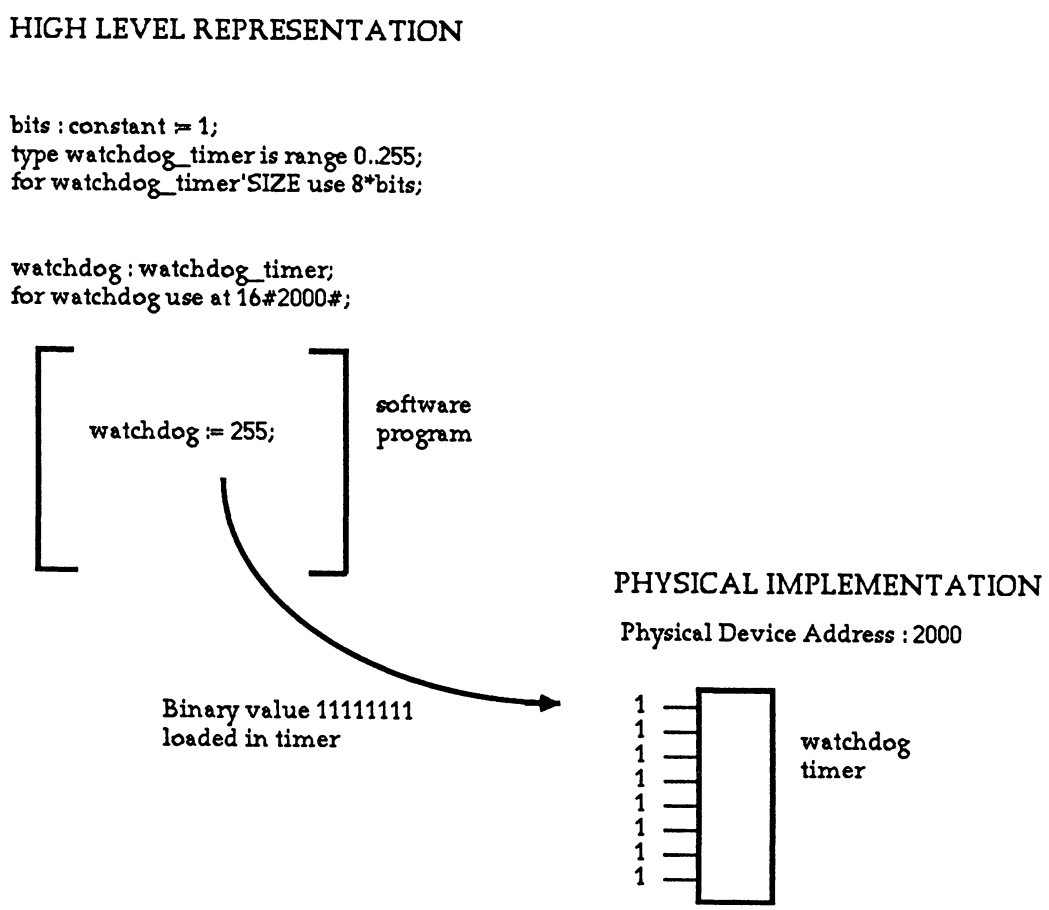

Figure 4. Implementation of Watchdog Timer

Chapter 4

Software Modifier for Upset Detection (SMUD)

Implementing a block-checking structure and watchdog timer mechanism requires that a programmer explicitly incorporate the upset detection mechanisms into the software at the time of software development. An automated approach to incorporate these upset detection mechanisms into software would free the programmer from the task of having to do this manually. In addition, the automated approach requires that a programmer have only a superficial knowledge of self-checking techniques. Automation also provides for uniformity of code. While incorporating a self-checking structure manually allows each software system to be tailored and perhaps optimized individually, large software systems would benefit from the development time savings produced by an automated modifying procedure.

A program to pre-process Ada application software has been developed to automatically insert block-checking constructs and a watchdog timer mechanism into applications software. The program, Software Modifier for Upset Detection (SMUD), has

been developed using Ada and is designed to modify Ada application software. Directives are available that instruct SMUD as to the placement of these detection mechanisms. These directives allow a programmer to tailor a program unit to omit or insert additional detection statements into the code. SMUD can be modified to handle other high level languages that are block structured, such as Pascal or C. The required modifications would mainly involve SMUD routines that depend upon keywords to indicate defined block boundaries.

4.1 Modification Procedure

The method in which SMUD processes the application software and inserts upset detection mechanisms is illustrated in Fig. 5 on page 36. As shown in the figure, the application software is input to SMUD. To keep the original software intact, the modified version is output to a different file. The modified software system must then be re-compiled and linked to update the executable version of the system. Since the actual program unit names remain unaltered, all previous executable files for the unmodified system are replaced by the executable versions for the modified system after the re-compilation.

The method of inserting block-checking constructs into existing software relies on the use of certain keywords defined in Ada. Since the executable body of each program unit is bounded by the keywords ***begin*** and ***end***, a block boundary is easily recognized. When either of these keywords is encountered, a TAG initialization construct (after the keyword ***begin***) and TAG reset construct (before the keyword ***end***) is in-

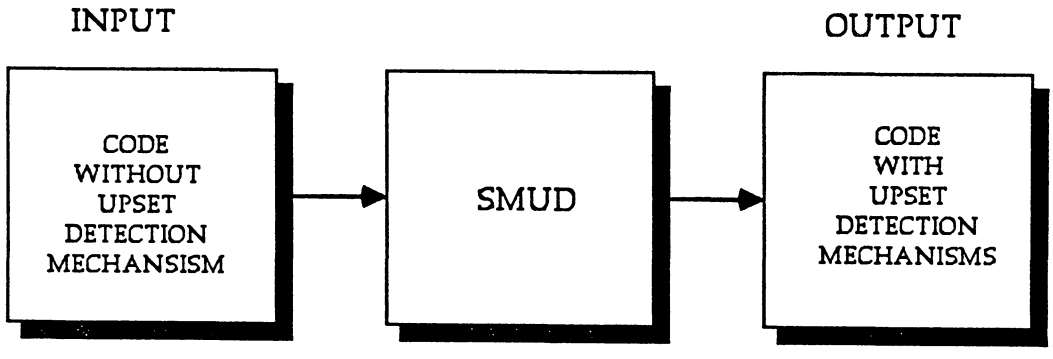

Figure 5. Block Diagram of SMUD

serted into the source file. The instruction for initialization of a watchdog timer is placed after the initialization of the TAG. Details of the operation of SMUD is discussed in further detail below.

The modification procedure requires that the application software be partitioned into ***virtual nodes***. As discussed in Chapter 2, a ***virtual node*** is defined as a set of program units that execute on the same processor. Assuming that the application software to be modified has been partitioned into virtual nodes by the programmer, SMUD first prompts for the number of nodes contained within the entire system. SMUD processes each program unit contained within a node before processing another node. The modification process consists of two passes (PASS_ONE and PASS_TWO) through each node. This is illustrated in Figure 6 on page 39. A brief description of the modification process for a virtual node is described in the following sections. Listed below is the interactive information required from the programmer for each node:

1. Names of program units residing within the node
2. Physical memory mapped address of watchdog timers
3. The base in which the timer address is represented
   (i.e., binary, hexadecimal, etc.)

4.1.1 PASS_ONE

Procedure PASS_ONE is called to gather information about each program unit. Each program unit is assigned a record structure that contains the following information:

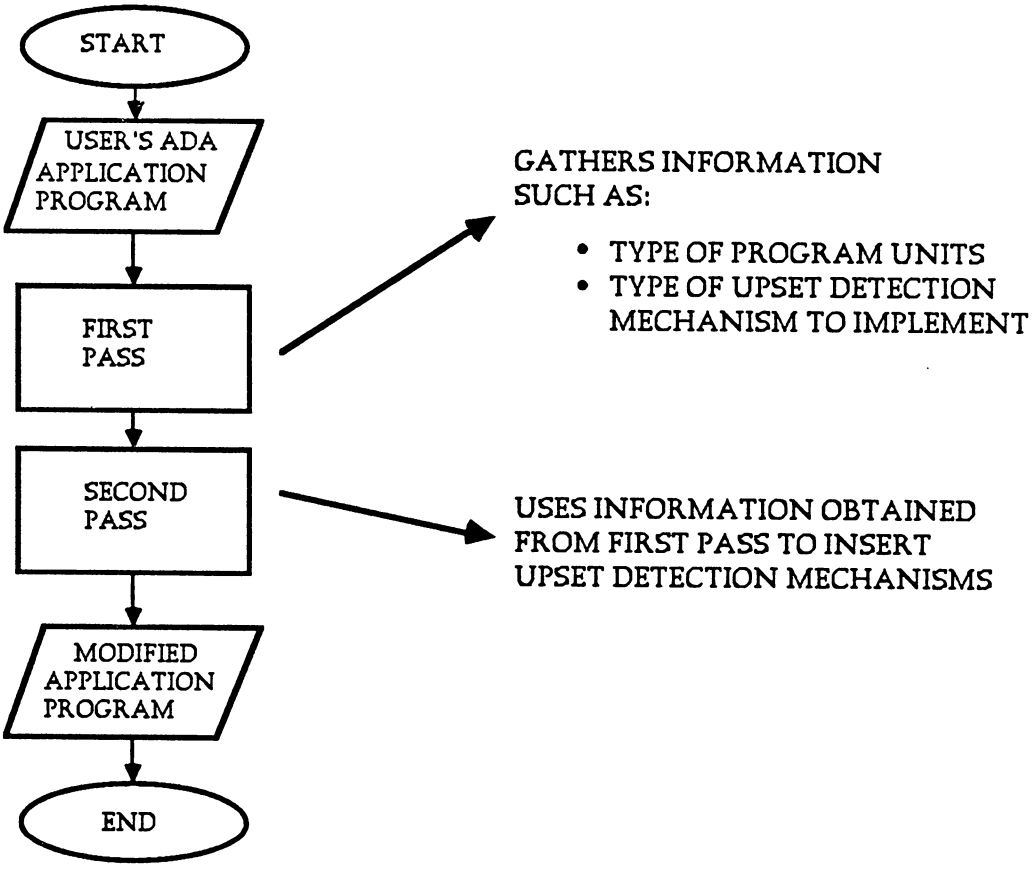

Figure 6. Modification Process

```
type check is (place, show,omit);

type program_unit is
  record
       kind : string (1..9);                    --program unit type
       name : string (1..max_length);           --name of unit
       len : integer;                           --string length of name
       id : integer := 0;                       --block ID
       num_lines : integer := 0;                --no. executable statements
       watchdog : integer := 0;                 --watchdog timer ID
       block : check;                           --type indicates option
       WDT : check;                             --type indicates option
       TAG_name : string(1..4);                 --TAG variable name
  end record;
```

The recorded information identifies the type of program unit (procedure, function, task), the unit name, the identification number, and the modification method. There are three types of defined modifications: ***place***, ***show***, and ***omit***. The modification method is selected via special instructions, known as directives, that instruct SMUD on precisely what modifications are to be done. The use of these directives is discussed later.

4.1.2 PASS_TWO

After PASS_ONE has processed each program unit and recorded the pertinent information, PASS_TWO is then called to reprocess each program unit. Procedure PASS_TWO implements the upset detection mechanisms according to the information obtained in procedure PASS_ONE. The information stored in the record structure of each program unit is examined before insertion of any block-checking constructs or timer activation instructions. PASS_ONE merely gathers information about the program units whereas PASS_TWO actually modifies them.

As mentioned in Chapter 3, detection of program flow deviation is signalled by calling the procedure HANDLER which, in turn, raises the user defined exception INCORRECT_TAG. This procedure is contained within a package (Upset Detection) that is made accessible to each program unit via the ***with*** clause. In order to provide an automated method for modifying the application software, certain requirements need to be met. These limitations imposed on the application software of the distributed system are:

- tasks residing on the same node communicate via the rendezvous mechanism
- no sharing of data or code via library packages or nesting structures
- each program unit must have access to the package UPSET_DETECTION

Two additional files result from the processing of each node: ***Data.Ada*** and ***Upset_Detection.Ada***. These files are created after each program unit has been processed by PASS_ONE. The first file, ***Data.Ada***, only provides documentation of the modifications made to the application software. The data includes the block ID of each program unit and the directives that were encountered in the parsing of each unit. Shown below is a sample ***Data.Ada*** file resulting from a node requiring three watchdog timers. The number of watchdog timers required for a node is dependent on the number of task units residing on the same node. The number of watchdog timers necessary for each node is defined by the number of program units that may run concurrently on a single processor. In addition to the main program, there may be tasks defined within the node. Since these tasks may run asynchronously, each task requires a watchdog timer different from those used by other tasks. If separate timers are not provided for each task, the timer being accessed simultaneously by the tasks will be reset at improper times. Each procedure and function contained within the scope of a task unit or the main procedure accesses the same watchdog timer

as its parent task unit. This configuration does not present a problem of conflicting settings of watchdog timers since each parent task unit may be viewed as a process running sequentially executed program units.

number of watchdog timers needed for node: 3

UNIT ID	BLOCK	WDT	TIMER	PROGRAM UNIT	NAME
1	place	place	1	procedure	t1
2	place	place	2	task	test1
3	place	place	1	procedure	t2
4	omit	place	3	task	test2
5	place	omit	1	procedure	t3

As shown in the sample above, procedure t1 is assigned the unique identifying value of 1 and incorporates both block-checking constructs and a watchdog timer mechanism. The numbers in the column under **TIMER** correspond to which watchdog timer each program unit may access. As discussed earlier, each task unit requires a separate watchdog timer from other task units and the main program unit. In this example, procedure t1 is the main procedure and accesses watchdog timer #1. All functions and procedures that are nested in procedure t1 (procedure t2 and procedure t3) may also access watchdog timer #1 since each of these units are executed sequentially. However, since task units may execute asynchronously, a different watchdog timer is assigned to each task (task test1 and task test2).

The second file created by SMUD, ***Upset_Detection.Ada***, is a package unit that must be made accessible to the application software by using the ***with*** clause. The declaration for the watchdog timers and the procedure HANDLER is contained within this package. The exception handler for the user-defined exception, INCORRECT_TAG, is

also contained within the package body UPSET_DETECTION. It is by means of this exception handler that a upset error recovery sequence may be initiated. This recovery mechanism may consist of a backward recovery scheme that allows some portion of the software to be re-executed. For purposes of testing the upset detection mechanisms in the simulation, the handler displays an error message each time a deviation in program flow is detected. A sample ***Upset-Detection.Ada*** file is given below for a node that employs a watchdog timer at the hexadecimal address **1234.**

```
with system; use system;
package UPSET_DETECTION is

  bit : constant := 1;
  type watchdog_timer is range 0..255;
  for watchdog_timer'SIZE use 8*bit;

  WDT1 : watchdog_timer;
  for WDT1 use at 16#1234#;

  procedure HANDLER;

end UPSET_DETECTION;


with text_io; use text_io;
package body UPSET_DETECTION is

  procedure HANDLER is

    INCORRECT_TAG : exception;

  begin
    raise INCORRECT_TAG;

  exception
    when INCORRECT_TAG => PUT_LINE("incorrect block entry");
  end HANDLER;


end UPSET_DETECTION;
```

4.2 Directives

A directive is defined as an instruction to SMUD concerning placement of the two upset detection mechanisms. The purpose of providing directives is to allow a programmer to dictate what modifications should occur in each program unit. These directives are placed in the application software by the programmer prior to executing SMUD. When any of these directives are encountered within defined block boundaries during procedure PASS_ONE, the information is stored in the record file of the program unit. In the absence of any directives, the program will insert block-checking constructs and a watchdog timer mechanism in the unit according to the techniques outlined in the previous sections. A list of the available directives and their functions is given below:

omit_blk omit insertion of block-checking constructs in program unit

omit_wdt omit insertion of watchdog timer instructions in program unit

show_blk instructs SMUD to indicate locations where block-checking constructs would appear. The locations are shown by commented statements that are inserted into the application software.

show_wdt instructs SMUD to indicate the default locations for placing the reset instruction of the watchdog timer.

blk instructs SMUD to insert a TAG check into the application software wherever this directive appears

wdt instructs SMUD to insert the instruction to reset a watchdog timer wherever this directive appears

omit_tag instructs SMUD to forego inserting a TAG check into a loop structure. This directive must be placed on the same line as the word ***loop***

4.3 Examples of Code with Upset Detection Mechanisms

This section contains examples of the incorporation of the block checking method discussed in Section 3.2. These examples serve to illustrate the uses of the available directives. Example 1a shows a procedure that does not employ any directives. As mentioned earlier, the absence of any directives causes the block checking constructs and watchdog timer mechanism to be inserted automatically. Example 1b shows the modified procedure. Example 2a shows a procedure that contains a nested task unit. The ***omit_blk*** and ***show_wdt*** directives are used here. As shown in Example 2b, the modified main procedure contains a commented statement indicating where an instruction would be placed to set a watchdog timer. The nested task does not contain any block-checking constructs due to the ***omit_blk*** directive. Example 3 illustrates the use of the ***show_blk***, ***omit_wdt***, and ***omit_tag*** directives. The ***show_blk*** directive is used in the nested task body test (3a) and results in commented statements, rather than block-checking constructs, being inserted into the task body (3b). Use of the ***omit_wdt*** directive in procedure t3 prevents any watchdog timer mechanism from being inserted into the code here. The ***omit_tag*** directive is used to show how block-checking constructs can be omitted from loop structures. The use of the ***blk*** directive is shown in Example 4. Placing the ***blk*** directive in procedure t4 (4a) results in the insertion of a block-checking construct where the directive appeared.

As indicated by these examples, each program unit can be individually tailored to utilize block-checking constructs and a watchdog timer mechanism. Using the directives ***omit_blk*** and ***omit_wdt*** will cause the pre-processor to ignore other direc-

tives to insert tag checks or watchdog timer instructions. However, when using the directives ***show_blk*** or ***show_wdt***, the directives ***blk*** and ***wdt*** may also be used. The tag check or watchdog timer instruction will be preceded by comment indicators.

