NASA-TM-112502

# Using ADA tasks to simulate operating equipment

Louis A. DeAcetis

Oron Schmidt and Kumar Krishen

Reprinted from COMPUTERS IN PHYSICS, September/October 1990 © American Institute of Physics

# Using ADA tasks to simulate operating equipment

Louis A. DeAcetis
*Physics Department, Bronx Community College/CUNY, Bronx, New York 10453*
Oron Schmidt and Kumar Krishen
*NASA, Johnson Space Center, Houston, Texas 77058*

*(Received 15 November 1989; accepted 5 January 1990)*

A method of simulating equipment using ADA tasks is discussed. Individual units of equipment are coded as concurrently running tasks that monitor and respond to input signals. This technique has been used in a simulation of the space-to-ground Communications and Tracking subsystem of Space Station Freedom.

## INTRODUCTION

Many computer simulations[1-3] written in procedural languages (e.g., C, FORTRAN, or PASCAL) simulate systems of equipment by tracking signals through the components. Although this may represent a logical solution to the problem, it usually requires that a piece of equipment know not only what it is connected to for input, but also the destination of its output. This is contrary to the way equipment generally operates. For example, an amplifier may have inputs of line voltage, signal level (volts), and load impedance, and control settings of gain and ON/OFF switch position. Its output would include the output signal level, and perhaps some parameter indicating the quality of the output. When the input values change, the values of the output parameters change accordingly. The destination of the output signal is of no concern to the amplifier, and it therefore does not know (or care) what is connected to its output. (Note that we are treating loading as the input parameter "load impedance.")

The ADA language is especially suited to simulating a piece of equipment because of the "task" construct.[4] Each piece of equipment can be modeled as a concurrent free-running task that constantly monitors its input values and adjusts the outputs accordingly. As each unit reacts to changes in its input values, one can monitor the signal flow through a collection of components by placing "sensors" at strategic locations. Issues of signal transition delays, and other equipment characteristics can be addressed as needed.

## I. METHOD

Each unit of equipment is modeled as an ADA task. The individual equipment characteristics are supplied in separate ADA packages (one for each unit of equipment), which include the appropriate transfer function(s) for the input signal(s). All of the values that are external to a piece of equipment are stored in a global database (or "blackboard" data structure[5]), and the individual components link their internal values to these blackboard values. Each component task then monitors those blackboard values that serve as input to the equipment it is simulating, and while the equipment is ON and running, appropriate output values for that equipment are generated and written out to the "blackboard" where they can then be monitored for input by those tasks using them. For example, if all of the equipment is plugged into the same power source, then each monitors the blackboard value of the line voltage. If the line voltage vanishes ("blackout") or is low ("brownout"), then this can be incorporated into the determination of the appropriate output signal(s).

In order to prevent a task from "running" when the equipment is in the OFF state, an algorithm is used that requires that the equipment be turned ON in order for it to process input information. Table I contains the structure of this ON/OFF algorithm, which is coded in a generic ADA package so that it can be implemented ("instantiated" in ADA terminology) for each piece of equipment. One of the features of the algorithm is that it will accept and ignore any ON/OFF requests that are redundant, rather than queue them as might normally be the case with ADA task rendezvous (i.e., if an ON command is sent to a task that is already ON, then the command is discarded). Another feature is that the task of a piece of equipment in the OFF state does no processing, but merely waits for a rendezvous to turn it ON. This minimizes the use of CPU resources.

## II. ILLUSTRATIVE EXAMPLE

Consider the equipment depicted in Fig. 1, which consists of a saw-tooth function generator attached to a pulse generator/amplifier. The circled numbers refer to sensor or test points whose values are to be monitored. The waveform produced by the function generator is used by the pulse generator to determine the pulse width as follows: While the value of the input to the pulse generator is at or beyond a certain threshold value (taken as 0.5 V), the value of the pulse generator output is + 10.0 V; when the value of the input is below threshold, then the pulse generator output is 0.0 V. We thus have a waveform transformer that converts a saw-tooth signal into a rectangular pulse. The frequency of both active signals is the same, and the pulse width can be varied by changing the amplitude of the saw-tooth signal. In addition, effects of a "brownout" (line voltage less than a nominal value of 120 V) have been incorporated in