Example 1a

```
procedure t1 is

 add : integer := 0;

begin

   add := add + 12;
   add := 2;
   begin
    loop
      add := add + 1;
      exit when add = 10;
    end loop;
   end;
   add := add + 125;
end;
```

Example 1b

```
procedure t1 is

 TAG1 : integer := 0;
 add : integer := 0;

begin

  If TAG1 = 0 then
    TAG1 := 1;
  else
     HANDLER;
  End if;

  WDT := 255;

  add := add + 12;
  add := 2;
  begin
   loop
     If TAG1 /= 1 then
       HANDLER;
     End if;
     add := add + 1;
     exit when add = 10;
   end loop;
  end;
  add := add + 125;

  If TAG1 /= 1 then
    HANDLER;
  else
    TAG1 := 0;
  End if;

end;
```

Example 2a

```
procedure t2 is

add : integer := 0;
task  test;

task body  test is

temp : integer := 0;
count : integer;
begin
 --$$omit_blk
 for count in 1..10 loop
   temp := temp + 1;
 end loop;
end;


begin

 --$$show_wdt
 add := add + 12;
 add := add + 12;
 begin
  loop
    add := add + 1;
    exit when add = 10;
  end loop;
 end;
 add := add + 125;
end;
```

Example 2b

```
procedure t2 is

TAG1 : integer := 0;
add : integer := 0;
task  test;

task body  test is

temp : integer := 0;
count : integer;
begin
  WDT := 255;
  for count in 1..10 loop
    temp := temp + 1;
  end loop;
end;


begin

 If TAG1 = 0 then
   TAG1 := 1;
 else
   HANDLER;
End if;

--watchdog timer set here

 add := add + 12;
 add := add + 12;
 begin
  loop
    If TAG1 /= 1 then
      HANDLER;
    End if;
    add := add + 1;
    exit when add = 10;
  end loop;
 end;
 add := add + 125;

 If TAG1 /= 1 then
  HANDLER;
 else
  TAG1 := 0;
 End if;

end;
```

Example 3a

```
procedure t3 is

add : integer := 0;
task  test;

task body  test is

temp : integer := 0;
count : integer;
begin
 --$$show_blk
 for count in 1..10 loop
    temp := temp + 1;
 end loop;
end;

begin
 --$$omit_wdt
 add := add + 12;
 add := add + 12;
 add := 2;
 begin
  --$$wdt123
  loop  --$$omit_tag
     add := add + 1;
     exit when add = 10;
  end loop;
 end;
 add := add + 125;
end;
```

Example 3b

```
procedure t3 is

TAG1 : integer := 0;
add : integer := 0;
task  test;

task body  test is

temp : integer := 0;
count : integer;
begin
 -- block check here

 WDT := 255;
 for count in 1..10 loop
    -- tag check here
    temp := temp + 1;
 end loop;
 -- tag reset here
end;

begin

TAG1 = 0 then
  TAG1 := 1;
else
  HANDLER;
End if;

 add := add + 12;
 add := add + 12;
 add := 2;
 begin
  loop  --$$omit_tag
     add := add + 1;
     exit when add = 10;
  end loop;
 end;
 add := add + 125;

 If TAG1 /= 1 then
   HANDLER;
 else
   TAG1 := 0;
 End if;

end;
```

Example 4a

```
procedure t4 is

add1 : integer := 0;
add2 : integer := 0

begin
  add1 := add1 + 12;
  add2 := add1 + 12;
  begin
    --$$blk
      add1 := add1 + 1;
  end;
  add2 := add1 + 125;
end;
```

Example 4b

```
procedure t4 is

TAG1 : integer := 0;
add1 : integer := 0;
add2 : integer := 0;

begin

  If TAG1 = 0 then
    TAG1 := 1;
  else
    HANDLER;
  End if;

  WDT := 255;
  add1 := add1 + 12;
  add2 := add1 + 12;
  begin

      If TAG1 /= 1 then
       HANDLER;
      End if;
      add1 := add1 + 1;
  end;
  add2 := add1 + 125;

  If TAG1 /= 1 then
    HANDLER;
  else
   TAG1 := 0;
  End if;

end;
```

Chapter 5

System Model

In order to test the effectiveness of the software modifier for upset detection (SMUD), a distributed computer system (Figure 7) has been modeled. As mentioned in Chapter 1, the system implements the MIL-STD-1553B communications protocol, which in this example defines the modes of operation between satellite subsystems. The modeled system was developed in Ada with DEC's VAX 11/785 acting as the host. The model developed provides a good example of how a simulation environment may be created using the concurrent processing features of Ada. This chapter will provide a detailed description of the modeled system. The example system is divided into two pieces: the model of the 1553B hardware and the Ada software the implements the 1553B communications protocol which will be run on the 1553B hardware. The portion of the modeled system that is modified by SMUD to include the upset detection mechanisms is the software that implements the 1553B communications bus protocol.

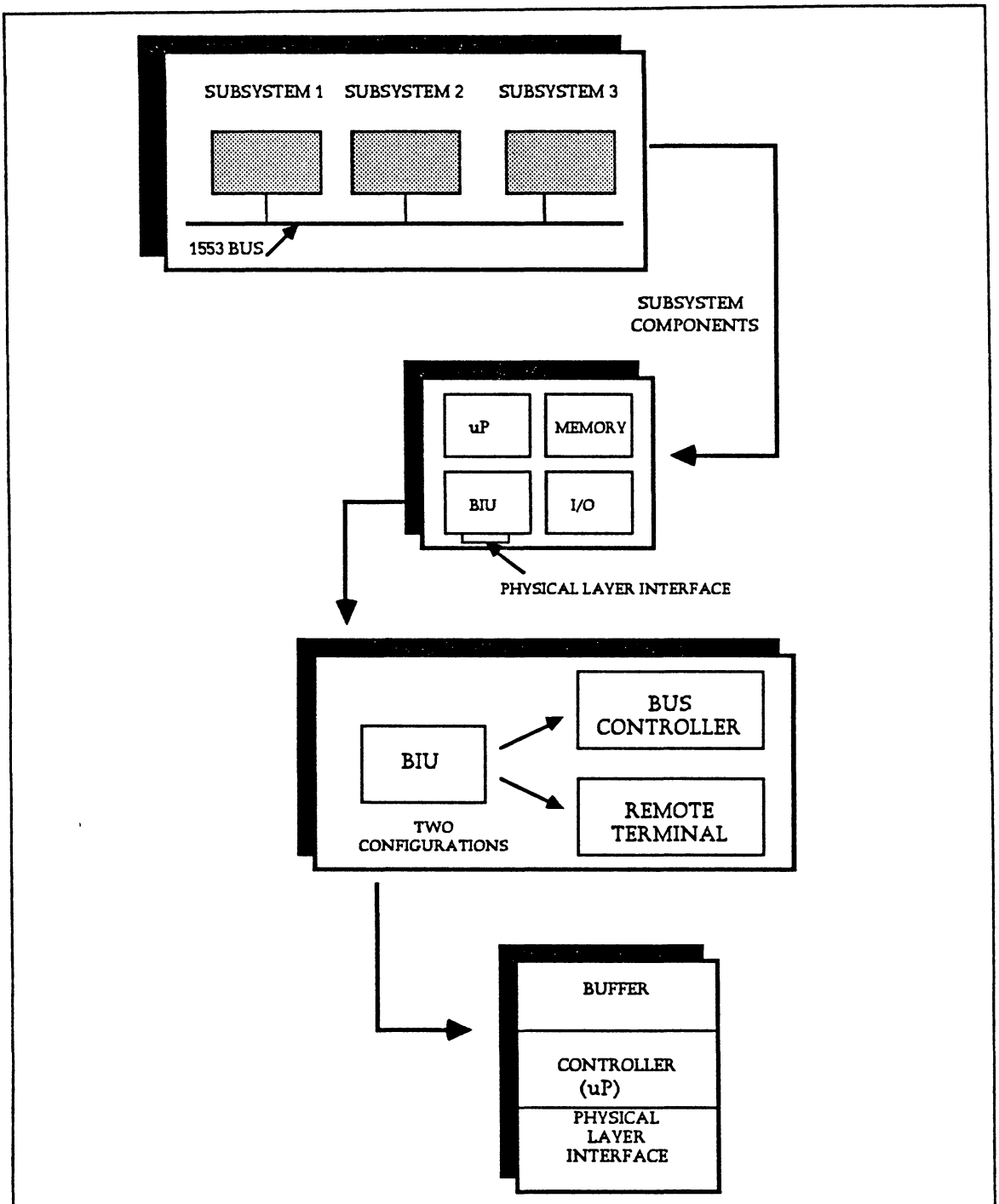

Figure 7. Distributed Computer System Model

5.1 Distributed Computer System Model

The distributed system model (Figure 7) is comprised of three subsystems that communicate via a 1553B bus. Each subsystem of the computer communication system contains an application processor and an electronic module called a BIU, that interfaces a serial data bus to the subsystem. Each BIU in the system can take on one of two roles: Bus Controller (BC) or Remote Terminal (RT). A BC is a system component that is responsible for initiation of any information transfer between subsystems. An RT is incapable of initiating information transfers and responds to command words sent out by the BC. Each RT responds to its own unique address as well as a common broadcast address.

The MIL-STD-1553B defines a three layered communication protocol that includes the physical layer, the medium access control layer, and the logical link layer. The physical layer and the medium access control layer, which are modeled as a portion of the simulation environment, define the interface between each subsystem and the 1553B bus. The logical link layer, which manages the protocol at a higher level, consists of the software that has been implemented in Ada.

There are four types of information transfers that are used in the communication protocol operation [4]:

1. Bus Controller to Remote Terminal data transfer (BC to RT)
2. Remote Terminal to Bus Controller data transfer (RT to BC)
3. Remote Terminal to Remote Terminal data transfer (RT to RT)
4. Mode Command

Mode commands are used in managing the information flow of the communication system and detecting possible errors having occurred in the form of data corruption. There are three types of words that are transferred during information transactions: command words, status words, and data words. The command word defines what type of information transfer is to occur. The status word is transmitted from an RT after an information transfer unless an error has occurred during the transfer or the command was sent in broadcast mode. The three data transfer types allow data words to be transferred between the BC and an RT or between any two RTs.

In a BC to RT data transfer, the BC sends out a command containing an RT's address (or the binary address 11111 in the case of a broadcast command) and the number of data words to be transferred. Once all the data words have been transferred, the receiving RT sends the BC its status word (except in the broadcast mode). The BC verifies the status word to determine whether or not any data corruption has occurred.

The RT to BC data transfer works in a manner similar to the BC to RT data transfer. Once the BC issues a command indicating that an RT to BC data transfer should occur, the BC first waits for the transmitting RT to send back its status word. After the RT has sent its status word to the BC, it then proceeds to transmit the required number of data words.

In an RT to RT data transfer, the BC sends out two command words. The first command word indicates which RT will be receiving the data while the second indicates which RT will be transmitting the data. The BC waits for the transmitting RT to send its status word. After the transmitting RT has transferred the correct number of

words, the receiving RT sends the BC its status word. The BC checks the status word to determine the validity of the information transfer.

Additional details of these information transfers can be found in Military Standard Aircraft Internal Time Division [25].

5.2 *Modeled System*

The modeled system consists of the Ada implementation of the 1553B communications protocol software and a modeled hardware environment. The hardware environment has been modeled to allow simulation of the different data transfers between subsystems in addition to exercising the 1553B communications protocol.

5.2.1 Hardware Environment

The hardware environment has been modeled at the functional level in Ada using package program units to encapsulate the components of a subsystem. Each subsystem of the hardware environment consists of an application processor, a BIU, a DMA controller, and a host memory. Each component is modeled as a task, with the exception of the host memory (which is modeled as an array), so that they may run concurrently. The block diagram of one subsystem model is shown in Figure 8 on page 57. The BIU consists of a memory buffer, a controller, and a physical layer

interface. The memory buffer is modeled as an array of words with each word consisting of 16 bits (elements).

As shown in Figure 8, the subsystem models communicate via a data bus also modeled at the functional level. The data bus, for purposes of simulation, is modeled as a parallel bus, rather than a serial bus. This simplification is reasonable since the main purpose of the simulation is to test the upset detection mechanisms incorporated into the communication protocol software.

The hardware environment described in this section uses package program units to encapsulate the components of the hardware environment. These packages consist of both a specification and a body. The specification of one subsystem is given below. The package body is provided in the appendix. The package specification contains the program unit specifications for the models used in the simulation environment. Each declaration in the package specification below corresponds to the modeled components in Figure 8.

```
package subsystem1 is

   HOST_MEMORY : array (0..150) of command_type;

   task DMA_1 is
     entry WORD(instruction_word : out command_type;address : in integer);
     entry STORE(word : in command_type; address : in integer);
   end DMA_1;

   task APPLICATION_PROCESSOR_1 is
     entry START;
     entry BIU_DONE;
   end APPLICATION_PROCESSOR_1;

   task BIU_1 is
     entry START;
     entry GO;
   end BIU_1;

end subsystem1;
```

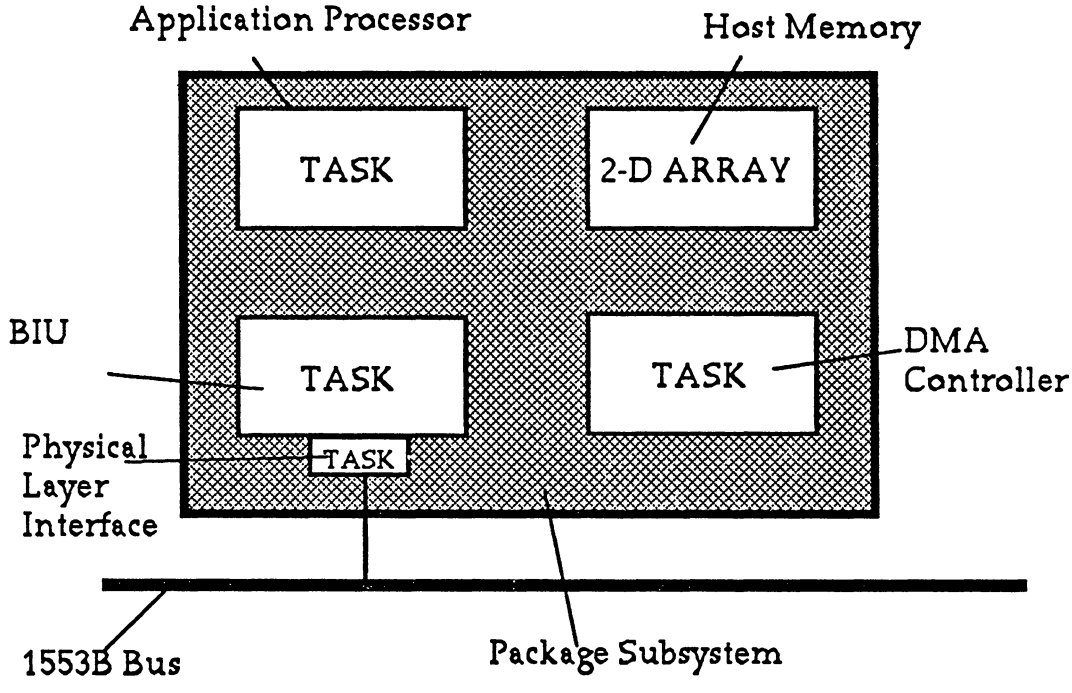

Figure 8. Block Diagram of Modeled Subsystem

The hardware registers associated with each BIU are defined in a separate package to provide greater clarity between the hardware environment portion and the implementation of the protocol communication software.

```
package PLI_1_INTERFACE is

  Mode_Data_Register : command_type;
  Instruction_Address_Register : integer;
  Base_Address_Register : command_type;
  Processor_Control_Register : command_type;
  Internal_Status_Register : command_type;
  Status_Word_Data_Register : command_type;
  Built_In_Test_Register : command_type;

end PLI_1_INTERFACE;
```

This package models the interface between the application processor and the BIU. Each subsystem package accesses its PLI_HARDWARE package by including the keywords:

```
with PLI_HARDWARE; use PLI_HARDWARE;
```

In a real-time situation, these registers would be accessed using low level I/O features. Shown below is how the registers were modeled for the hardware environment and how they would be implemented in a real-time situation.

Modeled environment:

```
MODE_DATA_REGISTER : command_type
```

where command_type is defined as an array(0..15) of integer.

Real-time environment:

```
bits : constant := 1;
type register is range 0..36556;
for register'SIZE use 16*bits;

MODE_DATA_REGISTER : register;
for MODE_DATA_REGISTER  use at 16#1542#;
```

The address clause, **use at 16#1542#**, specifies the physical address of the register while the SIZE attribute specifies the number of bits to be used in representing the type **register**.

5.2.2 Subsystem Communication

The vehicle by which each subsystem communicates with other subsystems is through the physical layer interface (PLI) of the BIUs. Each PLI has been modeled as a task to allow simultaneous execution of all the physical layer interfaces. The physical layer interfaces have been placed together in one package along with the data bus model. The package DATA_BUS is accessed by each subsystem package to allow inter - subsystem communication. Figure 9 on page 60 shows a block diagram of a data transfer between BIUs.

The PLIs are numbered according to which BIU they are associated with (e.g., **PLI_1** is associated with **BIU_1**). When data needs to be transferred from one subsystem to another subsystem, an entry call is made to the BIU's physical layer interface. For example, if **subsystem_1** needs to send data to **subsystem_4**, the statement

PLI_1.GET_DATA(data,synch)

causes the task PLI_1 to rendezvous with BIU_1 at the point of the entry call. Upon accepting this entry call, the task PLI_1, in turn, makes an entry call to the task **BUS** by issuing the statement **BUS.GET_DATA(data,synch)** along with the necessary parameters. When the entry call to the task **BUS** has been accepted, the task **PLI_1** completes its rendezvous with the task **BUS**. The calling BIU can then proceed past the point of rendezvous with its PLI. The information passed as parameters on each

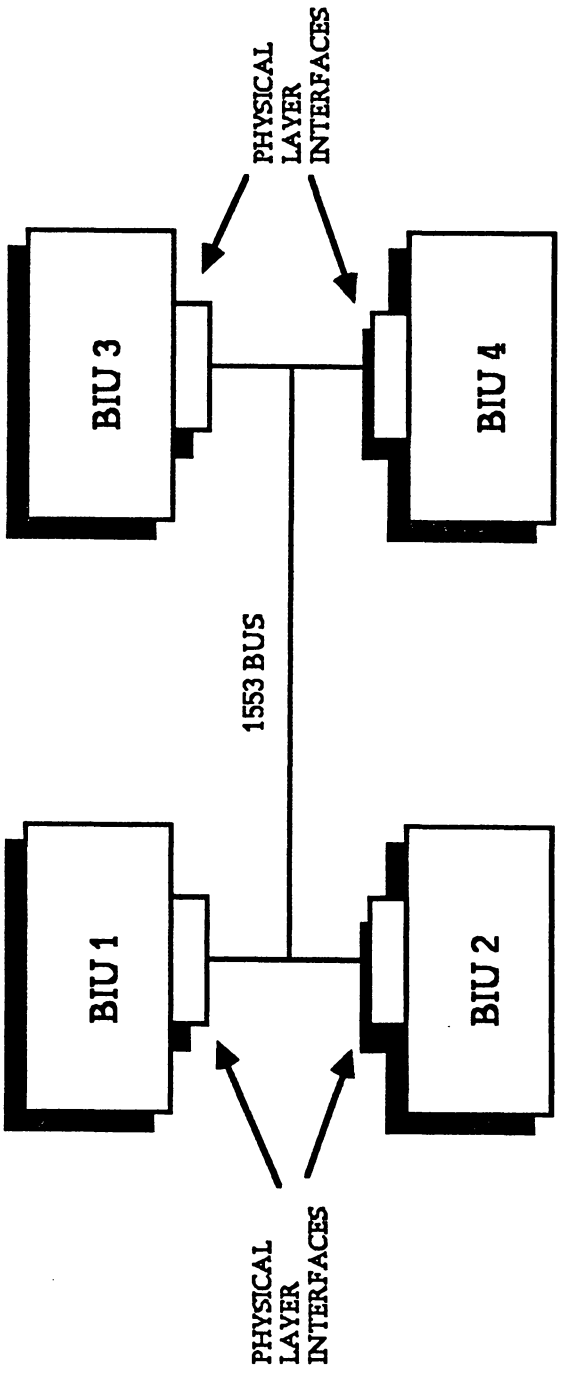

Figure 9. Inter - BIU communication

entry call are a 16-bit word and a *synch value*. The synch signal is a 1553B physical mechanism to distinguish between a command word and a data word.

The same procedure is followed when a BIU is expecting data from another BIU. The receiving BIU cannot proceed past the rendezvous point, where the entry call was made, until its PLI has retrieved data from the **BUS**. When the task **BUS** has been referenced by one of the PLIs, the information is passed onto the ***data lines*** as parameters of the entry call. The data on the lines is received by each PLI when the following statements are issued by the task **BUS**:

```
PLI_1.GET_DATA(data,synch);
PLI_2.GET_DATA(data,synch);
PLI_3.GET_DATA(data,synch);
```

By making the package **DATA_BUS** accessible to each subsystem model, any subsystem can receive/send data from/to any other subsystem. The source code for the package DATA_BUS is included in the source code for the modeled distributed computer system in Appendix A.

5.3 Ada Implementation of MIL-STD-1553B Communications Protocol

SMUD modifies that part of the modeled system which is responsible for exercising the 1553B communications protocol. The protocol software resides in the controller element which forms a part of the BIU. Each subsystem has been modeled identically with the exception of the BIU ***addresses*** and the application processor software.

Each application processor of the subsystems performs a unique function for that particular subsystem. The software structure of the 1553B protocol implementation, which is shown in Figure 10 on page 63, consists of a main tasking unit and fourteen subroutines:

1. **task BIU**
2. **procedure FORM COMMAND**
3. **function CONVERT**
4. **procedure BUS CONTROLLER**
5. **procedure REMOTE TERMINAL**
6. **procedure CHECK STATUS**
7. **procedure MODE COMMAND**
8. **procedure DATA TRANSFER**
9. **procedure BC TO RT**
10. **procedure RT TO RT**
11. **procedure RT TO BC**
12. **procedure PROCESS MODE COMMAND**
13. **procedure PROCESS DATA TRANSFER**
14. **procedure SEND DATA**
15. **procedure RECEIVE DATA**

Each BIU is assigned a unique address. The BIU labeled BIU_1 is given the binary address 00001 while BIU_2 and BIU_3 have the binary addresses 00010 and 00011, respectively. This software model can easily be extended to include up to 31 BIUs (addresses 0-30). Binary address 11111 is reserved for the BC to communicate in broadcast mode with selected RTs simultaneously. Since each BIU may have the capability of being either a bus controller or a remote terminal, two boolean flags that indicate a units configuration are included. The two flags, BUS_CONTROLLER_MODE and REMOTE_TERMINAL_MODE, are initialized in the declarative section of each BIU

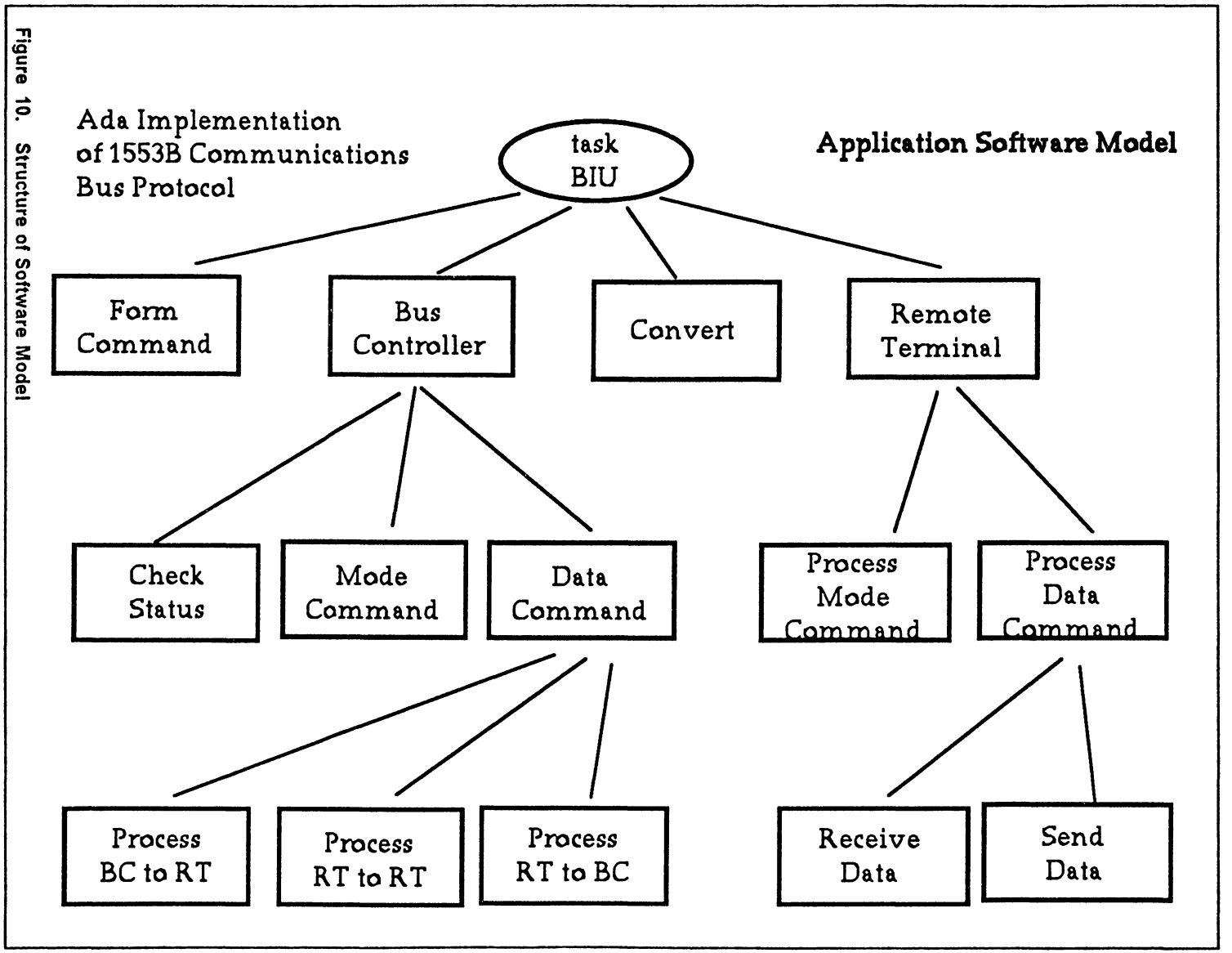

Figure 10. Structure of Software Model

model. Only one BIU model has BUS_CONTROLLER_MODE assigned the value of TRUE at any given instant while the other BIUs have REMOTE_TERMINAL_MODE assigned the value of TRUE.

Each task BIU has two procedures that are executed depending upon the configuration type they are: BUS_CONTROLLER and REMOTE_TERMINAL. For the BIU model that is designated the Bus Controller, the procedure BUS_CONTROLLER_1 determines whether the information transfer type is a mode command or a data transfer. Another procedure named MODE_COMMAND is called to further decode the command word. There are two types of mode commands: those that have an associated data word and those that do not. A data word that needs to be received/sent from/to the remote BIU is stored or read from the buffer. If the information transfer is a data transfer, the procedure DATA_TRANSFER is called. The command word passed into the procedure is checked for the type of data transfer and calls an appropriate procedure to carry out the data transfer.

Each BIU that is configured as an RT calls the procedure REMOTE_TERMINAL to determine whether or not it should respond to the data on the bus lines. If the data is a command word, the RT whose address is indicated in the command word further decodes the word. One of two procedures, PROCESS_MODE_COMMAND or PROCESS_DATA_TRANSFER is called to complete the required information transfer.

5.3.1 Simulation Example

The following section describes a simulation run of the Ada implementation of the 1553B communications protocol. The output produced by the simulation described

is provided at the end of this section with a detailed explanation of each information transfer. A driver routine (procedure ***SYSTEM_MODEL,*** shown below) is used to control the execution of the task units.

```
procedure SYSTEM_MODEL is

 begin

  APPLICATION_PROCESSOR_1.START;
  APPLICATION_PROCESSOR_2.START;
  APPLICATION_PROCESSOR_3.START;
  BIU_1.START;
  BIU_2.START;
  BIU_3.START;
  loop
    .
    .   Activate each BIU task periodically
    .
  end loop;
  BIU_1.done;
  BIU_2.done;
  BIU_3.done;
  DMA_1.done;
  DMA_2.done;
  DMA_3.done;
  BUS.done;
  PLI_1.done;
  PLI_2.done;
  PLI_3.done;
end SYSTEM_MODEL;
```

Each application processor of the subsystems is activated by the statements

```
APPLICATION_PROCESSOR_1.START
APPLICATION_PROCESSOR_2.START
APPLICATION_PROCESSOR_3.START
```

Once this is done, each application processor model initializes the various registers of its corresponding BIU. These registers contain information such as whether the BIU should behave as a BC or an RT and the address that will identifies a BIU. After

each application processor has initialized the registers, the driver routine then activates each BIU with the statements

BIU_1.START

BIU_2.START

BIU_3.START

Task BIU_1 is initially configured as the bus controller while tasks BIU_2 and BIU_3 are configured as remote terminals. The main program unit of BIU_1 is shown below. Each BIU model has identical code with the exception of the BIU **address** and the initial BIU configuration.

```
task body BIU_1 is

  mode : boolean;
  bus_controller_mode : boolean;
  remote_terminal_mode : boolean;
  command : command_type;
  command2 : command_type;
  halt : boolean;
  last_command_word : command_type;
  last_status_word : command_type;
  status_word : command_type := (0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0);


  begin
    accept START;
    if Processor_Control_Register(10) = 1 then
      bus_controller_mode := true;
      remote_terminal_mode := false;
    else
      bus_controller_mode := false;
      remote_terminal_mode := true;
    end if;

    COMMUNICATION_MODE_LOOP :
    loop
     select
     accept go;
     if bus_controller_mode then
      NEW_LINE;
      NEW_LINE;
      PUT_LINE("BIU_1 IS BUS CONTROLLER");
      NEW_LINE;
      FORM_COMMAND(command,command2,halt);
      Base_Address_Register(10..15) := command(5) & command(6..10);
      exit when halt;
      BUS_CONTROLLER_1(command,command2,mode);
```

```
      bus_controller_mode := mode;
      remote_terminal_mode := not mode;

     elsif remote_terminal_mode then
      Base_Address_Register(10..15) := command(5) & command(6..10);
      REMOTE_TERMINAL_1(mode);
      remote_terminal_mode := mode;
      bus_controller_mode := not mode;
      status_word := (0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0);
     end if;
    or
     delay 10.00;
     exit COMMUNICATION_MODE_LOOP;
    end select;
  end loop COMMUNICATION_MODE_LOOP;

  APPLICATION_PROCESSOR_1.BIU_DONE;
 end BIU_1;
```

Initially, since BIU_1 is configured as the bus controller, the procedure FORM_COMMAND is called to retrieve two instruction words from the host memory via the DMA controller. The procedure returns a command word based on the information in the instruction words. The procedure BUS_CONTROLLER is then called to process the command word. Meanwhile, BIU_2 and BIU_3 each call the procedure REMOTE_TERMINAL to process the command word as remote terminals. Each remote terminal determines whether or not the address field in the command word contains its address or the broadcast address. If not, the message error bit of the status word is set to one by the procedure REMOTE_TERMINAL. The RT then waits for new data to be placed on the bus. If the remote terminal does respond to the address, the command word is further decoded by the procedure REMOTE_TERMINAL to determine whether it is a mode command or a data transfer request. REMOTE_TERMINAL subsequently performs the appropriate action for the command word.

Listed below are examples of the simulation output. The sample output contains examples of all four types of information transfers discussed in section 5.1 Each infor-

mation transfer section identifies the BIU that is configured as the BUS CONTROLLER, the BIU configured as the REMOTE TERMINAL, and the type of information transfer that is taking place. In the case of a data transfer, the data words that are transferred are also listed. Commands are coded according to the 1553B definitions given in [25].

Example A Mode Command

BIU_1 IS BUS CONTROLLER
BIU_2 IS A REMOTE TERMINAL
COMMAND WORD SENT FROM BUS CONTROLLER : 0001010000000001

Mode Code : 00001 = > Synchronize

---- CHECKING STATUS WORD --------

STATUS WORD RECEIVED FROM REMOTE TERMINAL : 0001000000000000

Example B RT to BC data transfer

BIU_1 IS BUS CONTROLLER
BIU_2 IS A REMOTE TERMINAL
COMMAND WORD SENT FROM BUS CONTROLLER : 0001011110000010
COMMAND TYPE : DATA TRANSFER - RT TO BC

---- CHECKING STATUS WORD --------

STATUS WORD RECEIVED FROM REMOTE TERMINAL : 0001000000000000

2 DATA WORDS RECEIVED FROM REMOTE TERMINAL :

WORD 0 1111111111000000
WORD 1 1111111111100000

Example C RT to RT data transfer

BIU_1 IS BUS CONTROLLER
BIU_3 IS A REMOTE TERMINAL
BIU_2 IS A REMOTE TERMINAL
COMMAND WORD SENT FROM BUS CONTROLLER : 0001101000011110

COMMAND TYPE : DATA TRANSFER - RT TO RT

5 WORDS SENT TO REMOTE TERMINAL

---- CHECKING STATUS WORD --------

```
STATUS WORD RECEIVED FROM REMOTE TERMINAL :    0001000000000000

WORD 0    1111100000000000
WORD 1    1111110000000000
WORD 2    1111111000000000
WORD 3    1111111100000000
WORD 4    1111111110000000

 ---- CHECKING STATUS WORD --------

STATUS WORD RECEIVED FROM REMOTE TERMINAL :    0001100000000000
```

Example D BC to RT data transfer

```
BIU_1  IS BUS CONTROLLER
BIU_3  IS A REMOTE TERMINAL
COMMAND WORD SENT FROM BUS CONTROLLER :        0000111110000011

COMMAND TYPE :  DATA TRANSFER - BC to RT


3  DATA WORDS SENT TO REMOTE TERMINAL :

WORD 0  1111111111000000
WORD 1  1111111111100000
WORD 2  1111111111110000

 ---- CHECKING STATUS WORD --------

STATUS WORD RECEIVED FROM REMOTE TERMINAL :    0000100000000000
```

Example A shows the output result of a mode command information transfer. Examples B, C, and D show the results of data transfers. In Example B, an RT to BC data transfer, the transmitting RT first sends its status word to the BC, followed by the required number of data words. In Example C, the data transfer is between two remote terminals. According to protocol specifications, the transmitting RT first sends back its status word to be read by the BC. It then sends the required number of data words to be read by the receiving terminal. Once all the data words have been transferred, the receiving RT sends back its status word. Example D shows the out-

put results of the remaining data transfer type: BC to RT. In this example, the data words are transferred first, followed by the status word of the receiving RT.

This chapter has described a distributed computer system model developed using the tasking features of Ada. The modeling of the intercommunications of individual subsystems of the 1553B bus structure has been described. The simulation model was thoroughly tested for proper execution of all 1553B mode commands and data transfers. Each information transfer was successfully carried out. The next step after verifying the correct operation of the modeled system was to modify the 1553B protocol implementation to test the upset detection mechanisms. The next chapter discusses how these mechanisms were tested.

Chapter 6

Testing and Results

This chapter will discuss the effectiveness of the upset detection techniques inserted into Ada application software. Also discussed are the time and memory overheads incurred by the incorporation of a block-checking structure and watchdog timer mechanism. Testing methodology and performance measures are presented here.

The testing methodology consisted of two parts. The first part was concerned with determining the upset detection coverage provided by the block-checking constructs. The second part was concerned with the effectiveness of the watchdog timer in detecting deviations in program flow that went undetected by the block-checking constructs.

6.1 Testing

Testing of the upset detection mechanisms involved simulating the effects of SEUs causing a deviation from normal program flow. To simulate this type of upset, the application model software containing the detection mechanisms was once again altered, this time by hand. A generalized block diagram of how the protocol software was modified to include several entry points is shown in Figure 11 on page 73. These entry points would be activated to simulate the result of an erroneous jump by the program counter.

The following sample program serves to illustrate how each procedure and function in the application software was altered by hand for testing purposes. The sample program consists of four output statements to give an external indication of the execution flow.

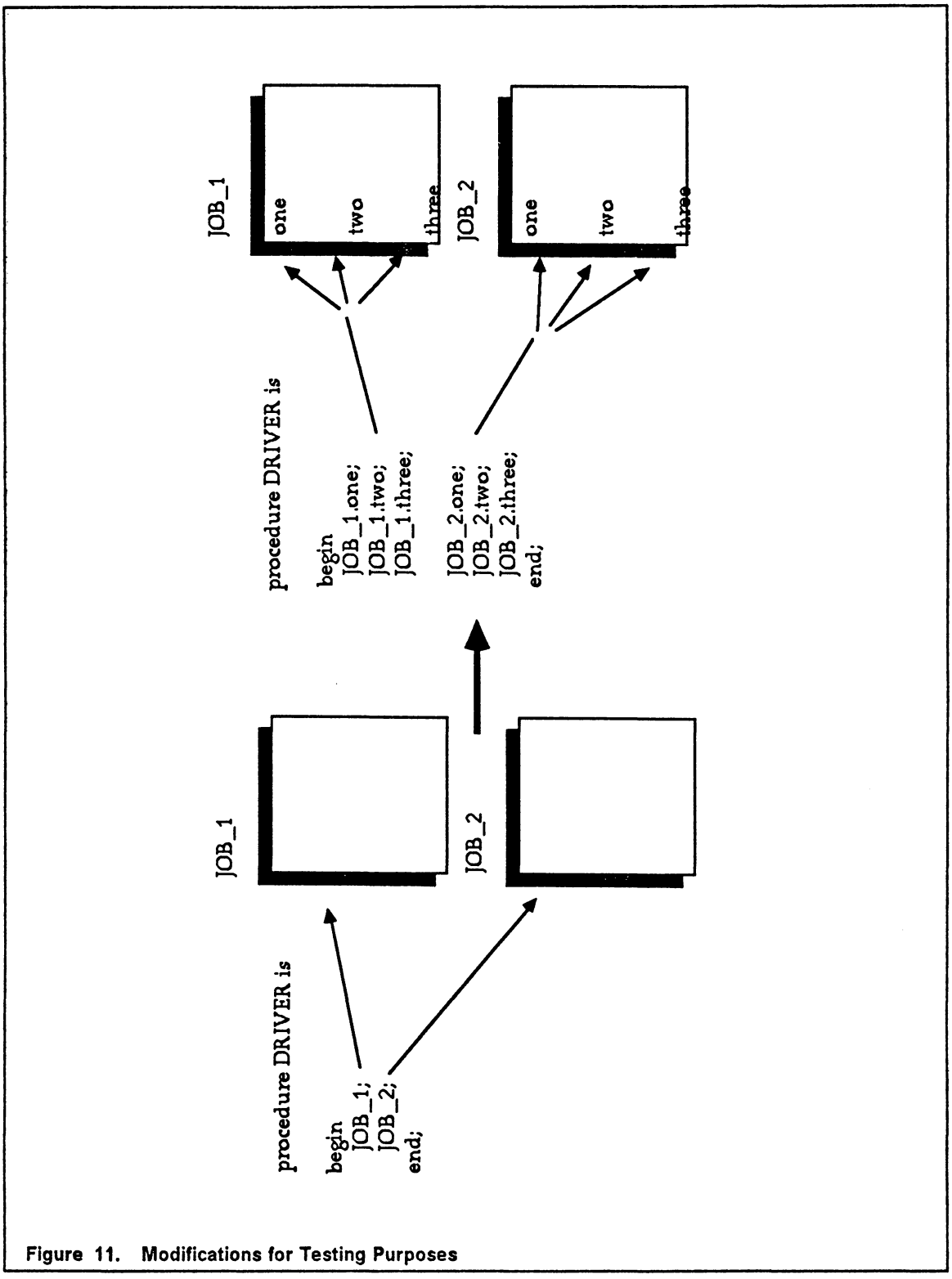

Figure 11. Modifications for Testing Purposes

```
procedure test is

TAG : integer := 0;

begin

  IF tag = 0 then
    tag := 1;
  else
    raise INCORRECT_TAG;
  end if;

  WDT1 := 255;

  PUT_LINE("statement 1");
  PUT_LINE("statement 2");
  PUT_LINE("statement 3");
  PUT_LINE("statement 4");

  if tag = 1 then
    tag := 0;
  else
    raise INCORRECT_TAG;
  end if;

end test;
```

```
task body test is

TAG : integer := 0;

begin

 loop
  select
  accept ONE;
  IF tag = 0 then
    tag := 1;
  else
    HANDLE;
  end if;

  WDT1 := 255;
  PUT_LINE("statement 1");
or
  accept TWO;
  PUT_LINE("statement 2");
or
  accept THREE;
  PUT_LINE("statement 3");
or
  accept FOUR;
  PUT_LINE("statement 4");
  if tag = 1 then
    tag := 0;
  else
    HANDLE;
  end if;
  exit;
end select;
end loop;

end test;
```

The output shown below is the result of (a) and (b) for a correctly executed set of instructions.

```
statement 1
statement 2
statement 3
statement 4
```

The modified test code will output the same sequence of statements provided each entry point is accessed successively. A driver routine was required to invoke the

entry calls of the modified block. In the protocol software, each program unit behaves as a driver to each program unit contained within its scope.