TABLE I. ADA package with task code for algorithm used to simulate an equipment unit.

```
--Package Generic_Equipment
--  Package containing task to implement specific instances of equipment
--  Last update: 11-02-89   LAD
----------
--This package has Ada procedures as formal parameters and therefore
--must be instantiated with procedures which implement the transfer
--functions of the actual equipment used.
--In Particular:
--  Procedure Set_OFF_Values       : Parameter values for equip. OFF;
--  Procedure Set_INITIAL_ON_Values:  " Values for equipment just turned ON;
--  Procedure Set_Running_Values   :  " Values for equipment ON & running
--------------------------------------------------------------------------
generic
      with procedure Set_OFF_Values;
      with procedure Set_INITIAL_ON_Values;
      with procedure Set_Running_Values;

package GENERIC_EQUIPMENT is
  ---
  task SWITCH_CONTROL is
    entry CLOSE_SWITCH;
    entry OPEN_SWITCH;
  end SWITCH_CONTROL;
  ---
  procedure DESTROY;
  ---
end GENERIC_EQUIPMENT;

-------------------------------------
package body GENERIC_EQUIPMENT is
  --
  SWITCH_IS_OPEN : boolean := true; -- Switch starts open (equip. is OFF)
  --                                --   Task must therefore be "turned on"
                                    --   before it will "run".
  ----------
  task body SWITCH_CONTROL is
   begin
     SWITCH_OPEN_OFF:
     loop
       SWITCH_CLOSED_ON:
       loop
                   --*** Switch Control Loop ***--
         SWITCH_CONTROL:
         loop
           if SWITCH_IS_OPEN       --Select when SWITCH_IS_OPEN:
            then                   -- Wait for rendezvous to close it
             select
              accept OPEN_SWITCH;       --Accept and ignore OPEN requests
             or
              accept CLOSE_SWITCH;      --Where Switch is actually Closed
              SWITCH_IS_OPEN := false;
              -----------------------
              Set_INITIAL_ON_Values; --
              -----------------------
              exit SWITCH_CONTROL;
             end select;
          else
            select
             accept CLOSE_SWITCH;      --Accept and ignore CLOSE requests
             exit SWITCH_CONTROL;
            or
             accept OPEN_SWITCH;       --Where Switch is actually Opened
             SWITCH_IS_OPEN := true;
             exit SWITCH_CLOSED_ON;
            else
             exit SWITCH_CONTROL;
            end select;
          end if;
        end loop SWITCH_CONTROL;
                  --*** End Switch Control Loop ***--
      --Should only get here if Switch is Closed/ON:
        ---------------------
        Set_Running_Values; --
        ---------------------
        delay 0.001;  --delay/Reschedule;
        --
      end loop SWITCH_CLOSED_ON;
     --
     --Should only get here if Switch was just opened:
       -----------------
       Set_OFF_Values; --
       -----------------
       delay 0.001;  --delay/Reschedule;
       --
     end loop SWITCH_OPEN_OFF;

   end SWITCH_CONTROL;
  ---------
  procedure DESTROY is    --Command to abort task (for orderly shutdown)
    begin
      abort SWITCH_CONTROL;
    end DESTROY;
  ---------
end GENERIC_EQUIPMENT;
```

that the output levels of each device will decrease until a "minimum operating voltage" for the equipment is reached: Output from the ~~function~~ pulse generator vanishes when the line voltage drops below 90 V, and for line voltages below 80 V, the output from the ~~pulse~~ function generator also ceases. Figure 2 is a dataflow diagram for this equipment setup.

Table II contains ADA code that implements the above for each piece of equipment and includes the structure of the global database. Table III is a main ADA proce-

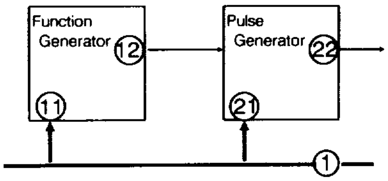


FIG. 1. Illustrative Example equipment setup consisting of a (saw-tooth) function generator whose output drives a pulse generator. Circled values indicate sensors or test points.

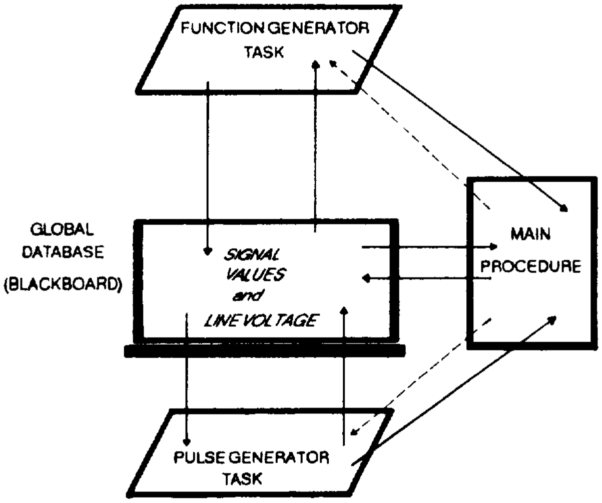


FIG. 2. Flow diagram of data and command information flowing between the blackboard global database, the ADA equipment simulator tasks, and the main procedure of the Illustrative Example. Straight line (3/ M) = data; dashed hne (– – – –) = commands.

TABLE II. ADA code for Illustrative Example equipment, including the Global Database definition and the packages for the function and pulse generators.

```
--package GLOBAL_VARIABLES is a global database where shared info is
--  stored and where connections between signals are made.
--Last Update: 11-02-89 LAD
-----------------------------------------------------------------------
package GLOBAL_VARIABLES is

  --Power Source Quantities:
    STD_LINE_VOLTAGE               : constant float := 120.0;
    MAX_LINE_VOLTAGE               : constant float := 130.0;
    LINE_VOLTAGE                   : float := STD_LINE_VOLTAGE;

  --Function Generator Signals and Levels
    FUNCTION_GENERATOR_AMPLITUDE : float           :=   1.0;
    FUNCTION_GENERATOR_OUTPUT    : float           :=   0.0;

  --Pulse Generator Signals and Levels
    PULSE_GENERATOR_AMPLITUDE    : float           :=  10.0;
    PULSE_TRIGGER_LEVEL          : float           :=   0.5;
    PULSE_GENERATOR_OUTPUT       : float           :=   0.0;
    --Connect Pulse Generator trigger input line to Funct. Gen. Output
    PULSE_TRIGGER_LINE           : float renames FUNCTION_GENERATOR_OUTPUT;

end GLOBAL_VARIABLES;
--------------------------------------------------------------------------
-- Function Generator Simulator -- Last update: 11-03-89 LAD
--***********************************************************************
--* Package with Procedures for the simulation of a Function Generator *
--*  which will generate a  Saw-tooth  waveform  of variable amplitude *
--*  (nominal value is 1.0 V).  Output is degraded if  LINE_VOLTAGE_IN *
--*  is less than 120 V ("brownout"),  and ceases when LINE_VOLTAGE_IN *
--*  is < 80 V ("blackout").                          Louis A. DeAcetis    *
--***********************************************************************

with GENERIC_EQUIPMENT;

with GLOBAL_VARIABLES;  -- Global Database where signal values are stored
                        --   and signal connections are made

package FUNCTION_GENERATOR is

  procedure OFF_Values;
  procedure INITIAL_ON_Values;
  procedure OUTPUT_Value;
  -----
  package EQUIPMENT is new GENERIC_EQUIPMENT(
                          OFF_Values, INITIAL_ON_Values, OUTPUT_Value);
  -----
  type PROBE_NAMES is (ON_OFF_SW, OUTPUT_SIGNAL);
  type PROBE_ARRAY_TYPE is array(PROBE_NAMES) of float;
  PROBE   : PROBE_ARRAY_TYPE := (others => 0.0); --Initialize sensors to 0
  -----
  COUNTER : float := 0.0;                          --Cycle count
  -----
end FUNCTION_GENERATOR;

----------------------------------------------------------------------
with calendar;

package body FUNCTION_GENERATOR is
  -----
  --Explicitly associate local variables with values in Global Database:
    LINE_VOLTAGE_IN : float renames GLOBAL_VARIABLES.LINE_VOLTAGE;
    STD_LINE_VOLTAGE: float renames GLOBAL_VARIABLES.STD_LINE_VOLTAGE;
    --
```