The driver routine shown below accesses each entry point in succession. The output resulting from the execution is identical to that of the original test program.

```
procedure driver is

 begin

  TEST.ONE;
  TEST.TWO;
  TEST.THREE;
  TEST.FOUR;

 end;
```

Since the altered protocol software provides the means for selection of any four output statements, a program flow deviation can be simulated by eliminating an entry call from the driver routine and successively invoking the remaining entry calls. In order to simulate the effect of an SEU that causes an erroneous block to start executing, the driver routine bypasses the first entry point. This results in the omission of the TAG initialization. As an example, consider the driver routine shown below:

```
procedure driver is

 begin

   TEST.THREE;
   TEST.FOUR;

 end;
```

Output from test program

```
statement 3
statement 4
incorrect block entry
```

Upon reaching the TAG check at the end of the test program unit, the TAG is checked for the identifying value of the block. Since the TAG was not reset at the start, the value is incorrect and the exception INCORRECT_TAG is raised. The exception handler displays a statement indicating an incorrect block entry has occurred.

The previous example served to illustrate a program flow deviation into another block. However, SEUs that cause execution to continue in the same block, though skipping several instructions, will not be detected by the block-checking constructs. For example, consider the following driver sequence:

```
procedure driver is

begin

  TEST.ONE;
  TEST.FOUR;

end;
```

Output from test program

```
  statement 1
  statement 4
```

The driver accesses the first entry point, initializing the TAG to the identifying value of the block. The second and third entry points have been omitted, simulating a program flow deviation contained within the block. Upon reaching the end of the block, the TAG is checked for the ID value. Since the TAG was initialized properly, the deviation is not detected.

Detection of this type of program flow deviation depends on the location where execution resumes in the block. As discussed earlier, a block containing a loop structure poses a potential infinite loop situation. A deviation near or into a loop structure may

prevent the setting of necessary loop variables, thereby leading to the possibility of an infinite loop. It is for this purpose that a watchdog timer mechanism was also utilized.

6.2 *Simulation of Watchdog Timer*

In addition to testing the effectiveness of the block-checking constructs, the ability of the watchdog timer in detecting SEUs was also tested. A hardware watchdog timer has been modeled at the functional level using the tasking facilities and VAX/Ada library package CALENDAR available in Ada. Using the package CALENDAR allows access to the clock and time functions. The model of the watchdog timer is given below:

```
with TEXT_IO; use TEXT_IO;

package wdt_time is

task WDT is
  entry START(time : in duration);
end WDT;

end wdt_time;

package body wdt_time is

task body  WDT is

package duration_io is new fixed_IO(duration); use duration_io;
WDT_time : duration := 1.0;

begin

  loop
   select
    accept START(time : in duration) do
      WDT_time := time;
    end START;
```

```
    or
     delay WDT_time;
     PUT_LINE("WDT expired");
     exit;
    end select;
   end loop;

end WDT;

end WDT_TIME;
```

Each watchdog timer initialization in the modified 1553B protocol implementation invoked an entry call to the watchdog timer task. The value of initialization was passed from the protocol into the watchdog timer task through the entry point **START**. The parameter, **time**, was then assigned to a local variable, **WDT_time**, of the timer task. It is this variable that is evaluated in the delay statement. If a new value is assigned to **WDT_time**, the delay statement starts a new countdown time. If the delay time reaches the value of WDT_time, an error message is displayed.

6.3 *Results*

The two upset detection mechanisms yielded an overall fault coverage of 91%. Undetected deviations were mainly due to those program flow deviations that were contained within the block. The simulated deviations were limited in that improper instruction boundaries could not be accesed, nor could restricted areas of memory be accessed. This was because testing was carried out at high-order language level running under the protectiveness of the VMS operating system. As shown in Table 1, 71% of the deviations were detected by the block-checking constructs, while 20% were detected by a watchdog timer timeout. As stated earlier, the main reason for

the undetected program flow deviations was that the deviation was contained in the block and did not result in an infinite loop. Thus, neither block-checking nor watchdog timer detected these faults. However, some deviations were detected by the 1553B protocol software before being detected by the SMUD detection mechanisms.

6.4 *Time and Memory Overheads*

Time overhead is defined as the additional time needed to execute a defined set of instructions while memory overhead is defined as the amount of extra memory required to include additional upset detection instructions. The time and memory overheads incurred by the upset detection mechanisms are shown in Tables 2, 3, 4, and 5. Tables 2 and 3 provide data concerning the memory overhead incurred by incorporating the upset detection mechanisms. The two watchdog timer strategies discussed in Chapter 3 required different amounts of additional bytes. Memory overhead values have been obtained for both versions. The memory overhead for each program unit in the 1553B protocol software is listed. The values represent the total number of bytes in both the unmodified and modified versions and the percent increase in the number of bytes of storage necessary. The percent increase is related to the original number of bytes in a procedure. A procedure that contains a greater number of bytes will incur a lower percentage memory overhead than will a smaller procedure. The total number of additional bytes required was 947; the average size increase of a program unit was 63.13 bytes.

Simulated SEUs	**55**	**Percent**
Detected by TAG Checking	39	71
Detected by Watchdog Timer	11	20
Undetected	5	9
Total Detections	**50**	**91**

Table 1. Detection of SEUs by Upset Detection Techniques

In addition to an incurred memory overhead, two time overheads were incurred. A slight time overhead was incurred for both the compilation time of the Ada application software system and the execution time for a defined fixed length sequence. As with the memory overheads, time overhead values have been obtained for both versions of the watchdog timer strategies. The compilation time overhead represents a one-time overhead while the execution time overhead is incurred repentedly. The compilation time for the unmodified and modified application models were dependent on the load of the host computer. Similarly, the execution times varied according to the load and the non-determinism of task selection. For this reason, ten golden unmodified simulation runs were executed for both the modified and unmodified protocol models so that an average value could be obtained for each.

Unit name	Unmodified #bytes	Modified #bytes	% Increase
BIU 1	402	498	23.88
FORM COMMAND	378	406	7.4
CONVERT	479	499	4.17
BUS CONTROLLER	235	297	26.38
CHECK STATUS	561	614	9.45
MODE COMMAND	469	555	18.34
DATA TRANSFER	503	662	31.61
BC TO RT	104	144	38.46
RT TO RT	71	114	60.56
RT TO BC	443	495	11.74
REMOTE TERMINAL	324	394	21.60
PROCESS MODE COMMAND	1301	1332	2.38
PROCESS DATA TRANSFER	669	785	17.34
RECEIVE DATA	338	389	15.09
SEND DATA	100	140	40.00
AVERAGE VALUES	**425.13**	**488.27**	**14.86**

Table 2. Memory Overhead with Upset Detection Technique (version I)

Unit name	Unmodified #bytes	Modified #bytes	% Increase
BIU 1	402	460	14.43
FORM COMMAND	378	407	7.68
CONVERT	479	500	4.38
BUS CONTROLLER	235	276	17.45
CHECK STATUS	561	615	9.63
MODE COMMAND	469	522	11.30
DATA TRANSFER	503	553	9.94
BC TO RT	104	145	39.42
RT TO RT	71	115	61.97
RT TO BC	443	496	11.96
REMOTE TERMINAL	324	373	15.12
PROCESS MODE COMMAND	1301	1333	2.46
PROCESS DATA TRANSFER	669	717	7.17
RECEIVE DATA	338	390	15.38
SEND DATA	100	141	41.00
AVERAGE VALUES	**425.13**	**469.53**	**10.44**

Table 3. Memory Overhead with Upset Detection Technique (version II)

Unit name	Unmodified CPU sec	Modified CPU sec	% Increase
Compilation Time	47.12	53.33	13.18
Execution Time	1.411	1.438	1.91

Table 4. Time Overhead (version I)

Unit name	Unmodified CPU sec	Modified CPU sec	% Increase
Compilation Time	47.12	51.40	9.08
Execution Time	1.411	1.434	1.63

Table 5. Time Overhead (version II)

Chapter 7

Conclusion

The work presented in this thesis involves upset detection techniques for computer systems subject to high amounts of radiation. Single event upsets resulting from these environmental particles can affect several hardware components of the system. The focus of this thesis has been on those single event upsets occurring in the program counters of microprocessors. The effect of an SEU in the program counter results in a deviation from normal flow of program execution. In order to not impose additional hardware requirements, a software technique of detecting SEUs in program counters was investigated.

A program has been written in Ada to automatically insert a block checking structure and watchdog timer mechanisms into Ada application software. The program, Software Modifier for Upset Detection (SMUD) requires little interactive information from the application programmer and relies mainly on available directives to determine how to modify each program unit.

A system model that employs the 1553B bus communications protocol has been used to verify proper operation of the automated modification. The effectiveness of the upset detection techniques was determined through computer simulations. All information transfer formats of the protocol have been incorporated and the model has been extensively tested. A sample of the modified code output and simulation results have been presented in this thesis. The modified system model utilizing the upset detection mechanisms has been included in Appendix A. Examples of software structures using the available directives are provided in Chapter 3. These examples serve to clarify the use and effects of the directives.

The initial system model was slightly different from the one presented here, in that, much of the logical link layer was implemented using the tasking feature of Ada. However, since the simulation was being run on a uniprocessor (DEC's VAX 11/785), inter-task communication consumed an inordinate amount of time. This problem was remedied by only using tasks to model the simulation environment, while procedures and functions were used in implementing the logical link layer. These modifications will allow different entry points in the software. The simulation runs, besides validating the fault detection mechanisms, also tested the 1553B protocol.

The results obtained from the testing phase yielded an overall SEU detection rate of 91% (71% were due to the block-checking constructs and 20% were due to the watchdog timer mechanism). A small percentage of the deviations that went undetected by either method were detected by the 1553B protocol definition itself.

References

1. Elbert, T.F.,***Embedded Programming in Ada***, Van Nostrand Reinhold Company, New York, 1986.

2. United States Department of Defense, ***Reference Manual for the Ada Programming Language***, ANSI/MIL-STD-1815A-1983, February 17, 1983.

3. Volz,R., Mudge,T.N. Mayer,J., and Naylor,A.,***Some Problems in Distributed Real-Time Ada Programs Across Machines***, Proceedings of Ada Intl. Conference, Paris, May 1985 pp. 72-84.

4. Tedd,M., Crespi-Reghizzi,S., and Natali,A.,***Ada for Multi-microprocessors***, Cambridge University Press, Great Britain, 1984.

5. Schmid, M.E., Trapp, R., Davidoff, A., & Masson,G., ***Upset Exposure By Means of Abstraction Verification***, 12th Annual International Conference on Fault Tolerant Computing, pp. 237-244.

6. Dahbura, A. and Masson, G. ***A New Diagnosis Theory as the Basis of Intermittent-Fault/Transient-Upset Toelrant System Design***, 12th Annual International Conference on Fault Tolerant computing, 1982, pp. 353-356.

7. Westermeier, T.F. and Hansen, H.E., ***The Use of High Order Languages in Microprocessor-Based Systems***,

8. Sosnowski, J., ***Evaluation of Transient Hazards in Microprocessor Controllers***, 16th Fault Tolerant Computing Symposium, 1986, pp. 364-369.

9. Sosnowski, J., ***Transient Fault Effects in Microprocessor Controllers***,REL-CON EUROPE 86 Conference, Copenhagen, June 1986.

10. Sosnowski, J., ***Transient Fault Tolerance in a Data Acquisition System***, Micro-processing and Microprogramming 16,1985.

11. Halse, R.G., and Preece, C., ***Erroneous Execution and Recovery in Microprocessor Systems***, Software and Microsystems, Vol. 4. no. 3, June 1985, pp.63-70.

12. Oak, J., ***Block Checking Approach to Self-Testing Software***, MS Thesis EE, Virginia Tech, Sept. 1984.

13. Siewiorek, D. and Swarz, R.,***The Theory and Practice of Reliable System Design***, Bedford, MA, Digital, 1982

14. .Shin, K. and Penz,D., ***Modeling of Concurrent Task Execution in a Distributed System for Real-Time Control,*** IEEE Transactions on Computers,vol.C-36, No.4, pp. 500-516, April 1987.

15. Wheeler,T., ***Distributed Program Design in Ada: An Example***, 2nd Intl. Conference on Ada Applications and Environments, 1986, pp.21-29.

16. Cornhill,D., ***Four Approaches to Partitioning Ada Programs for Execution on Distributed Targets***,IEEE Computer Society, 1984 Conference on Ada Application and Environments, Oct. 15-18, St. Paul, Minnesota.

17. Li, Kai Wing, ***Detection of Transient Faults in Microprocessors by Means of External Hardware***, Master's Thesis, March 1984, Blacksburg, Va.

18. Schwan, K. Bihari, T.E., Blake, B.A., ***Adaptive, Reliable Software for Distributed and Parallel Real-Time Systems***, IEEE Computer Society, 1987 Conference on Distributed Environments, pp.32-42.

19. Wellings, A.J., Tomlijson, G.M., Keefe, D., and Wand, J.C.,***Communication between Ada Programs***, IEEE Computer Society, 1984 Conference on Ada Applications and Environments, Oct 15-18, St. Paul, Minnesota, pp. 145-152.

20. Stammers, R.A. ***Ada on Distributed Hardware***, Proceeding Workshop on Hardware Supported Implementation on Concurrent Language in Distributed Systems, 1985, Elsevier Science Publishers, pp. 35-40.

21. Randell,B., Lee.R.A., Treleaven, P.C., ***Reliability Issues in Computing system Design***, Computing Surveys, Vol. 10 no. 2, June 1978, pp. 123-165.

22. Rasmussen, R., ***Computing in the Presence of Soft Bit Errors***, Proceedings of Americans Control Conference, June 1984, pp. 1125-1130.

23. Baker,T.P. and Riccardi,G.A., ***Implementing Ada Exceptions***, IEEE Software, Sept. 1986, pp. 42-51.

24. Booch, G. ***Software Engineering with Ada***, Benjamin Cummings Publishing Co., Menlo Park, CA, 1983.

25. Military Standard (MIL-STD-1553B), Aircraft Internal Time Division, Command/Response Multiplex Data Bus, Department of Defense, USA , 21 September 1978.