TABLE II. (*Continued.*)

```
   AMPLITUDE : float renames GLOBAL_VARIABLES.FUNCTION_GENERATOR_AMPLITUDE;
   --
   OUTPUT    : float renames GLOBAL_VARIABLES.FUNCTION_GENERATOR_OUTPUT;
   --
 -----
 INCREMENT       : calendar.day_duration;
 SAW_TOOTH_VALUE : float;
 BASE_TIME       : calendar.day_duration := calendar.seconds(calendar.clock);
 ---------------------------------------------------------------------------
 procedure OFF_Values is
   begin          --Set outputs to values for equipment in OFF state:
     PROBE(ON_OFF_SW)     := 0.0;
     OUTPUT               := 0.0;
     PROBE(OUTPUT_SIGNAL) := OUTPUT;
   end;
 -----
 procedure INITIAL_ON_Values is
   begin          --Set values for when equipment just turned ON:
     PROBE(ON_OFF_SW)     := 1.0;
     OUTPUT               := 0.0;
     PROBE(OUTPUT_SIGNAL) := OUTPUT;
     SAW_TOOTH_VALUE      := 0.0;
     BASE_TIME            := calendar.seconds(calendar.clock);
   end;
 -----
 procedure OUTPUT_VALUE is   --Set output values for equipment ON and running
     -----
     function WAVE_VALUE (AMPLITUDE : float) return float is
         --
         function SAW_TOOTH return float is
         begin
           INCREMENT       := calendar.seconds(calendar.clock) - BASE_TIME;
           SAW_TOOTH_VALUE := float(INCREMENT)/5.0;
           if SAW_TOOTH_VALUE > 1.0 then
              SAW_TOOTH_VALUE := 0.0;
              BASE_TIME       := calendar.seconds(calendar.clock);
           end if;
           return SAW_TOOTH_VALUE;
         end SAW_TOOTH;
         --
     begin
       return AMPLITUDE*SAW_TOOTH;
     end WAVE_VALUE;
   ---------
   begin
     COUNTER := COUNTER + 1.0;        --Count cycles for monitoring purposes

     if LINE_VOLTAGE_IN < 80.0 then --"Blackout" condition
       OUTPUT := 0.0;
      else
       OUTPUT := LINE_VOLTAGE_IN/STD_LINE_VOLTAGE*  --"Brownout factor"
                 WAVE_VALUE(AMPLITUDE);
     end if;
     --
     PROBE(OUTPUT_SIGNAL) := OUTPUT;
     --
   end OUTPUT_VALUE;
   -----
end FUNCTION_GENERATOR;
--*********************************************************************--
-- Pulse Generator Simulator -- Last Update: 11-07-89 LAD
--*********************************************************************
--* Package with Procedures for the simulation of a Pulse Generator which *
--*  generates a pulse of height PULSE_GENERATOR_AMPLITUDE when the value *
--*  of the TRIGGER_INPUT is greater than the value of the TRIGGER_LEVEL  *
--*  (output is zero otherwise). Output is degraded if LINE_VOLTAGE_IN is *
--*  less than 120 V ("brownout"),  and  ceases  when  LINE_VOLTAGE_IN is *
--*  < 90 V ("blackout").                             Louis A. DeAcetis    *
--*********************************************************************
with GENERIC_EQUIPMENT;

with GLOBAL_VARIABLES;  -- Global Database where signal values are stored
                        --   and signal connections are made

package PULSE_GENERATOR is
  --
  procedure OFF_Values;
  procedure INITIAL_ON_Values;
  procedure OUTPUT_Value;
  -----
  package EQUIPMENT is new GENERIC_EQUIPMENT(
                           OFF_Values,INITIAL_ON_Values,OUTPUT_Value);
  -----
  type PROBE_NAMES is (ON_OFF_SW, OUTPUT_SIGNAL);
  type PROBE_ARRAY_TYPE is array(PROBE_NAMES) of float;
  PROBE   : PROBE_ARRAY_TYPE := (others => 0.0); --Initialize sensors to 0
  -----
  COUNTER : float := 0.0;                         --Cycle count
  -----
end PULSE_GENERATOR;
--------------------------------------------------------------------------

package body PULSE_GENERATOR is
  -----
  --Explicitly associate local variables with values in Global Database
    LINE_VOLTAGE_IN : float renames GLOBAL_VARIABLES.LINE_VOLTAGE;
    STD_LINE_VOLTAGE: float renames GLOBAL_VARIABLES.STD_LINE_VOLTAGE;
    --
    AMPLITUDE       : float renames GLOBAL_VARIABLES.PULSE_GENERATOR_AMPLITUDE;
    --
    TRIGGER_INPUT   : float renames GLOBAL_VARIABLES.PULSE_TRIGGER_LINE;
    TRIGGER_LEVEL   : float renames GLOBAL_VARIABLES.PULSE_TRIGGER_LEVEL;
    --
    OUTPUT          : float renames GLOBAL_VARIABLES.PULSE_GENERATOR_OUTPUT;
    --
  -------------------------------------------------------------------------
  procedure OFF_Values is
    begin          --Set outputs to values for equipment in OFF state:
      OUTPUT               := 0.0;
      PROBE(OUTPUT_SIGNAL) := OUTPUT;
      PROBE(ON_OFF_SW)     := 0.0;
    end OFF_Values;
  -----
  procedure INITIAL_ON_Values is
    begin          --Set outputs for values when equipment just turned ON:
      PROBE(ON_OFF_SW)     := 1.0;
      OUTPUT               := 0.0;
      PROBE(OUTPUT_SIGNAL) := OUTPUT;
    end;
  -----
  procedure OUTPUT_VALUE is   --Set output values for equipment ON and running
    begin
       PROBE(ON_OFF_SW) := 1.0;     -- Switch sensor On
       COUNTER          := COUNTER + 1.0;
       --
       if LINE_VOLTAGE_IN < 90.0 or    -- "Blackout" condition check
                TRIGGER_INPUT < TRIGGER_LEVEL
        then
         OUTPUT := 0.0;
        else
         OUTPUT := LINE_VOLTAGE_IN/STD_LINE_VOLTAGE *  --"Brownout factor"
                   AMPLITUDE;
       end if;
       --
       PROBE(OUTPUT_SIGNAL) := OUTPUT;
       --
    end OUTPUT_VALUE;
  -----
end PULSE_GENERATOR;
```