Appendix A

1553B System Model

One subsystem of the distributed computer system is provided in this appendix. As discussed in Chapter 5, only the protocol software which is exercised by the BIU has been modified by SMUD to contain block-checking constructs and a watchdog timer mechanism. These modifications have been boldfaced to distinguish them from the original protocol software. The software that models the other components of the subsystem (DMA, application processor, data bus, PLI, etc.) have also been included.

```
-- This procedure provides the entry signal for the application
-- processors and the BIUs of the system. The loop structure
-- controls the number of times the BIUs get activated.
--
with TEXT_IO;use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with DATA_BUS; use DATA_BUS;
with subsystem_1,subsystem_2,subsystem_3;
use subsystem_1,subsystem_2,subsystem_3;
procedure SYSTEM_MODEL is


  count : integer := 0;

  begin
   PUT_LINE("SIMULATION OF MIL-STD-1553B PROTOCOL");
   NEW_LINE;
   NEW_LINE;
   NEW_LINE;

   APPLICATION_PROCESSOR_1.START;
   APPLICATION_PROCESSOR_2.START;
   APPLICATION_PROCESSOR_3.START;
   BIU_1.START;
   BIU_2.START;
   BIU_3.START;
   loop

     select
      BIU_1.GO;
     or
      delay 1.00;
     end select;

     select
      BIU_2.GO;
     or
      delay 1.0;
     end select;

     select
       BIU_3.GO;
     or
       delay 1.0;
     end select;

     count:= count+ 1;
     exit when count=25;
  end loop;
  BIU_1.done;
  BIU_2.done;
  BIU_3.done;
  DMA_1.done;
  DMA_2.done;
  DMA_3.done;
  BUS.done;
```

```
    PLI_1.done;
    PLI_2.done;
    PLI_3.done;
end SYSTEM_MODEL;
```

```
-- The hardware interface between the application processor and the
-- BIU is represented in the simulation environment as a package.
-- Placing the registers of the BIU together in one package allows
-- registers of the BIU to be accessed by both the application processor
-- model and the BIU model.
--
--
with DATA_BUS;use DATA_BUS;
package PLI_1_INTERFACE is

   Mode_Data_Register : command_type;
   Instruction_Address_Register : integer;
   Base_Address_Register : command_type;
   Processor_Control_Register : command_type;
   Internal_Status_Register : command_type;
   Status_Word_Data_Register : command_type;
   Built_In_Test_Register : command_type;

end PLI_1_INTERFACE;
```

```
-- This package consists of the software models of the processing
-- units, DMA controller, and the host memory.  The processing units
-- and the DMA controller are modeled as tasks while the host memory
-- is modeled as a 2-D array.  Both processing units, the APPLICATION
-- PROCESSOR and BIU, are able to access the host memory.  The APPLICATION
-- PROCESSOR, however, is able to access the host memory directly
-- while the BIU must retrieve and store data via the DMA controller.
-- In this simulation environment, the task BIU invokes the task DMA
-- to access the host memory.
--
-- The Ada implementation of the 1553B protocol is the software
-- which is contained within the scope of the BIU task unit.
-- The software in the application processor task unit loads the
-- host memory with instruction words and sets the registers of the BIU.


with DATA_BUS; use DATA_BUS;
with PLI_1_INTERFACE; use PLI_1_INTERFACE;
with TEXT_IO; use TEXT_IO;
with upset_detection1; use upset_detection1;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
package subsystem_1 is

   HOST_MEMORY : array (0..150) of command_type;

   task DMA_1 is
     entry WORD(instruction_word : out command_type;address : in integer);
     entry STORE(word : in command_type; address : in integer);
     entry DONE;
   end DMA_1;

   task APPLICATION_PROCESSOR_1 is
     entry START;
     entry BIU_DONE;
   end APPLICATION_PROCESSOR_1;

   task BIU_1 is
     entry START;
     entry GO;
     entry done;
   end BIU_1;

end subsystem_1;
```

```
package body subsystem_1 is

task body DMA_1 is

-- This task body models a DMA controller at the functional level.
-- It is used to retrieve and store words from the host memory for
-- the BIU model.  An address value is provided in decimal form
-- to access the memory, which is modeled as a 2-D array.
--
-- The two branches of the select statement correspond to the
-- functions of retrieving a word (accept WORD) and storing a
-- word (accept STORE) from and to the host memory.


begin
  loop
   select
    accept WORD(instruction_word : out command_type;address : in integer) do
      instruction_word := host_memory(address);
    end WORD;
   or
    accept STORE(word : in command_type; address : in integer) do
      host_memory(address) := word;
    end STORE;
   or
    accept done;
    exit;
   end select;
  end loop;
end DMA_1;
```

```
task body APPLICATION_PROCESSOR_1 is

-- This task body models the function of the application processor
-- at the functional level.  It is used to initialize the host
-- memory with instruction words and the registers of the BIU.
-- A procedure is called to initialize the host memory.
--


procedure INIT_1 is
 begin
   host_memory(100) := (1,1,0,0,0,0,1,1,1,1,1,1,1,1,1,1);
   host_memory(101) := (1,0,0,0,1,1,0,0,1,1,0,1,1,1,1,1);
   host_memory(102) := (1,1,1,1,0,1,0,0,1,1,0,1,1,1,1,1);
   host_memory(103) := (1,0,0,1,0,1,0,0,1,1,1,1,1,1,1,1);
   host_memory(104) := (1,1,1,1,0,1,0,0,1,1,0,1,1,1,1,1);
   host_memory(105) := (1,0,0,1,0,1,0,1,0,0,0,1,1,1,1,1);
   host_memory(106) := (1,1,1,1,0,0,0,0,1,1,1,0,1,0,1,0);
   host_memory(107) := (0,0,1,1,1,1,0,0,1,1,0,0,1,1,0,1);
   host_memory(108) := (1,1,1,1,0,0,0,0,1,1,0,0,0,1,0,1);
   host_memory(109) := (0,1,0,0,0,1,0,0,1,1,1,0,0,1,0,0);
   host_memory(110):= (1,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0);
   host_memory(111):= (0,0,1,0,0,1,0,0,1,1,1,0,0,1,0,1);
   host_memory(112):= (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
   host_memory(113):= (1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1);
   host_memory(114):= (1,1,0,0,1,0,0,1,0,0,0,1,1,1,1,1);
   host_memory(116):= (1,1,0,0,1,1,0,0,1,1,0,1,1,1,1,1);
   host_memory(118):= (0,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1);
   host_memory(119):= (0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0);
 end;


begin    -- application processor --
   INIT_1;
   accept START do
     Processor_Control_Register := (0,0,1,1,0,1,1,0,1,0,1,0,1,1,0,0);
     Mode_Data_Register := (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1);
     Instruction_Address_Register := 100;
     Base_Address_Register(0..9) := (0,0,0,0,0,0,0,0,0,0);
   end START;
   accept BIU_DONE;
end APPLICATION_PROCESSOR_1;

task body BIU_1 is separate;



end subsystem_1;
```

```
separate (subsystem_1)
task body biu_1 is

TAG1 : integer := 0;
-- This tasks performs the function of acting as a BIU that can be
-- configured either as a remote terminal or a bus controller.  The
-- configuration is determined by examining the Processor Control
-- Register that was initialized by the application processor.
-- The registers of the BIU model are accessible by using the
-- BIU_INTERFACE package.
--
-- If the BIU is configured as a bus controller, the procedure
-- FORM_COMMAND is called to form the command words that are
-- to be transmitted.  If the BIU is configured as a remote terminal,
-- all data placed on the BUS is examined for a command word
-- containing its "address" in the address field.
--
-- The BUS model is made accessible by using the package DATA_BUS,
-- which contains the physical layer interface models for each
-- which contains the BUS model and the physical layer interface
-- models for each BIU in the system.
--


mode : boolean;
bus_controller_mode : boolean;
remote_terminal_mode : boolean;
command : command_type;
command2 : command_type;
halt : boolean;
last_command_word : command_type;
last_status_word : command_type;
status_word : command_type := (0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0);
```

```
procedure form_command(command ,command2 : out command_type;halt : out boolean)
is


TAG2 : integer := 0;
-- This procedure is used to form the command words transmitted by the
-- BIU acting as Bus Controller. Two instruction words are retrieved
-- from the host memory via the task DMA controller. The instruction
-- words are examined to verify that they are different. If they are
-- the same, bit 0 of the Built_In _Test_Register is set; otherwise
-- the command words are formed.


 IW1,IW2 : command_type;

 begin

If TAG2 = 0 then
  TAG2 := 2;
else
  HANDLER;
End if;

WDT1 := 255;

   -- retrieve instruction words --

   halt := false;
   DMA_1.WORD(IW1,INSTRUCTION_ADDRESS_REGISTER);
   Instruction_Address_Register := Instruction_Address_Register + 1;
   DMA_1.WORD(IW2,INSTRUCTION_ADDRESS_REGISTER);
   Instruction_Address_Register := Instruction_Address_Register + 1;

   -- form command words --

   if IW1(6..10) /= IW2(6..10) then

     if Processor_Control_Register(0..4) = IW1(6..10) then
       command(0..4) := IW2(6..10);
       command(5) := 1;
       command(6..10) := IW2(11..15);
       command2(0..4) := Processor_Control_Register(0..4);
     elsif Processor_Control_Register(0..4) = IW2(6..10) then
       command(0..4) := IW1(6..10);
       command(5) := 0;
       command(6..10) := IW1(11..15);
       command2(0..4) := Processor_Control_Register(0..4);
     else
       command(0..4) := IW1(6..10);
       command(5) := 0;
       command(6..10) := IW1(11..15);
       command2 (0..4) := IW2(6..10);
       command2 (5) := 1;
       command2 (6..10) := IW2(11..15);
       command2 (11..15) := IW2(0..4);
     end if;
```

```
    command(11..15) := IW2(0..4);
  else
    Built_In_Test_Register(0) := 1;
  end if;

  If (IW1(0..1) = (0,0)) and (IW1(4..5) = (0,0)) then
    halt := true;
    PUT_LINE("End of Cycle ==> Interrupt Processor");
  end if;

  If TAG2 /= 2 then
    HANDLER;
  else
    TAG2 := 0;
  End if;
end;
```

```
function convert_1(temp : in command_type) return integer is


TAG3 : integer := 0;
-- This purpose of this function is to convert five elements of
-- an array to the equivalent decimal value.  The input is in
-- binary form.

 integer_value : integer;

 begin

If TAG3 = 0 then
  TAG3 := 3;
else
  HANDLER;
End if;

WDT1 := 255;

   if temp(0..4)   = (0,0,0,0,0) then
     integer_value := 0;
   elsif temp(0..4) = (0,0,0,0,1) then
     integer_value := 1;
   elsif temp(0..4) = (0,0,0,1,0) then
     integer_value := 2;
   elsif temp(0..4) = (0,0,0,1,1) then
     integer_value := 3;
   elsif temp(0..4) = (0,0,1,0,0) then
     integer_value := 4;
   elsif temp(0..4) = (0,0,1,0,1) then
     integer_value := 5;
   elsif temp(0..4) = (0,0,1,1,0) then
     integer_value := 6;
   elsif temp(0..4) = (0,0,1,1,1) then
     integer_value := 7;
   elsif temp(0..4) = (0,1,0,0,0) then
     integer_value := 8;
   elsif temp(0..4) = (0,1,0,0,1) then
     integer_value := 9;
   elsif temp(0..4) = (0,1,0,1,0) then
     integer_value := 10;
  elsif temp(0..4) = (0,1,0,1,1) then
     integer_value := 11;
  elsif temp(0..4) = (0,1,1,0,0) then
     integer_value := 12;
  elsif temp(0..4) = (0,1,1,0,1) then
     integer_value := 13;
  elsif temp(0..4) = (0,1,1,1,0) then
     integer_value := 14;
  elsif temp(0..4) = (0,1,1,1,1) then
     integer_value := 15;
  elsif temp(0..4) = (1,0,0,0,0) then
     integer_value := 16;
  elsif temp(0..4) = (1,0,0,0,1) then
     integer_value := 17;
```

```
    elsif temp(0..4) = (1,0,0,1,0) then
       integer_value := 18;
    elsif temp(0..4) = (1,0,0,1,1) then
       integer_value := 19;
    elsif temp(0..4) = (1,0,1,0,0) then
       integer_value := 20;
    elsif temp(0..4) = (1,0,1,0,1) then
       integer_value := 21;
    elsif temp(0..4) = (1,0,1,1,0) then
       integer_value := 22;
    elsif temp(0..4) = (1,0,1,1,1) then
       integer_value := 23;
    elsif temp(0..4) = (1,1,0,0,0) then
       integer_value := 24;
    elsif temp(0..4) = (1,1,0,0,1) then
       integer_value := 25;
    elsif temp(0..4) = (1,1,0,1,0) then
       integer_value := 26;
    elsif temp(0..4) = (1,1,0,1,1) then
       integer_value := 27;
    elsif temp(0..4) = (1,1,1,0,0) then
       integer_value := 28;
    elsif temp(0..4) = (1,1,1,0,1) then
       integer_value := 29;
    elsif temp(0..4) = (1,1,1,1,0) then
       integer_value := 30;
    elsif temp(0..4) = (1,1,1,1,1) then
       integer_value := 31;
  end if;
  return integer_value;


If TAG3 /= 3 then
   HANDLER;
else
  TAG3 := 0;
End if;

end CONVERT_1;
```

```
------- PROCEDURE BUS_CONTROLLER_1 --------
--
--
-- This procedure is called only when the terminal is configured as a
-- bus controller.  It is this procedure that initiates an information
-- transfer by sending out the command word passed to it as a parameter.
-- Once the command word has been placed on the bus via the PLI_1.GET_DATA
-- call, the procedure then determines what type of information transfer
-- is to take place and calls the appropriate procedure.  The procedure
-- CHECK_STATUS is used by the other two procedures listed in the
-- declarative section and is made to both by declaring it one level
-- higher.
--
--
-- Called by·          BIU_1
-- Procedures called :  DATA_TRANSFER
--                      MODE_COMMAND
-- Tasks accessed :      PLI_1
--


procedure bus_controller_1( command  : command_type;
                            command2 : command_type;
                            mode     : out boolean) is

TAG4 : integer := 0;
synch : constant := 1;
temp : command_type;
```

```
-------- PROCEDURE CHECK_STATUS ------------
--
--
-- CHECK_STATUS : Procedure to process a status word received
--              by the BUS CONTROLLER.  This procedure is called
--              by the procedures MODE_COMMAND and DATA_TRANSFER
--              when a status word is expected.  If no status
--              word arrives via the PLI_1.GET_DATA request
--              within a specified period of time, a timeout
--              occurs and an error message is sent out.  When
--              a status word does arrive, bits are examined for
--              any indication of an error having occurred during
--              a mode command or a data transfer.  Any sign of
--              error causes an error flag to be set and an
--              error message to be printed out.
--
--
-- Called by :       MODE_COMMAND
--                 DATA_TRANSFER
-- Procedures called: NONE
-- Tasks accessed :    PLI_1
--


procedure check_status(command : in command_type; mode : out boolean) is

TAG5 : integer := 0;
busy : boolean := false;
error : boolean := false;
status : command_type;
synch : integer;
element : integer;

begin
 If TAG5 = 0 then
   TAG5 := 5;
 else
   HANDLER;
 End if;

 WDT1 := 255;
 mode := true;
 if command(0..4) /= (1,1,1,1,1) then
   select
     PLI_1.SEND_DATA(status,synch);
     NEW_LINE;
     PUT_LINE(" ---- CHECKING STATUS WORD -------- ");
     NEW_LINE;
     PUT("STATUS WORD RECEIVED FROM REMOTE TERMINAL :    ");
     for element in 0..15
      loop


      If TAG5 /= 5 then
```

```
        HANDLER;
       End if;
        PUT(status(element),1);
       end loop;
      NEW_LINE;
      NEW_LINE;
      if synch = 1 then
       if status(0..4) /= command(0..4) then
         error := true;
         PUT_LINE("ERROR IN STATUS WORD :  INCORRECT ADDRESS");
        end if;
    if status(5) = 1 then
      error := true;
      PUT_LINE("MESSAGE ERROR BIT HAS BEEN SET");
    end if;
    if ((status(7) = 1) or (status(11) = 1)) then
       error := true;
    end if;

    if ((status(15) = 1 ) or (status(13) = 1)) then
      error := true;
    end if;
    if status(12) = 1 then
      busy := true;
      PUT_LINE("  TERMINAL IS BUSY");
    end if;

   if (not busy) and (not error)  and (status(14) = 1) then
      mode := false;
   end if;
else
   PUT_LINE(" ERROR IN SYNCH BIT : STATUS WORD EXPECTED");
end if;

or
  delay 15.0;
  PUT_LINE("TIME_OUT : WAITING FOR STATUS WORD");
end select;
end if;


If TAG5 /= 5 then
   HANDLER;
else
  TAG5 := 0;
End if;
end CHECK_STATUS;
```

```
procedure mode_command(command : in command_type;mode : out boolean) is


TAG6 : integer := 0;
-- This procedure is called by BUS_CONTROLLER_1 when it has been
-- determined that the type of information transfer is that of a mode
-- command. The mode commands may or may not have an associated data
-- word. If there is no associated data word, CHECK_STATUS is called
-- to check the validity of the returned status word. If a data word is
-- associated with the mode command, a data word is either sent or
-- received along with the status word of the accessed remote terminal.
--
--
-- Called  By :      BUS_CONTROLLER_1
-- Procedures called : CHECK_STATUS
-- Tasks accessed :     PLI_1


data : command_type;
synch : integer := 0;
status : command_type;
element : integer;

begin

 If TAG6 = 0 then
   TAG6 := 6;
 else
   HANDLER;
 End if;

 WDT1 := 255;
  if command(11) = 0 then
     CHECK_STATUS(command,mode);
     WDT1 := 255;
  else
     if command(11..15) = (1,0,0,0,1) then
        data := MODE_DATA_REGISTER;
        PLI_1.GET_DATA(data,synch);
        CHECK_STATUS(command,mode);
        WDT1 := 255;
        PUT("Data word sent to Remote Terminal :    ");
        for element in 0..15  loop


        If TAG6 /= 6 then
          HANDLER;
        End if;

         PUT(data(element),1);
        end loop;
        NEW_LINE;
     else
        CHECK_STATUS(command,mode);
        WDT1 := 255;
```

```
        PLI_1.SEND_DATA(data,synch);
        if command (11..15) = (1,0,0,1,1) then
           BUILT_IN_TEST_REGISTER := data;
        else
           MODE_DATA_REGISTER := data;
           INTERNAL_STATUS_REGISTER (6) := 1;
        end if;
        if synch = 1 then
           PUT_LINE("SYNCH BIT ERROR : DATA WORD EXPECTED");
        end if;
        PUT("Data word received from Remote Terminal :  ");
        for element in 0..15
         loop


         If TAG6 /= 6 then
           HANDLER;
         End if;

          PUT(data(element),1);
         end loop;
        NEW_LINE;
      end if;
    end if;


    If TAG6 /= 6 then
      HANDLER;
    else
     TAG6 := 0;
    End if;

end MODE_COMMAND;
```

```
procedure data_transfer(command, command2 : in command_type) is


TAG7 : integer := 0;
-- DATA_TRANSFER : This procedure is called when it has been
--                 determined that the command word sent out
--                 indicates that a data transfer is to take place.
--                 Three types of valid data transfers have been
--                 defined :
--                         RT TO BC
--                         BC TO RT
--                         RT TO RT
--                 The type of data transfer is determined by
--                 the command word.  Once a particular type has
--                 been selected, a procedure corresponding to
--                 the type of data transfer is called to process
--                 the data transfer.
-- Called by :          BUS_CONTROLLER_1
-- Procedures called :  PROCESS_RT_TO_BC
--                      PROCESS_RT_TO_RT
--                      PROCESS_BC_TO_RT
-- Tasks accessed :     PLI_1
--
--


  type transfer_type is (RT_TO_BC, BC_TO_RT, RT_TO_RT, ABORT_TRANSFER);
  data_count : integer;
  data : command_type;
  synch : integer;
  transfer : transfer_type;
  ram_address : integer;
```

```
procedure process_rt_to_bc(data_count,address : in integer) is

TAG8 : integer := 0;
 -- This procedure stores the data words received from a remote terminal in
 -- the buffer array.  If the synch bit value associated with each data word
 -- is a 1, an error message is displayed, but the data word is still stored.
 -- If there is too long a delay during a data word has been received, the
 -- procedure selects the delay option and exits the loop.

 ram_address : integer;
 data : command_type;
 synch : integer;
 element : integer;
 count : integer;

 begin
  If TAG8 = 0 then
    TAG8 := 8;
  else
    HANDLER;
  End if;

  WDT1 := 255;
  count := data_count;
  ram_address := address;
  PUT(data_count,1);
  PUT_LINE("  DATA WORDS RECEIVED FROM REMOTE TERMINAL :  ");
  NEW_LINE;
  loop


  If TAG8 /= 8 then
    HANDLER;
  End if;

   exit when count = 0;
   select
      PLI_1.SEND_DATA(data,synch);
      PUT("WORD  ");
      PUT(data_count - count,1);
      PUT("  ");
      for element in 0..15
       loop
         If TAG8 /= 8 then
           HANDLER;
         End if;
         PUT(data(element),1);
       end loop;
      NEW_LINE;
      DMA_1.STORE(data,ram_address);
      ram_address := ram_address + 1;
      if synch = 0 then
        PUT_LINE("ERROR IN DATA TRANSMISSION RT TO BC");
      end if;
      count := count - 1;
  or
```

```
      delay 25.0;
      PUT_LINE(" TIME OUT");
     exit;
   end select;
end loop;


If TAG8 /= 8 then
   HANDLER;
else