TABLE III. ADA code for the procedure SIMULATE which displays the sensor and signal values for the Illustrative Example equipment.

```
-----------------------------------------------------------------------
-- Main program to exercise equipment: Function generator
--                                     Pulse generator
--   and display sensor readings.
--
-- Last Update: 11-02-89 Louis A. DeAcetis
-----------------------------------------------------------------------

with GLOBAL_VARIABLES;

with FUNCTION_GENERATOR;

with PULSE_GENERATOR;

with text_io;
 use text_io;

--DOS interface packages for Alsys or Meridian compilers:
--with tty; with video; --Meridian compiler
--with DOS;             --Alsys compiler

procedure SIMULATE is

  package INT_IO is new integer_io(integer);
      use INT_IO;

  package FLOT_IO is new float_io(float);
      use FLOT_IO;

  VALUE : float := 0.0;
  PARAM : character := ascii.nul;

  subtype ABSISSA  is integer range 5..36;
  subtype ORDINATE is integer range 9..23;

  X1, X2 : ABSISSA;
  LAST_LINE : ORDINATE := ORDINATE'last;
  THARKER : ABSISSA;
  BLANK_LINE   : string(ABSISSA'first..(ABSISSA'last+4)) := (others =>' ');
  WINDOW_ARRAY : array(ORDINATE) of string(ABSISSA'first..(ABSISSA'last+4))
                    := (others => BLANK_LINE);
  -----
  type PROBE_NAME is (MAIN_POWER_LINE_VOLTAGE,
                      F_GEN_ON_OFF_SW,
                      F_GEN_OUTPUT,
                      P_GEN_ON_OFF_SW,
                      P_GEN_OUTPUT);

  type PROBE_ARRAY_TYPE is array(PROBE_NAME) of float;

  PROBE   : PROBE_ARRAY_TYPE := (others => 0.0); --Initialize sensors to 0
  PREVIOUS_PROBE : PROBE_ARRAY_TYPE := (others => 0.0);
  COUNTER: PROBE_ARRAY_TYPE := (1.0, others => 0.0);
  -----
--Create link between local values and Global Values:
  LINE_VOLTAGE     : float renames GLOBAL_VARIABLES.LINE_VOLTAGE;
  MAX_LINE_VOLTAGE : float renames GLOBAL_VARIABLES.MAX_LINE_VOLTAGE;
  FUNCTION_GENERATOR_AMPLITUDE : float renames
                          GLOBAL_VARIABLES.FUNCTION_GENERATOR_AMPLITUDE;
  PULSE_GENERATOR_AMPLITUDE    : float renames
                          GLOBAL_VARIABLES.PULSE_GENERATOR_AMPLITUDE;
  PULSE_TRIGGER_LEVEL : float renames GLOBAL_VARIABLES.PULSE_TRIGGER_LEVEL;
---------------------------------------------------------------------
procedure SET_CURSOR  ( X: in integer;      --NOTE: Uses ANSI escape
                        Y: in integer) is   --  sequences. Requires
  SCREEN_WIDTH : integer := 80;             --  DRIVER = ANSI.SYS
  XX,YY : integer;                          --  in MS-DOS CONFIG.SYS file
begin

  if X < 1 then XX := 1;
    elsif X > SCREEN_WIDTH then XX := SCREEN_WIDTH;
    else XX := X;
  end if;

  if Y < 1 then YY := 1;
    elsif Y > 24 then YY := 24;
    else YY := Y;
  end if;

  text_io.put(ascii.esc & "[" & integer'image(YY+100)(3..4) &
    ";" & integer'image(XX+1000)(3..5) & "H");
end SET_CURSOR;
--
procedure CLEAR_SCREEN is
  begin
    text_io.put(ascii.esc & "[2J");
end CLEAR_SCREEN;
--
  procedure DISPLAY_SENSORS is
  begin
   --Display screen labels:
    if COUNTER(MAIN_POWER_LINE_VOLTAGE) = 1.0 then
      SET_CURSOR(2,1);
        text_io.put("Probe                       Reading        Cycle Count");
      for I in PROBE_NAME'FIRST..PROBE_NAME'LAST loop
        SET_CURSOR(2,PROBE_NAME'pos(I)+2);
        text_io.put(PROBE_NAME'IMAGE(I));
      end loop;
    end if;

    for I in PROBE_NAME'FIRST..PROBE_NAME'LAST loop
     if COUNTER(I) /= 0.0 then
       SET_CURSOR(40,PROBE_NAME'pos(I)+2); put(integer(COUNTER(I)),5);
     end if;

     if not (PROBE(I) = PREVIOUS_PROBE(I)) then
      SET_CURSOR(27,PROBE_NAME'pos(i)+2);
      put(PROBE(I), 3, 2, 0);
      PREVIOUS_PROBE(I) := PROBE(I);
     end if;
    end loop;
    --
    new_line;  -- Force output to screen with new_line
  end DISPLAY_SENSORS;
--
 procedure PLOT_SIGNALS is
 begin
 --Scroll "window" contents:
   --Include following if Meridian Compiler:
   --video.scroll_up(1, ORDINATE'first-1, ABSISSA'first-1,
   --                   ORDINATE'last, ABSISSA'last+2);

 --\/ Include following if not Meridian Compiler
   for IY in ORDINATE'first+1..LAST_LINE loop
    SET_CURSOR(ABSISSA'first,IY-1);
    text_io.put(WINDOW_ARRAY(IY));
    WINDOW_ARRAY(IY-1) := WINDOW_ARRAY(IY);
   end loop;

 --Blank out last line:
   WINDOW_ARRAY(LAST_LINE) := BLANK_LINE;

   WINDOW_ARRAY(LAST_LINE)(
     (ABSISSA'last-ABSISSA'first)/2+ABSISSA'first+1) := '|';

   X1 := integer(PROBE(F_GEN_OUTPUT)*float(ABSISSA'last-ABSISSA'first)/
        2.0/FUNCTION GENERATOR AMPLITUDE) + ABSISSA'first;
```