  TAG8 := 0;
End if;

end PROCESS_RT_TO_BC;
```

```
procedure process_rt_to_rt(data_count : in integer) is

TAG9 : integer := 0;
-- This procedure monitors the transfer of data between two terminals set up as
-- as remote terminals.Since each terminal is set up to exit the data transfer
-- loop after a designated time delay after a data word transmission, this
-- procedure also uses the same time delay to exit the monitoring loop.

data : command_type;
synch : integer;
count : integer;


begin

 If TAG9 = 0 then
   TAG9 := 9;
 else
   HANDLER;
 End if;

 WDT1 := 255;
  count := data_count;
  loop

    If TAG9 /= 9 then
       HANDLER;
    End if;
    exit when count = 0;
    select
      PLI_1.SEND_DATA(data,synch);
     or
       delay 25.0;
     exit;
  end select;
   count := count - 1;
  end loop;


If TAG9 /= 9 then
   HANDLER;
else
  TAG9 := 0;
End if;

end PROCESS_RT_TO_RT;
```

```
procedure process_bc_to_rt(data_count,address : in integer) is

TAG10 : integer := 0;
-- This procedure sends the required number of data words to the bus. The
-- words are sent out continuously without any time delay.

data : command_type;
count : integer;
synch : integer := 0;
ram_address : integer;

--
begin
 If TAG10 = 0 then
   TAG10 := 10;
 else
   HANDLER;
 End if;

 WDT1 := 255;
  count := data_count;
  ram_address := address;
  loop

    If TAG10 /= 10 then
      HANDLER;
    End if;
    exit when count = 0;
    DMA_1.WORD(data,ram_address);
    ram_address := ram_address + 1;
    PLI_1.GET_DATA(data,synch);
    count := count - 1;
  end loop;


If TAG10 /= 10 then
  HANDLER;
else
  TAG10 := 0;
End if;
end PROCESS_BC_TO_RT;
```

```
  begin -- DATA TRANSFER --

If TAG7 = 0 then
  TAG7 := 7;
else
  HANDLER;
End if;

WDT1 := 255;
   data := command;
   temp(0..4) := command(11..15);
   data_count := CONVERT_1(temp);
   WDT1 := 255;
   temp(0..4) := command(6..10);
   ram_address := CONVERT_1(temp);
   WDT1 := 255;
   if data_count = 0 then
     data_count := 32;
   end if;
   if command(5) = 1 then
      transfer := RT_TO_BC;
      PUT_LINE("COMMAND TYPE :  DATA TRANSFER - RT TO BC");
      NEW_LINE;
   else
      data := command2;
      if Processor_Control_Register(0..4) /= command2(0..4) then
        synch := 1;
      else
        synch := 0;
      end if;
      PLI_1.GET_DATA(command2,synch);
      for element in 0..15
      loop


      If TAG7 /= 7 then
        HANDLER;
      End if;

       PUT(command2(element),0);
      end loop;
      if synch = 1 then
        transfer := RT_TO_RT;
        PUT_LINE("COMMAND TYPE : DATA TRANSFER - RT TO RT");
        NEW_LINE;
      else
        transfer := BC_TO_RT;
        PUT_LINE("COMMAND TYPE : DATA TRANSFER - BC TO RT");
        NEW_LINE;
        data_count := data_count -1;
      end if;
   end if;

   case transfer is
      when RT_TO_BC = > CHECK_STATUS(command,mode);
                  WDT1 := 255;
```

```
                PROCESS_RT_TO_BC(data_count,ram_address);
                WDT1 := 255;
        when RT_TO_RT  => CHECK_STATUS(data,mode);
                WDT1 := 255;
                PROCESS_RT_TO_RT(data_count);
                WDT1 := 255;
                CHECK_STATUS(command,mode);
                WDT1 := 255;
        when BC_TO_RT => PROCESS_BC_TO_RT(data_count,ram_address);
                WDT1 := 255;
                CHECK_STATUS(command,mode);
                WDT1 := 255;
        when ABORT_TRANSFER => NULL;
    end case;

If TAG7 /= 7 then
  HANDLER;
else
  TAG7 := 0;
End if;
end DATA_TRANSFER;
```

```
-- procedure BUS CONTROLLER --
--
begin

    If TAG4 = 0 then
      TAG4 := 4;
    else
      HANDLER;
    End if;

    WDT1 := 255;
    PLI_1.GET_DATA(command,synch);
    PUT("COMMAND WORD SENT FROM BUS CONTROLLER :    ");

    for element in 0..15 loop


      If TAG4 /= 4 then
        HANDLER;
      End if;
      PUT(command(element),1);
    end loop;
    NEW_LINE;
    NEW_LINE;

    if (command(6..10) = (0,0,0,0,0) or command(6..10) = (1,1,1,1,1))
      then
        MODE_COMMAND(command,mode);
        WDT1 := 255;

      else
        DATA_TRANSFER(command, command2);
        WDT1 := 255;
      end if;



      If TAG4 /= 4 then
        HANDLER;
      else
        TAG4 := 0;
      End if;

end BUS_CONTROLLER_1;
```

```
procedure remote_terminal_1(mode : out boolean) is

TAG11 : integer := 0;
-- This procedure is called only when the calling task BIU_1 is
-- configured as a remote terminal. The procedure receives a command word from
-- its associated BIU and determines whether a MODE COMMAND or a
-- DATA TRANSFER is to take place.

me_bit : constant := 5;
broadcast : constant := 11;
busy_bit : constant := 12;
sf_bit   : constant := 13;
tf_bit   : constant := 14;
temp : command_type;
synch : integer;
busy, s_failure,t_failure : boolean := FALSE;
element : integer;
command : command_type;
```

```
procedure process_mode_command(command : in command_type;mode : out boolean) is


TAG12 : integer := 0;
 -- This procedure is called when is determined that a mode command has
 -- been placed on the bus. The command_mode variable defines the type of
 -- mode command that is to be processed. If a broadcast command is attempted
 -- with command modes that require a data word to be sent back to the bus
 -- controller, the command_mode is assigned invalid and an error message is
 -- displayed.

 DBC_bit : constant := 14;
 TF_bit : constant := 15;
 data : command_type;
 synch : integer;
 status : command_type;
 ram_address : integer := 0;
 type mode_command is (invalid, dynamic_bus_control, synchronize,
       transmit_status_word, initiate_self_test, inhibit_flag_bit,
       override_itfb, reset_rt, transmit_command, synch_with_data,
       transmit_vector, transmit_bit);
 command_mode : mode_command;


 begin

 If TAG12 = 0 then
   TAG12 := 12;
 else
   HANDLER;
 End if;

 WDT1 := 255;
   mode := true;
  if command(11..15) = (0,0,0,0,0) then
     command_mode := DYNAMIC_BUS_CONTROL;
     PUT_LINE("Mode Code : 00000  => Dynamic Bus Control");
     if Processor_Control_Register(12) = 1 then
       mode := false;
       status_word(DBC_bit) := 1;
       PUT_LINE("Bus Control Accepted");
     else
       PUT_LINE("Bus Control Rejected");
     end if;

  elsif command(11..15) = (0,0,0,0,1) then
     command_mode := SYNCHRONIZE;
     PUT_LINE("Mode Code : 00001  => Synchronize");
  elsif command(11..15) = (0,0,0,1,0) then
     command_mode := TRANSMIT_STATUS_WORD;
     PUT_LINE("Mode Code : 00010  => Transmit Status Word");
  elsif command(11..15) = (0,0,0,1,1) then
     command_mode := INITIATE_SELF_TEST;
     PUT_LINE("Mode Code : 00011  => Initiate Self-Test");
  elsif command(11..15) = (0,0,1,1,0) then
     command_mode := INHIBIT_FLAG_BIT;
```

```
    PUT_LINE("Mode Code : 00110  => Inhibit Flag Bit");
  elsif command(11..15) = (0,0,1,1,1) then
    command_mode := OVERRIDE_ITFB;
    PUT_LINE("Mode Code : 00111  => Override Inhibit Flag Bit");
  elsif command(11..15) = (0,1,0,0,0) then
    command_mode := RESET_RT;
    PUT_LINE("Mode Code : 01000  => Reset Remote Terminal");
  elsif command(11..15) = (1,0,0,0,0) then
    command_mode := TRANSMIT_VECTOR;
    PUT_LINE("Mode Code : 10000  => Transmit Vector Word");
  elsif command(11..15) = (1,0,0,0,1) then
    command_mode := SYNCH_WITH_DATA;
    PUT_LINE("Mode Code : 10001  => Synchronize with Data");
  elsif command(11..15) = (1,0,0,1,0) then
    command_mode := TRANSMIT_COMMAND;
    PUT_LINE("Mode Code : 10010  => Transmit Last Command");
  elsif command(11..15) = (1,0,0,1,1) then
    command_mode := TRANSMIT_BIT;
    PUT_LINE("Mode Code : 10011  => Transmit BIT Word");
  end if;


-- The following code determines if the mode command was sent
-- in broadcast mode and is incorrectly associated with a data
-- word to be received by the bus controller.  If so, the
-- command_mode is assigned invalid.

if (command(0..4) = (1,1,1,1,1)) and ((command(11..15) = (0,0,0,0,0)) or
 (command(11..15) = (0,0,0,1,0)) or (command(11..15) = (1,0,0,0,0)) or
 (command(11..15) = (1,0,0,1,0)) or (command(11..15) = (1,0,0,1,1))) then
    command_mode := invalid;
end if;

synch := 1;
case command_mode is
  when DYNAMIC_BUS_CONTROL   => data := status_word;
  when SYNCHRONIZE           => data := status_word;
  when TRANSMIT_STATUS_WORD  => data := last_status_word;
  when INITIATE_SELF_TEST    => data := status_word;
  when INHIBIT_FLAG_BIT      => data := status_word;
                        data(TF_bit) := 0;
  when OVERRIDE_ITFB         => data := status_word;
  when RESET_RT              => data := status_word;
  when TRANSMIT_COMMAND      => data := status_word;
                        PLI_1.GET_DATA(data,synch);
                        data := last_command_word;

                        synch := 0;
  when SYNCH_WITH_DATA       => PLI_1.SEND_DATA(data,synch);
                        DMA_1.STORE(data,ram_address);
                        data := status_word;
                        synch := 1;
  when TRANSMIT_VECTOR       => data := status_word;
                        PLI_1.GET_DATA(data,synch);
                        DMA_1.WORD(data,ram_address);
                        synch := 0;
```

```
 when TRANSMIT_BIT        => data := status_word;
                       PLI_1.GET_DATA(data,synch);
 when INVALID             =>  PUT_LINE(" INVALID MODE REQUEST");
end case;
if command(0..4) /= (1,1,1,1,1) then
 PLI_1.GET_DATA(data,synch);
end if;


If TAG12 /= 12 then
  HANDLER;
else
 TAG12 := 0;
End if;

end PROCESS_MODE_COMMAND;
```

```
procedure process_data_transfer(command : in command_type) is

TAG13 : integer := 0;
-- This procedure is called by the procedure REMOTE_TERMINAL
-- when it is determined that the command is a data transfer
-- type.  This procedure, in turn, calls two procedures depending
-- on the whether is to transmit data or receive data.
--          RECEIVE_DATA
--          SEND_DATA


type transfer_type is (RT_TO_RT,RT_TO_BC,BC_TO_RT,ABORT_TRANSFER);
status : command_type;
data : command_type;
transfer : transfer_type;
data_count : integer;
synch : integer;
ME_bit : constant := 5;
element : integer;
ram_address : integer;
```

```
procedure send_data(data_count,address : in integer) is

TAG14 : integer := 0;
  count : integer;
  data : command_type;
  synch : integer;
  ram_address : integer;

begin

 If TAG14 = 0 then
    TAG14 := 14;
 else
    HANDLER;
 End if;

 WDT1 := 255;
  count := data_count;
  ram_address := address;
  loop


  If TAG14 /= 14 then
     HANDLER;
  End if;

    DMA_1.WORD(data,ram_address);
    ram_address := ram_address + 1;
    PLI_1.GET_DATA(data,synch);
    count := count - 1;
    exit when count = 0;
  end loop;


If TAG14 /= 14 then
   HANDLER;
else
  TAG14 := 0;
End if;

end SEND_DATA;
```

```
procedure receive_data(data_count,address: in integer) is

TAG15 : integer := 0;
  count : integer;
  element : integer;
  data : command_type;
  ME_bit : constant := 5;
  synch : integer;
  ram_address : integer;

begin

 If TAG15 = 0 then
    TAG15 := 15;
 else
    HANDLER;
 End if;

 WDT1 := 255;
  count := data_count;
  ram_address := address;
  loop


  If TAG15 /= 15 then
     HANDLER;
  End if;

   select
      PLI_1.SEND_DATA(data,synch);
      DMA_1.STORE(data,ram_address);
      ram_address := ram_address + 1;
      PUT("WORD ");
      PUT(data_count - count,1);
      PUT("     ");
      for element in 0..15
       loop


       If TAG15 /= 15 then
          HANDLER;
       End if;

         PUT(data(element),1);
       end loop;
      NEW_LINE;
      if synch = 1 then
       status_word(ME_bit) := 1;
      end if;
      count := count - 1;
      exit when count = 0;
    or
      delay 10.0;
      PUT_LINE("TIME OUT IN RECEIVING DATA");
      exit;
    end select;
```

```
   end loop;

If TAG15 /= 15 then
   HANDLER;
else
  TAG15 := 0;
End if;

end RECEIVE_DATA;
```

```
      --- procedure PROCESS DATA TRANSFER ---
begin
 If TAG13 = 0 then
   TAG13 := 13;
 else
   HANDLER;
 End if;

 WDT1 := 255;
 temp(0..4) := command(11..15);
 data_count := CONVERT_1(temp);
 WDT1 := 255;
 if data_count = 0 then
    data_count := 32;
 end if;
 temp(0..4) := command(6..10);
 ram_address := CONVERT_1(temp);
 WDT1 := 255;

if command(5) = 1 then
   if command(0..4) = (1,1,1,1,1) then
      transfer := abort_transfer;
      status_word(ME_bit) := 1;
   else
      transfer := RT_TO_BC;
   end if;

else
   PLI_1.SEND_DATA(data,synch);
   if synch = 1 then
     if (data(5) = 0) or (command(0..4) = data(0..4)) then
       status_word(ME_bit) := 1;
     else
       transfer := RT_TO_RT;
       PUT(data_count,1);
       PUT(" WORDS SENT TO REMOTE TERMINAL");
       NEW_LINE;
     end if;
   else
     transfer := BC_TO_RT;
     PUT(data_count,1);
     PUT_LINE("   WORDS SENT TO REMOTE TERMINAL");
     NEW_LINE;
     PUT("WORD 0    ");
     for element in 0..15
      loop


      If TAG13 /= 13 then
        HANDLER;
      End if;

       PUT(data(element),1);
      end loop;
```

```
    NEW_LINE;
    DMA_1.STORE(data,ram_address);
    ram_address := ram_address + 1;
    data_count := data_count - 1;
  end if;
end if;

synch := 1;
case transfer is
 when RT_TO_BC    => PLI_1.GET_DATA(status_word,synch);
                     SEND_DATA(data_count,ram_address);
                     WDT1 := 255;
 when  RT_TO_RT   => PLI_1.SEND_DATA(data,synch);
                     RECEIVE_DATA(data_count,ram_address);
                     WDT1 := 255;
                     PLI_1.GET_DATA(status_word,synch);
 when BC_TO_RT    => RECEIVE_DATA(data_count,ram_address);
                     WDT1 := 255;
                     PLI_1.GET_DATA(status_word,synch);
 when ABORT_TRANSFER => NULL;
end case;


If TAG13 /= 13 then
  HANDLER;
else
 TAG13 := 0;
End if;

end PROCESS_DATA_TRANSFER;
```

```
    --- procedure REMOTE TERMINAL ---

begin

If TAG11 = 0 then
  TAG11 := 11;
else
  HANDLER;
End if;

WDT1 := 255;
    mode := true;
    PLI_1.SEND_DATA(command,synch);
    if ((synch = 1) and (command(0..4) = (0,0,1,1,0) or
        command(0..4) = (1,1,1,1,1))) then
      PUT_LINE("BIU_1 IS A REMOTE TERMINAL");
      if busy or s_failure or t_failure then
       if busy then
         status_word(busy_bit) := 1;
       end if;

       if s_failure then
         status_word(sf_bit) := 1;
       end if;

       if t_failure then
         status_word(tf_bit) := 1;
       end if;
      else
       if command(0..4) = (1,1,1,1,1) then
         status_word(broadcast) := 1;
       end if;
       if command(6..10) = (0,0,0,0,0) or command(6..10) = (1,1,1,1,1) then
          PROCESS_MODE_COMMAND(command,mode);
          WDT1 := 255;
       else
          PROCESS_DATA_TRANSFER(command);
          WDT1 := 255;
       end if;
      end if;
      last_command_word := command;
      last_status_word := status_word;
    else
      status_word(me_bit) := 1;
    end if;


If TAG11 /= 11 then
  HANDLER;
else
 TAG11 := 0;
End if;

end REMOTE_TERMINAL_1;
```

```
begin  -- BIU_1 --

 If TAG1 = 0 then
   TAG1 := 1;
 else
   HANDLER;
 End if;

 WDT1 := 255;
accept START;
if Processor_Control_Register(10) = 1 then
    bus_controller_mode := true;
    remote_terminal_mode := false;
else
    bus_controller_mode := false;
    remote_terminal_mode := true;
end if;

COMMUNICATION_MODE_LOOP :
 loop


 If TAG1 /= 1 then
   HANDLER;
 End if;

  select
    accept go;
    if bus_controller_mode then
       NEW_LINE;
       NEW_LINE;
       PUT_LINE("BIU_1 IS BUS CONTROLLER");
       NEW_LINE;
       FORM_COMMAND(command,command2,halt);
       WDT1 := 255;
       Base_Address_Register(10..15) := command(5) & command(6..10);
       exit when halt;
       BUS_CONTROLLER_1(command,command2,mode);
       WDT1 := 255;
       bus_controller_mode := mode;
       remote_terminal_mode := not mode;

    elsif remote_terminal_mode then
       Base_Address_Register(10..15) := command(5) & command(6..10);
       REMOTE_TERMINAL_1(mode);
       WDT1 := 255;
       remote_terminal_mode := mode;
       bus_controller_mode := not mode;
       status_word := (0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0);
    end if;
  or
     accept done;
     exit COMMUNICATION_MODE_LOOP;
  end select;
 end loop COMMUNICATION_MODE_LOOP;
```

```
 APPLICATION_PROCESSOR_1.BIU_DONE;

If TAG1 /= 1 then
   HANDLER;
else
  TAG1 := 0;
End if;

end BIU_1;
```

```
with TEXT_IO;use TEXT_IO;
package DATA_BUS is


  type command_type is array(0..15) of integer;


  task PLI_1 is
    entry GET_DATA(in_data : in command_type; in_synch : in integer);
    entry SEND_DATA(out_data : out command_type; out_synch : out integer);
    entry GET_DATA_FROM_BUS(bus_data : in command_type; bus_synch : in integer);
    entry DONE;
  end PLI_1;

  task PLI_2 is
    entry GET_DATA(in_data : in command_type; in_synch : in integer);
    entry SEND_DATA(out_data : out command_type; out_synch : out integer);
    entry GET_DATA_FROM_BUS(bus_data : in command_type; bus_synch : in integer);
    entry DONE;
  end PLI_2;

  task PLI_3 is
    entry GET_DATA(in_data : in command_type; in_synch : in integer);
    entry SEND_DATA(out_data : out command_type; out_synch : out integer);
    entry GET_DATA_FROM_BUS(bus_data : in command_type; bus_synch : in integer);
    entry DONE;
  end PLI_3;

  task BUS is
    entry GET_DATA(in_data : in command_type;in_synch : in integer);
    entry DONE;
  end BUS;
end DATA_BUS;
```

```
with TEXT_IO;use TEXT_IO;
package body DATA_BUS is

task body BUS is

data : command_type;
synch : integer;

begin
BUS_TRANSFER :
loop
  select
    accept GET_DATA(in_data : in command_type;in_synch : in integer) do
        data := in_data;
        synch := in_synch;
    end GET_DATA;
  select
    PLI_1.GET_DATA_FROM_BUS(data,synch);
  or
    delay 5.0;
  end select;
 select
    PLI_2.GET_DATA_FROM_BUS(data,synch);
 or
  delay 5.0;
end select;

select
    PLI_3.GET_DATA_FROM_BUS(data,synch);
or
 delay 5.0;
end  select;
or
  accept done;
  exit;
end select;
end loop BUS_TRANSFER;
end BUS;
```

```
task body PLI_1  is

data : command_type;
synch : integer;

begin

loop
  select

     accept GET_DATA(in_data : in command_type; in_synch : in integer) do
        data := in_data;
        synch := in_synch;
     end GET_DATA;
select
     BUS.GET_DATA(data,synch);
or
delay 5.0;
end select;
select
     accept GET_DATA_FROM_BUS(bus_data : in command_type;bus_synch : in integer)
 do
        data := bus_data;
        synch := bus_synch;
     end GET_DATA_FROM_BUS;
or
  delay 5.0;
end select;

  or

     accept GET_DATA_FROM_BUS(bus_data : in command_type; bus_synch : in integer
) do
        data := bus_data;
        synch := bus_synch;
     end GET_DATA_FROM_BUS;
    select
     accept SEND_DATA(out_data : out command_type; out_synch : out integer) do
        out_data := data;
        out_synch := synch;
     end SEND_DATA;
  or
    delay 5.0;
  end select;

 or


      accept done;
     exit;

 end select;
end loop;
end PLI_1;
```

```
task body biu2  is

data : command_type;
synch : integer;

begin

loop
  select

     accept GET_DATA(in_data : in command_type; in_synch : in integer) do
        data := in_data;
        synch := in_synch;
     end GET_DATA;
select
     BUS.GET_DATA(data,synch);
or
delay 5.0;
end select;
select
     accept GET_DATA_FROM_BUS(bus_data : in command_type;bus_synch : in integer)
 do
        data := bus_data;
        synch := bus_synch;
     end GET_DATA_FROM_BUS;
or
  delay 5.0;
end select;

  or

     accept GET_DATA_FROM_BUS(bus_data : in command_type; bus_synch : in integer
) do
        data := bus_data;
        synch := bus_synch;
     end GET_DATA_FROM_BUS;
    select
     accept SEND_DATA(out_data : out command_type; out_synch : out integer) do
        out_data := data;
        out_synch := synch;
     end SEND_DATA;
  or
    delay 5.0;
  end select;

 or


    accept done;
     exit;

 end select;
end loop;
end biu2;
```

```
task body biu3  is

data : command_type;
synch : integer;

begin

loop
  select

     accept GET_DATA(in_data : in command_type; in_synch : in integer) do
        data := in_data;
        synch := in_synch;
     end GET_DATA;
select
     BUS.GET_DATA(data,synch);
or
delay 5.0;
end select;
select
     accept GET_DATA_FROM_BUS(bus_data : in command_type;bus_synch : in integer)
 do
        data := bus_data;
        synch := bus_synch;
     end GET_DATA_FROM_BUS;
or
  delay 5.0;
end select;

  or

     accept GET_DATA_FROM_BUS(bus_data : in command_type; bus_synch : in integer
) do
        data := bus_data;
        synch := bus_synch;
     end GET_DATA_FROM_BUS;
    select
     accept SEND_DATA(out_data : out command_type; out_synch : out integer) do
        out_data := data;
        out_synch := synch;
     end SEND_DATA;
  or
    delay 5.0;
  end select;

 or


      accept done;
     exit;

 end select;
end loop;
end biu3;

end DATA_BUS;
```