TABLE III. (*Continued.*)

```
    X2 := integer(PROBE(P_GEN_OUTPUT)*float(ABSISSA'last-ABSISSA'first)/
          2.0/PULSE_GENERATOR_AMPLITUDE)+(ABSISSA'last+ABSISSA'first)/2-1;

    WINDOW_ARRAY(LAST_LINE)(X1)    := '+';
    WINDOW_ARRAY(LAST_LINE)(X2+3)  := '*';

    SET_CURSOR(ABSISSA'first, LAST_LINE);
    text_io.put(WINDOW_ARRAY(LAST_LINE));

    new_line; -- Force output with new_line

  end PLOT_SIGNALS;

-----
begin
--Set up Screen for displays:
  CLEAR_SCREEN;

--Draw "box" around data plotting "window"
  SET_CURSOR(ABSISSA'first-1,ORDINATE'first-2);
  text_io.put('/');
   for I in ABSISSA'first+1..ABSISSA'last+5 loop
     text_io.put('-');
   end loop;
  text_io.put('\');

--Mark trigger level over Function Generator output plot:
  TMARKER := integer(PULSE_TRIGGER_LEVEL*
                float(ABSISSA'last-ABSISSA'first)/2.0/
                FUNCTION_GENERATOR_AMPLITUDE+ 0.5) + ABSISSA'first;
  SET_CURSOR(TMARKER,ORDINATE'first-2); text_io.put('V');

  for I in ORDINATE'first-1..ORDINATE'last loop
    SET_CURSOR(ABSISSA'first-1,I); text_io.put('|');
    SET_CURSOR(ABSISSA'last+5,I);  text_io.put('|');
  end loop;

  SET_CURSOR(ABSISSA'first + 1, ORDINATE'first-1);
    text_io.put("Function Gen:");

  SET_CURSOR((ABSISSA'last-ABSISSA'first)/2 + ABSISSA'first+1,
            ORDINATE'first-1);
    text_io.put('|');
  SET_CURSOR((ABSISSA'last+ABSISSA'first)/2+4,ORDINATE'first-1);
    text_io.put("Pulse:");

  SET_CURSOR(43, 7); text_io.put("<><><><><><><><><><><><><><><><><><>");
  SET_CURSOR(43, 8); text_io.put("To change equipment parameters use:");
  SET_CURSOR(45, 9); text_io.put("F: Function Generator");
  SET_CURSOR(45,10); text_io.put("P: Pulse Generator");
  SET_CURSOR(45,11); text_io.put("L: Line Voltage");
  SET_CURSOR(45,12); text_io.put("A: Output Ampl. of Function Gen.");
  SET_CURSOR(45,14); text_io.put("VALUE: Off = 0.0; On = 1.0");
  SET_CURSOR(52,15); text_io.put(       "Voltage, Amplitude: float");
  SET_CURSOR(45,17); text_io.put("Enter F,P,L,A(space)VALUE: ");

--Turn on equipment

  FUNCTION_GENERATOR.EQUIPMENT.SWITCH_CONTROL.CLOSE_SWITCH;
     PULSE_GENERATOR.EQUIPMENT.SWITCH_CONTROL.CLOSE_SWITCH;

  delay 0.001;

  DISPLAY_SENSORS;

OUTER_LOOP: loop
begin                --Exception block

loop

  loop -- Is there input from the keyboard?
    --exit when DOS.KBD_DATA_AVAILABLE; --Alsys compiler
    --exit when tty.CHAR_READY;          --Meridian compiler
    --
    ----\/Include if no interface to DOS is used:
      --\/ (pauses every 20 cycles for input):
    exit when integer(COUNTER(MAIN_POWER_LINE_VOLTAGE))/20*20 -
              integer(COUNTER(MAIN_POWER_LINE_VOLTAGE)) = 0;

    COUNTER(F_GEN_ON_OFF_SW) := FUNCTION_GENERATOR.COUNTER;
    COUNTER(P_GEN_ON_OFF_SW) := PULSE_GENERATOR.COUNTER;

  --Fetch probe values from equipment and store locally for display
    PROBE(MAIN_POWER_LINE_VOLTAGE) := LINE_VOLTAGE;
    PROBE(F_GEN_ON_OFF_SW) := FUNCTION_GENERATOR.PROBE
                                (FUNCTION_GENERATOR.ON_OFF_SW);
    PROBE(F_GEN_OUTPUT)    := FUNCTION_GENERATOR.PROBE
                                (FUNCTION_GENERATOR.OUTPUT_SIGNAL);
    PROBE(P_GEN_ON_OFF_SW) := PULSE_GENERATOR.PROBE
                                (PULSE_GENERATOR.ON_OFF_SW);
    PROBE(P_GEN_OUTPUT)    := PULSE_GENERATOR.PROBE
                                (PULSE_GENERATOR.OUTPUT_SIGNAL);

    if (LINE_VOLTAGE > MAX_LINE_VOLTAGE) and
       (PROBE(F_GEN_ON_OFF_SW) + PROBE(P_GEN_ON_OFF_SW) /= 0.0)
     then
      CLEAR_SCREEN; new_line(12);
      text_io.put("*****Overvoltage on Line Voltage--"); new_line;
      text_io.put("*****"); new_line;
      text_io.put("***** Fuses blown-- Output ceases"); new_line;
      text_io.put("***** Replace blown fuses and start again.");
      new_line(5);
      FUNCTION_GENERATOR.EQUIPMENT.DESTROY;  --Abort Funct. Gen. task
         PULSE_GENERATOR.EQUIPMENT.DESTROY;  --Abort Pulse Gen. task
      exit OUTER_LOOP;
    end if;

    COUNTER(MAIN_POWER_LINE_VOLTAGE) :=
           COUNTER(MAIN_POWER_LINE_VOLTAGE)+1.0;

    DISPLAY_SENSORS;
    PLOT_SIGNALS;
    delay 0.25;

  end loop;
    SET_CURSOR(72,17); text_io.put("         ");
    SET_CURSOR(72,17);
    get(PARAM); get(VALUE);

  --Increment local cycle count:
    COUNTER(MAIN_POWER_LINE_VOLTAGE) :=
                            COUNTER(MAIN_POWER_LINE_VOLTAGE)+1.0;

  --Clear out error message if present from previous input:
    SET_CURSOR(45,21); text_io.put("                          ");
    SET_CURSOR(47,22); text_io.put("                               ");
    SET_CURSOR(69,23); text_io.put("    ");

  --Check input from keyboard for valid command:
    case PARAM is
      when 'f'|'F' =>
        if VALUE = 0.0 then
          FUNCTION_GENERATOR.EQUIPMENT.SWITCH_CONTROL.OPEN_SWITCH;
         else
          FUNCTION_GENERATOR.EQUIPMENT.SWITCH_CONTROL.CLOSE_SWITCH;
        end if;

      when 'p'|'P' =>
        if VALUE = 0.0 then
```