Appendix B
Software Modifier for Upset Detection (SMUD)

This appendix contains the Ada code that implements SMUD.

```
with TEXT_IO; use TEXT_IO;
with INTEGER_TEXT_IO; use INTEGER_TEXT_IO;
with FLOAT_TEXT_IO; use FLOAT_TEXT_IO;
procedure SMUD is
-- The purpose of this program is to modify input files to include two
-- upset detection mechanisms: block-checking constructs and a watchdog
-- timer mechanism. This program can handle software systems that will
-- run on several processors ( as in distributed systems ) or in a
-- uni-processor based system. The modification process is done in
-- two passes (PASS_ONE and PASS_TWO). PASS_ONE parses each file to
-- determine the type of upset detection mechanism to insert, while
-- PASS_TWO modifies the file according to the information obtained
-- in PASS_ONE. Each program unit is assigned a record structure
-- that contains detailed information such as the program unit's ID
-- value and the type of upset detection mechanisms to incorporate.



  max_length : constant := 120;
  node : integer;
  number_of_nodes : integer;
  current_watchdog : integer;
  watchdogs : integer;
  unit_level : integer;
  n : array(1..10) of integer;
  nest : integer;
  save : array(1..20) of integer;
  in_save : array(1..20) of integer;
  nesting : integer;
  file : integer;
  done : boolean := false;
  program_units : integer;
  ada_file : string (1..max_length);
  ada_out : string (1..max_length);
  length : integer;
  length_1 : integer;
  file_num : integer;
  input_data : file_type;
  output_data : file_type;
  count : integer;

  type files is
    record
      name : string (1..max_length);
      len : integer;
    end record;

  file_list : array (1..20) of files;

  type check is (place, show,omit);
  type program_unit is
    record
      kind : string (1..9);
      name : string (1..max_length);
      len : integer;
```

```
  id : integer := 0;
  number_statements : integer := 0;
  watchdog : integer := 0;
  block : check;
  WDT : check;
  tag_name : string(1..4);
end record;

unit : array(1..50) of program_unit;
```

```
procedure CONVERT (line : in out string;
                   length : in integer) is separate;

procedure FORM_PACKAGE(watchdogs : in integer;node : integer) is separate;

procedure OUTPUT( program_units : in integer;
                  watchdogs : in integer;node : integer) is separate;

procedure PASS_ONE(input_data,
                   output_data : in file_type;
                   program_units : in out integer;
                   unit_level : in out integer;
                   watchdogs : in out integer) is separate;

procedure PASS_TWO(input_data,
                   output_data : in file_type;
                   program_units : in integer;
                   unit_level : in out integer;
                   file : in out integer) is separate;

procedure GET_FILE(done : in out boolean;
                   ada_file : in out string;
                   length : in out integer) is

-- The purpose of this procedure is to open each file entered as input
-- to SMUD.  If an entered filename does not exist on the current
-- directory, the exception NAME_ERROR is raised and the user is
-- prompted for a correct or existing filename.

input_data : file_type;

begin

  done := false;
  PUT_LINE("Enter file name :");
  GET_LINE(ada_file,length);
  if ada_file(1..4) = "done" then
    done := true;
  else
    OPEN(input_data,in_file,ada_file(1..length));
  end if;

  if not done then
    CLOSE(input_data);
  end if;

  exception
    when NAME_ERROR => PUT_LINE(" file cannot be found");
                       GET_FILE(done,ada_file,length);
end;
```

```
--- SMUD ----

begin

PUT_LINE("Enter the number of nodes in the system :");
GET(number_of_nodes);
NEW_LINE;
for node in 1..number_of_nodes loop
  unit_level := 0;
  nesting := 0;
  watchdogs := 1;
  current_watchdog := 1;
  nest := 1;
  file := 0;
  count := 0;
  program_units := 1;
  loop
    count := count + 1;
    GET_FILE(done,ada_file,length);
    file_list(count).name(1..length) := ada_file(1..length);
    file_list(count).len := length;
    exit when done;
  end loop;

  for file_num in 1..(count-1) loop
    length := file_list(file_num).len;
    ada_file(1..length) := file_list(file_num).name(1..length);
    OPEN(input_data,in_file,ada_file(1..length));
    PASS_ONE(input_data,output_data,program_units,unit_level,watchdogs);
    CLOSE(input_data);
  end loop;


  unit_level := 0;
  nesting := 0;
  FORM_PACKAGE(watchdogs,node);
  OUTPUT(program_units,watchdogs,node);

  for file_num in 1..(count-1) loop
    length := file_list(file_num).len;
    ada_file(1..length) := file_list(file_num).name(1..length);
    OPEN(input_data,in_file,ada_file(1..length));
    length_1 := length + 3;
    ada_out(1..length_1) := "UDM" & ada_file(1..length);
    CREATE(output_data,out_file,ada_out(1..length_1));
    PASS_TWO(input_data,output_data,program_units,unit_level,file);
    CLOSE(input_data);
    CLOSE(output_data);
  end loop;
end loop;


end SMUD;
```

```
separate (SMUD)
procedure FORM_PACKAGE( watchdogs : in integer;node : integer) is

-- This procedure is called after PASS_ONE the package UPSET_DETECTION
-- which contains the declarations for each watchdog timer required.
-- The package UPSET_DETECTION also contains the procedure which raises
-- the user defined exception INCORRECT_TAG and the exception handler.


count : integer;
len : integer;
package_out : file_type;
address : integer;
base : integer;
file : file_type;

begin
  create(file,out_file,"upset.ada");
  PUT_LINE(file,"with system, use system");
  PUT_LINE(file,"package UPSET_DETECTION is");
  NEW_LINE(file);
  PUT_LINE(file,"bits : constant := 1");
  PUT_LINE(file,"type watchdog_timer is range 0..255");
  PUT_LINE(file,"for watchdog_timer'SIZE use 8*bits");
  NEW_LINE(file);
  for count in 1..watchdogs loop
     PUT_LINE("Enter base and address of next watchdog timer:");
     PUT_LINE("   base:");
     GET(base);
     PUT_LINE("   address:");
     GET(address);
     PUT(file,"  WDT");
     PUT(file,count,0);
     PUT_LINE(file," : watchdog_timer;");
     PUT(file,"  for WDT");
     PUT(file,count,0);
     PUT(file," use at ");
     PUT(file,base,0);
     PUT(file,"#");
     PUT(file,address,0);
     PUT_LINE(file,"#;");
     NEW_LINE(file);
 end loop;

     PUT_LINE(file,"procedure HANDLER;");
     PUT_LINE(file,"end UPSET_DETECTION");

     PUT_LINE(file,"package body UPSET_DETECTION is");
     NEW_LINE(file);
     PUT_LINE(file,"procedure  HANDLER is");
     NEW_LINE(file);
     PUT_LINE(file,"begin");
     NEW_LINE(file);
     PUT_LINE(file,"raise INCORRECT_TAG;");
     PUT_LINE(file,"exception");
     PUT(file,"   when INCORRECT_TAG => PUT_LINE(");
```

```
    PUT(file,"");
    PUT(file," incorrect block entry");
    PUT(file,"");
    PUT_LINE(file,");");
    PUT_LINE(file,"end HANDLER;");
    PUT_LINE(file,"end UPSET_DETECTION;");
end FORM_PACKAGE;
```

```
separate (SMUD)
procedure OUTPUT(program_units : in integer;watchdogs : in integer;
                 node : integer) is

-- The purpose of this procedure is to create an output file that
-- contains the information obtained from PASS_ONE. The information
-- includes the type and name of each procedure, function, and task,
-- and the type of upset detection mechanisms that are to be employed.
-- The created file is used only for information purposes and is not
-- part of the executable system.


count : integer;
data_output : file_type;


begin

  CREATE(data_output,out_file,"data.ada");
  PUT(data_output,"Number of watchdog timers needed for node:  ");
  PUT(data_output,watchdogs,2);
  NEW_LINE(data_output);
  NEW_LINE(data_output);
  PUT_LINE(data_output,"unit id    block  wdt timer  program unit  name");
  for count in 1..(program_units - 1)
  loop
    PUT(data_output,unit(count).id,3);
    PUT(data_output,"  ");
    if unit(count).block = omit then
       PUT(data_output,"omit     ");
    elsif unit(count).block = show then
       PUT(data_output,"show     ");
    else
       PUT(data_output,"place    ");
    end if;
    if unit(count).wdt = omit then
       PUT(data_output,"omit     ");
    elsif unit(count).wdt = show then
       PUT(data_output,"show     ");
    else
       PUT(data_output,"place    ");
    end if;
    PUT(data_output,unit(count).watchdog,3);
    PUT(data_output,unit(count).kind(1..9));
    if unit(count).kind(1..4) = "task" then
     PUT(data_output,"    ");
    elsif unit(count).kind(1..8) = "function" then
     PUT(data_output," ");
    elsif unit(count).kind(1..7) = "package" then
     PUT(data_output,"  ");
    end if;
    PUT(data_output,"  ");
    PUT(data_output,unit(count).name(1..unit(count).len));
    PUT(data_output,"     ");
    PUT(data_output,unit(count).tag_name(1..4));
    PUT(data_output,"    ");
```

```
    new_line(data_output);
  end loop;
end OUTPUT;
```

```
separate (SMUD)
procedure CONVERT (line : in out string;
                   length : in integer) is

-- The purpose of this procedure is to convert each upper case letter
-- to a lower case letter.  This allows each string that is parsed
-- in a file to be compared consistently with a lower case string.


count : integer;

begin
  for count in 1..length loop
     if line(count) = 'A' then
       line(count) := 'a';
     elsif line(count) = 'B' then
       line(count) := 'b';
     elsif line(count) = 'C' then
       line(count) := 'c';
     elsif line(count) = 'D' then
       line(count) := 'd';
     elsif line(count) = 'E' then
       line(count) := 'e';
     elsif line(count) = 'F' then
       line(count) := 'f';
     elsif line(count) = 'G' then
       line(count) := 'g';
     elsif line(count) = 'H' then
       line(count) := 'h';
     elsif line(count) = 'I' then
       line(count) := 'i';
     elsif line(count) = 'J' then
       line(count) := 'j';
     elsif line(count) = 'K' then
       line(count) := 'k';
     elsif line(count) = 'L' then
       line(count) := 'l';
     elsif line(count) = 'M' then
       line(count) := 'm';
     elsif line(count) = 'N' then
       line(count) := 'n';
     elsif line(count) = 'O' then
       line(count) := 'o';
     elsif line(count) = 'P' then
       line(count) := 'p';
     elsif line(count) = 'Q' then
       line(count) := 'q';
     elsif line(count) = 'R' then
       line(count) := 'r';
     elsif line(count) = 'S' then
       line(count) := 's';
     elsif line(count) = 'T' then
       line(count) := 't';
     elsif line(count) = 'U' then
       line(count) := 'u';
     elsif line(count) = 'V' then
```

```
      line(count) := 'v';
    elsif line(count) = 'W' then
      line(count) := 'w';
    elsif line(count) = 'X' then
      line(count) := 'x';
    elsif line(count) = 'Y' then
      line(count) := 'y';
    elsif line(count) = 'Z' then
      line(count) := 'z';
    end if;
  end loop;
end CONVERT;
```

```
separate (SMUD)
procedure PASS_ONE(input_data,
                   output_data : in file_type;
                    program_units : in out integer;

                    unit_level : in out integer;
                    watchdogs : in out integer) is



-- This procedure is called by the main program SMUD to gather
-- information about the user's application program. This information
-- will be used in PASS_TWO to detemine  what type of fault detection
-- mechanism to implement. Two procedures are used to gather infor-
-- mation about the program units, program unit names, and the assined
-- tag variable. A record data structure is used to store the
-- information associated with each program unit.


  max_length : constant := 120;
  eol : boolean := false;
  in_block : boolean ;
  tag_level : integer :=0;
  WD : array (1..10) of integer;
  count : integer;
  tag_number : integer := 1;
  buffer : string(1..max_length);
  line : string(1..max_length);
  keyword : string(1..max_length);
  length : integer;
```

```
  procedure GET_KEY_WORD(line : in string;
                         length : in integer;
                         word : out string;
                         count : in out integer;
                         eol : out boolean) is


-- This procedure determine mainly the first of a line.  It is
-- used to determine whether the line read in indicates that a
-- program unit has been encountered by extracting the first ten
-- characters.


start : integer;
len : integer;

begin

  eol := false;
  for start in 1..max_length loop
     word(start) := ' ';
  end loop;

  loop
    exit when (line(count) /= ' ') or (count = length);
    count := count + 1;
  end loop;

  start := count;
  if count = length then
    eol := true;
  else
    loop
      exit when (line(count) = ' ') or (count = length);
      count := count + 1;
    end loop;
    if count = length then
       eol := true;
    end if;
    len := (count - start) + 1;
    word(1..len) := line(start..count);
  end if;

end GET_KEY_WORD;
```

```
procedure DO_BLOCK(level : in out integer) is
-- Procedure DO_BLOCK is called when the first BEGIN statement
-- of a program unit is encountered. The purpose of this procedure
-- is to read through the program unit and determine whether a block
-- checking structure and/or a watchdog timer is to be implemented.
-- The number of statements is also determined.


number_ends : integer := 1;
num_begins : integer := 0;
count : integer := 1;
line_number : integer := 0;
```

```
procedure CHECK_WORD (number_ends : in out integer;
                      keyword : in string) is


-- The purpose of this procedure is to determine if a word has an
-- associated end statement or if it is a directive to SMUD.  If the
-- word has an associated end statement, the number of end statements is
-- incremented;  if the word is part of an end statement, the number
-- of end statements is decremented.  The input parameters to this
-- procedure are the number of end statements and the word to be
-- checked.  The number of end statements is passed back to the
-- calling program unit so that the end of a block can be determined.


begin

    if (keyword(1..7) = "select ") or
        (keyword(1..5) = "loop ") or
        (keyword(1..3) = "do ") or
        (keyword(1..5) = "case ") or
        (keyword(1..6) = "begin ") or
        (keyword(1..3) = "if ") then
     number_ends := number_ends + 1;
     elsif keyword(1..4) = "--$$" then
        if keyword(5..12) = "omit_wdt" then
            unit(level).wdt := omit;
        elsif keyword(5..12) = "omit_blk" then
            unit(level).block := omit;
        elsif keyword(5..12) = "show_blk" then
            unit(level).block := show;
        elsif keyword(5..12) = "show_wdt" then
            unit(level).wdt := show;
        end if;
    elsif (keyword(1..4) = "end ") or (keyword(1..4) = "end;")
       or (keyword(1..10) = "exception ") then
      number_ends := number_ends - 1;
      eol := true;
     end if;
end CHECK_WORD;
```

```
--   procedure DO_BLOCK

begin

   unit(level).id := level;
   unit(level).block := place;
   unit(level).wdt := place;

   -- this loop reads in each line of the program unit

   loop
    exit when number_ends = 0;
    GET_LINE(input_data,buffer,length);
    line(1..length) := buffer(1..length);
    CONVERT(line,length);
    if length /= 0 then

      -- this loop pads the remaining spaces of a line with blanks

      for count in length+1..max_length
       loop
         line(count) := ' ';
       end loop;
      count := 1;

      -- this loop will parse each word of a line to determine if
      -- any SMUD directives have been placed in the code.

      loop
        GET_KEY_WORD(line,length,keyword,count,eol);
        if keyword(1..2) = "--" then
          eol := true;
        end if;
        CHECK_WORD(number_ends,keyword);
        exit when eol;
      end loop;
    end if;
 end loop;
end DO_BLOCK;
```

```
procedure DO_UNIT(line : in out string;
                  length : in out  integer;
                  word : in out string;
                  count : in out integer;
                  in_block : in out boolean;
                  nesting_level : in out integer;
                  unit_level : in out integer;
                  program_units : in out integer) is



-- This procedure is called after a program unit declaration has been
-- encountered. The purpose of this procedure is to determine the
-- type of program unit, the name of the program unit and the nesting
-- level. Procedure TEST is called to determine whether or not the
-- program unit is declared separate or not. If it is not declared
-- separate, the program unit is assigned an identifying
-- number that uniquely identifies it. The ID assigned
-- corresponds to the order in which the program unit was encountered
-- Although different variable names are given for each TAG memory
-- location, it is not necessary since each task or subprogram will
-- have the TAG variable declared in its declarative section.


temp : integer;
len : integer;
in_unit : boolean := false;
name : string(1..max_length);
```

```
procedure TEST(line : in out string;
               count : in out  integer;
               length : in out integer;
               in_unit : out boolean;
                keyword : in out string) is

-- This procedure is called by procedure PASS_ONE when one the following
-- words has been encountered while parsing:  procedure, function, or
-- task. The purpose of this procedure is to determine whether or not
-- the program unit has been declared as being separate. The parameters
-- passed into this procedure allow each word of the current line to be
-- parsed. The keywords is separate are searched. If the
-- keyword is is present without the keyword separate
--  then the boolean flag in_unit is set to the value
-- of true to indicate to the calling program that it has to assign
-- an ID value to the encountered program unit. If the keyword
-- separate is encountered, in_unit remains set to
-- false (as it was set upon entry to the procedure). It is this
-- parameter that the calling procedure examines to determine the next
-- course of action.


done : integer;
temp : integer;



begin

 in_unit := false;
 OUTER:
 loop
    if (count >= length) or (eol) then
       count := 1;
       for temp in 1..max_length loop
          line(temp) := ' ';
       end loop;
       GET_LINE(input_data,line,length);
     end if;

     if length /= 0 then
     loop
       GET_KEY_WORD(line,length,keyword,count,eol);
       if keyword(1..3) = "is " then
         loop
           if (count >= length) or (eol) then
             for temp in 1..max_length loop
               line(temp) := ' ';
             end loop;
           count := 1;
           GET_LINE(input_data,line,length);
         end if;

         if length /= 0 then
           loop
             GET_KEY_WORD(line,length,keyword,count,eol);
```

```
            if keyword(1) /= ' ' then
             if keyword(1..9) = "separate;" then
                in_unit := false;
                exit OUTER;
             else
                in_unit := true;
                exit OUTER;
             end if;
            end if;
             exit when eol;
           end loop;
         end if;
       end loop;
     end if;
     exit when eol;
    end loop;
  end if;
end loop OUTER;

end TEST;
```

```
begin  -- Determine if program unit is declared separately.

 if word(1..5) = "task " then
    GET_KEY_WORD(line,length,word,count,eol);
    if word(1..5) = "body " then
        unit(program_units).kind(1..4) := "task";
        GET_KEY_WORD(line,length,word,count,eol);
        name := word;
        if program_units > 1 then
           watchdogs := watchdogs + 1;
         if nest > 0 then
           in_save(nest) := 0;
           save(nest) := current_watchdog;
         end if;
           nest := nest + 1;
           current_watchdog := watchdogs;
           unit(program_units).watchdog := current_watchdog;
        end if;
        TEST(line,count,length,in_unit,word);
     end if;

 else
      if keyword(1..9) = "procedure" then
        unit(program_units).kind(1..9) := "procedure";
      else
        unit(program_units).kind(1..8) := "function";
      end if;
      GET_KEY_WORD(line,length,word,count,eol);
      name := word;
      TEST(line,count,length,in_unit,word);

 end if;

-- If the program unit is not declared separately, then the following
-- statements are executed.

 if unit(program_units).watchdog = 0 then
    unit(program_units).watchdog := current_watchdog;
 end if;
 if in_unit then
   temp := 1;
   loop
     exit when (name(temp) = '(') or (temp = max_length) or (name(temp) = ' ');
     temp := temp + 1;
   end loop;
   unit(program_units).len := temp - 1;
   len := temp - 1;
   unit(program_units).name(1..len) := name(1..len);
   if in_block = true then
      n(nesting_level) := unit_level;
   else
      in_block := true;
   end if;
  if nest > 0 then
   in_save(nest) := in_save(nest) + 1;
 end if;
```

```
   unit_level := unit_level + 1;
   nesting_level := nesting_level + 1;
   program_units := program_units + 1;
end if;
end DO_UNIT;
```

```
------- PASS_ONE -------

begin

  while not end_of_file(input_data)
   loop
    if keyword(1..6) /= "begin " then
     GET_LINE(input_data,line,length);
    end if;
     buffer(1..length) := line(1..length);
     CONVERT(line,length);
     if length /= 0 then
      for count in length+1..max_length
        loop
          line(count) := ' ';
        end loop;
     count := 1;
     GET_KEY_WORD(line,length,keyword,count,eol);
   if length /= 0 then
     if (keyword(1..10) = "procedure ") or
       (keyword(1..5)  = "task ")      or
       (keyword(1..9)  = "function ")  then
       DO_UNIT(line,length,count,in_block,
                 nesting,unit_level,program units);

     elsif
       keyword(1..6) = "begin " then
         if in_block then
           tag_level := unit_level;
         else
           tag_level := n(nesting);
         end if;
         DO_BLOCK(TAG_level);
       if nest > 0 then
         in_save(nest) := in_save(nest) - 1;
         if in_save(nest) = 0 then
           nest := nest - 1;
       if nest > 0 then
         current_watchdog := save(nest);
       end if;
         end if;
       end if;
         nesting := nesting - 1;
         in_block := false;
     end if;
   end if;
   end if;
  end loop;
end PASS_ONE;
```

```
separate (SMUD)
procedure PASS_TWO(input_data,
                   output_data : in file_type;
                    program_units : in integer;
                    unit_level : in out integer;
                    file : in out integer) is

-- This procedure is called by SMUD to process each file in the
-- application software.  While parsing each file, each program unit's
-- record file is exammined for the information obtained in PASS_ONE.
-- The program units are modified according to this information.


  max_length : constant := 120;
  eol : boolean := false;
  tag_level : integer;
  number_ends : integer := 1;
  line_number : integer := 0;
  in_block : boolean;
  line_limit :  constant := 20;
  count : integer;
  line : string(1..max_length);
  buffer : string(1..max_length);
  keyword : string(1..max_length);
  length : integer;
```

```
  procedure GET_KEY_WORD(line : in string;
                         length : in integer;
                         word : out string;
                         count : in out integer;
                         eol : out boolean) is


-- This procedure determine mainly the first of a line.  It is
-- used to determine whether the line read in indicates that a
-- program unit has been encountered by extracting the first ten
-- characters.


start : integer;
len : integer;

begin

  eol := false;
  for start in 1..max_length loop
    word(start) := ' ';
  end loop;

  loop
    exit when (line(count) /= ' ') or (count = length);
    count := count + 1;
  end loop;

  start := count;
  if count = length then
    eol := true;
  else
    loop
      exit when (line(count) = ' ') or (count = length);
      count := count + 1;
    end loop;
    if count = length then
       eol := true;
    end if;
    len := (count - start) + 1;
    word(1..len) := line(start..count);
  end if;

end GET_KEY_WORD;
```

```
procedure PROCESS_BLOCK(level : in out integer) is

-- This procedure is called by PASS_TWO when the first begin
-- statement has been encountered. Each line of the program unit is
-- parsed to determine where to place block-checking constructs and
-- a watchdog timer mechanism. Before actually modifying any program
-- unit, it's associated record file is examined to determine whether
-- or not any modifications are to take place.


temp : integer;
watchdog : boolean := false;
tag : boolean := false;
print_line : boolean := true;
reset : boolean := false;
eol : boolean := false;
len : integer;
start : integer;
name : string(1..max_length);
procedure_name : boolean := false;
count : integer;
number_ends : integer := 1;
time : string(1..5);
```

```
procedure PUT_SPACE(count : integer) is

space : integer;

begin
  for space in 1..count loop
    PUT(output_data," ");
  end loop;
end PUT_SPACE;



procedure SET_TAG(count,level : in integer) is

 -- This procedure is called by PROCESS_BLOCK to insert code that
 -- checks the value of a TAG.  The checking is done in the
 -- form of an if-then-else statement.  The format is given
 -- below:
 --
 --          if TAG3 = 0 then
 --            TAG3 := 3;
 --          else
 --            HANDLER;
 --          end if;
 --


begin

   NEW_LINE(output_data);
   PUT_SPACE(count);
   PUT(output_data,"If TAG");
   PUT(output_data,level,0);
   PUT_LINE(output_data," = 0 then");
   PUT_SPACE(count);
   PUT(output_data,"   TAG");
   PUT(output_data,level,0);
   PUT(output_data," := ");
   PUT(output_data,level,0);
   PUT_LINE(output_data,";");
   PUT_SPACE(count);
   PUT_LINE(output_data,"else");
   PUT_SPACE(count);
   PUT_LINE(output_data,"   HANDLER;");
   PUT_SPACE(count);
   PUT_LINE(output_data,"End if;");
   NEW_LINE(output_data);

end SET_TAG;
```

```
procedure TAG_CHECK(count ,level: in integer) is

-- This procedure is called to insert a TAG check into the code
-- whenever a tag check directive is encountered.  The code that
-- is inserted is the following:
--
--        If TAG3 /= 3 then
--          HANDLER;
--        end if;
--

begin
   NEW_LINE(output_data);
   NEW_LINE(output_data);
   PUT_SPACE(count);
   PUT(output_data,"If TAG");
   PUT(output_data,level,0);
   PUT(output_data," /= ");
   PUT(output_data,level,0);
   PUT_LINE(output_data," then");
   PUT_SPACE(count);
   PUT_LINE(output_data,"   HANDLER;");
   PUT_SPACE(count);
   PUT_LINE(output_data,"End if;");
   NEW_LINE(output_data);
end TAG_CHECK;
```

```
  procedure RESET_TAG(count,level : in integer) is
  -- This procedure is called to insert code at the end of a block,
  -- either before an exception handler, or before the final 'end'
  -- statement of a block.  The format is:
  --
  --          If TAG3 /= 3 then
  --             HANDLER;
  --          else
  --             TAG3 := 0;
  --          end if;
  --

  begin
     NEW_LINE(output_data);
     NEW_LINE(output_data);
     PUT_SPACE(count);
     PUT(output_data,"If TAG");
     PUT(output_data,level,0);
     PUT(output_data," /= ");
     PUT(output_data,level,0);
     PUT_LINE(output_data," then");
     PUT_SPACE(count);
     PUT_LINE(output_data,"   HANDLER;");
     PUT_SPACE(count);
     PUT_LINE(output_data,"else");
     PUT_SPACE(count);
     PUT(output_data,"  TAG");
     PUT(output_data,level,0);
     PUT_IINE(output_data," := 0;");
     PUT_SPACE(count);
     PUT_LINE(output_data,"End if;");
     NEW_LINE(output_data);
end RESET_TAG;
```

```
procedure WDT (count : integer; time : in string) is

-- This procedure is called by PROCESS_BLOCK to insert code that
-- resets a watchdog timer in the following format.
--
--      WDT := time;
--
-- where time is a value passed into the subroutine.
--

begin

   PUT_SPACE(count);
   PUT(output_data,"WDT := ");
   PUT(output_data,time(1..3));
   PUT_LINE(output_data,";");

end WDT;
```

```
procedure CHECK_WORD (number_ends : in out integer;
                      keyword : in string;
                      watchdog : out boolean;
                      reset : out boolean;
                      tag : out boolean) is


-- The purpose of this procedure is to determine if a word has an
-- associated end statement or if it is a directive to SMUD.  If the
-- word has an associated end statement, the number of end statements is
-- incremented;  if the word is part of an end statement, the number
-- of end statements is decremented.  The input parameters to this
-- procedure are the number of end statements and the word to be
-- checked.  The number of end statements is passed back to the
-- calling program unit so that the end of a block can be determined.


begin

    if (keyword(1..7) = "select ") or
        (keyword(1..3) = "do ") or
        (keyword(1..5) = "case ") or
        (keyword(1..6) = "begin ") or
        (keyword(1..3) = "if ") then
     number_ends := number_ends + 1;
     elsif keyword(1..5) = "loop " then
       number_ends := number_ends + 1;
       tag := true;
     elsif keyword(1..4) = "--$$" then
        if keyword(5..7) = "wdt" then
            time  :=  keyword(8..12);
             watchdog := true;
        end if;
        if keyword(5..7) = "blk" then
          tag := true;
        end if;
        if keyword(5..12) = "omit_tag" then
          tag := false;
        end if;
    elsif (keyword(1..4) = "end ") or (keyword(1..4) = "end;")
       or (keyword(1..10) = "exception ") then
      if number_ends = 1  then
        reset := true;
      end if;
      number_ends := number_ends - 1;
      eol := true;
    end if;
end CHECK_WORD;
```

```
procedure FIND_PROCEDURE( name : string;
                          procedure_name : out boolean;
                          program_units : in integer) is

-- The purpose of this procedure is to determine if the current line
-- contains the name of a program unit (such as a called procedure
-- or function). If so, the boolean flag procedure_name
-- is set the value of true.

length : integer;
file : integer;

begin

  procedure_name := false;
  for file in 1..(program_units-1) loop
    length := unit(file).len;
    if name(1..length) = unit(file).name(1..length) then
      procedure_name := true;
      exit;
    end if;
  end loop;

end FIND_PROCEDURE;
```

```
begin      -- PROCESS_BLOCK

  -- The following code checks the program unit's record file and
  -- modifies accordingly.

  procedure_name := false;
  count := 1;
   if unit(level).block = place then
     SET_TAG(count,level);
   elsif unit(level).block = show then
     PUT_LINE(output_data,"-- block check here");
     NEW_LINE(output_data);
   end if;
   if unit(level).WDT = place then
      time(1..3) := "255";
      WDT(count,time);
   elsif unit(level).WDT = show then
      PUT_LINE(output_data,"-- watchdog set here");
   end if;

  -- The following loop reads in each line of the program unit.

  loop
   exit when number_ends = 0;
   GET_LINE(input_data,buffer,length);
   line(1..length) := buffer(1..length);
   CONVERT(line,length);
   print_line := true;
   if length /= 0 then
     for count in length+1..max_length
      loop
        line(count) := ' ';
      end loop;
     count := 1;

  -- The following loop parses each word in a line.

 loop
   start := 1;
   loop
     exit when (line(start) /= ' ') or (start = max_length);
     start := start + 1;
   end loop;
   GET_KEY_WORD(line,length,keyword,count,eol);
   if keyword(1..4) = "--$$" then
     if keyword(5..12) /= "omit_tag" then
       print_line := false;
     end if;
   end if;
   if keyword(1..2) = "--" then
     eol := true;
   end if;
     CHECK_WORD(number_ends,keyword,watchdog,reset,tag);
     FIND_PROCEDURE(keyword,procedure_name,program_units);
```

```
  exit when eol or procedure_name;
end loop;

   -- The following code modifies the input file to contain a
   -- statement to set a watchdog timer or a commented statement
   -- indicating the location of the modification.

   count := start -1;
   if watchdog then
     if unit(level).wdt = place then
       WDT(count,time);
     elsif unit(level).wdt = show then
       PUT_LINE(output_data,"       --reset watchdog timer here");
     end if;
   end if;

   -- If the end of a program unit has been encountered, insert the
   -- TAG resetting instruction or commented statement.

   if reset then
      if unit(level).block = place then
        RESET_TAG(count,level);
      elsif unit(level).block = show then
        PUT_LINE(output_data,"-- tag reset here");
      end if;
      reset := false;
   end if;


   if print_line then
     PUT_LINE(output_data,buffer(1..length));
    end if;

   -- If a tag check directive has been encountered, insert the
   -- tag checking instructions.

   if tag then
      if unit(level).block = place then
        TAG_CHECK(count,level);
      elsif unit(level).block = show then
        PUT_LINE(output_data," -- tag check here");
      end if;
      line_number := 0;
      tag := false;
   end if;

   if procedure_name and (number_ends /= 0) and (not watchdog) then
      if unit(level).wdt = place then
        time(1..3) := "255";
        WDT(count,time);
      elsif unit(level).wdt = show then
        PUT_LINE(output_data,"      -- reset watchdog timer here");
      end if;
      procedure_name := false;
   end if;
```

```
      watchdog := false;
    else
      new_line(output_data);
    end if;
  end loop;
end PROCESS_BLOCK;
```

```
procedure DECLARE_TAG(line : in out string;
                      length : in out integer;
                      word : in out string;
                      count : in out integer;
                      in_block : in out boolean;
                      nesting_level : in out integer;
                      unit_level : in out integer;
                       file : in out integer) is


-- This procedure is called after a program unit declaration has been
-- encountered. The purpose of this procedure is to determine the
-- type of program unit, the name of the program unit and the nesting
-- level. Procedure TEST is called to determine whether or not the
-- program unit is declared separate or not. If it is not declared
-- separate, the program unit is assigned an identifying
-- number that uniquely identifies it. The ID assigned
-- corresponds to the order in which the program unit was encountered
-- Although different variable names are given for each TAG memory
-- location, it is not necessary since each task or subprogram will
-- have the TAG variable declared in its declarative section.



in_unit : boolean := false;
procedure TEST(line : in out string;
               count : in out  integer;
               length : in out integer;
               in_unit : out boolean;
                keyword : in out string) is


-- This procedure is called by procedure PASS_TWO when one the following
-- words has been encountered while parsing:  procedure, function, or
-- task. The purpose of this procedure is to determine whether or not
-- the program unit has been declared as being separate. The parameters
-- passed into this procedure allow each word of the current line to be
-- parsed. The keywords is separate are searched. If the
-- keyword is is present without the keyword separate
--  then the boolean flag in_unit is set to the value
-- of true to indicate to the calling program that it has to assign
-- an ID value to the encountered program unit. If the keyword
-- separate is encountered, in_unit remains set to
-- false (as it was set upon entry to the procedure). It is this
-- parameter that the calling procedure examines to determine the next
-- course of action.



done : integer;
temp : integer;
```

```
begin

 in_unit := false;
 OUTER:
 loop
    if (count >= length) or (eol) then
      count := 1;
      PUT_LINE(output_data,line(1..length));
      for temp in 1..max_length loop
        line(temp) := ' ';
      end loop;
      GET_LINE(input_data,line,length);
     end if;

     if length /= 0 then
     loop
      GET_KEY_WORD(line,length,keyword,count,eol);
      if keyword(1..3) = "is " then
        loop
          if (count >= length) or (eol) then
            PUT_LINE(output_data,line(1..length));
            for temp in 1..max_length loop
              line(temp) := ' ';
            end loop;
          count := 1;
          GET_LINE(input_data,line,length);
        end if;

        if length /= 0 then
          loop
            GET_KEY_WORD(line,length,keyword,count,eol);
           if keyword(1) /= ' ' then
            if keyword(1..9) = "separate;" then
              in_unit := false;
              exit OUTER;
            else
              in_unit := true;
              exit OUTER;
            end if;
          end if;
            exit when eol;
          end loop;
       end if;
      end loop;
    end if;
    exit when eol;
   end loop;
  else
    NEW_LINE(output_data);
  end if;
end loop OUTER;

end TEST;
```

```
begin  -- Procedure DECLARE_TAG

   -- Determine if program unit is declared as separate

  if word(1..5) = "task " then
     GET_KEY_WORD(line,length,word,count,eol);
     if word(1..5) = "body " then
          TEST(line,count,length,in_unit,word);
     else
        PUT_LINE(output_data,line(1..length));
      end if;
  else
        TEST(line,count,length,in_unit,word);
  end if;

     -- If the program unit is not declared separate, insert the
     -- declaration for the TAG variable.

  if in_unit then
      file := file + 1;
   if in_block = true then
       n(nesting_level) := unit_level;
   else
       in_block := true;
   end if;
   unit_level := unit_level + 1;
   nesting_level := nesting_level + 1;
   if unit(file).block = place then
     put(output_data,"TAG");
     put(output_data,unit_level,0);
     put_line(output_data," : integer := 0;");
   end if;
   if keyword(1..6) /= "begin " then
     PUT_LINE(output_data,line(1..length));
  end if;
end if;
end DECLARE_TAG;
```

```
      -- procedure PASS_TWO

  begin

  while not end_of_file(input_data)
    loop
     if keyword(1..6) /= "begin " then
      GET_LINE(input_data,line,length);
     end if;
      buffer(1..length) := line(1..length);
      CONVERT(line,length);
      if length /= 0 then
       for count in length+1..max_length
         loop
           line(count) := ' ';
         end loop;
      count := 1;
      GET_KEY_WORD(line,length,keyword,count,eol);

     -- This section of code looks for either the start of a procedure,
     -- task, or function, or for the first begin statement of a
     -- program unit.

     if length /= 0 then
      if (keyword(1..10) = "procedure ") or
        (keyword(1..5)  = "task ")      or
        (keyword(1..9)  = "function ")  then
       DECLARE_TAG(line,length,keyword,count,in_block,nesting,unit_level,file);
      elsif
        keyword(1..6) = "begin " then
          if in_block then
            TAG_level := unit_level;
          else
            TAG_level := n(nesting);
          end if;
          PUT_LINE(output_data,buffer(1..length));
          PROCESS_BLOCK(TAG_level);
          nesting := nesting - 1;
          in_block := false;
      else
        PUT_LINE(output_data,buffer(1..length));
      end if;
    else
      new_line(output_data);
    end if;
   else
     new_line(output_data);
    end if;
    end loop;
end PASS_TWO;
```

The vita has been removed from the scanned document