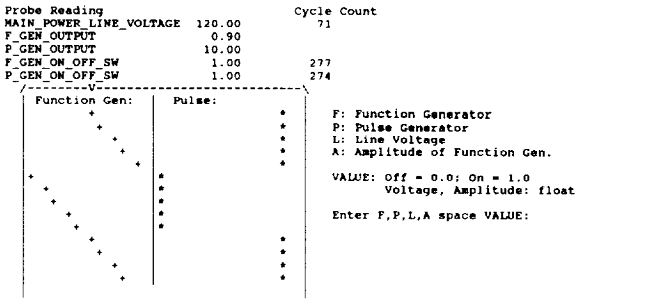
TABLE III. (*Continued.*)

```
          PULSE_GENERATOR.EQUIPMENT.SWITCH_CONTROL.OPEN_SWITCH;
         else
          PULSE_GENERATOR.EQUIPMENT.SWITCH_CONTROL.CLOSE_SWITCH;
        end if;

      when 'l'|'L' =>
        LINE_VOLTAGE := VALUE;

      when 'a'|'A' =>
        FUNCTION_GENERATOR_AMPLITUDE := VALUE;
        if PULSE_TRIGGER_LEVEL > VALUE then
          VALUE := FUNCTION_GENERATOR_AMPLITUDE;
         elsif VALUE < 0.0 then
          VALUE := 0.0;
         else VALUE := PULSE_TRIGGER_LEVEL;
        end if;

        TMARKER := integer(VALUE*float(ABSISSA'last-ABSISSA'first)/
            2.0/FUNCTION_GENERATOR_AMPLITUDE+0.5) + ABSISSA'first;

        SET_CURSOR(ABSISSA'first,ORDINATE'first-2);
        for I in ABSISSA'first+1..ABSISSA'last+5 loop
          text_io.put('-');
        end loop;
        SET_CURSOR(TMARKER,ORDINATE'first-2); text_io.put('V');

      when others =>
        raise data_error;

    end case;

    delay 0.001;

  end loop;
    exception
        when data_error =>
          SET_CURSOR(45,21); text_io.put("Erroneous input ignored--");
          SET_CURSOR(47,22); text_io.put("Proper form examples: L 110.0");
          SET_CURSOR(69,23); text_io.put("A 0.8");
          text_io.put(ascii.bel);
    end;              --Exception block

    end LOOP OUTER_LOOP;

  end SIMULATE;
```

dure, called SIMULATE, which interfaces with the simulator and produces output similar to that in Fig. 3. Although this is a simple application, it does illustrate the method and suggests how greater sophistication is possible.

## III. IMPLEMENTATION

Figure 4 is a block diagram of the starboard portion of the proposed Space-to-Ground subsystem of the Communications and Tracking System on Space Station Freedom. As above, the circled numbers represent sensors whose values can be monitored. The oval enclosed numbers represent sensors whose values indicate the ON/OFF state of the equipment. This system has been simulated using the above paradigm: Each rectangular box is represented by a task that, when "ON," monitors the values of its input signals and sets the values of the output values and sensors accordingly. The simulator may be controlled by other programs or using a keyboard interface program that permits the asynchronous entry of commands to turn equipment ON/OFF and set cross-strapping switches. There are upward of



```
Probe Reading                         Cycle Count
MAIN_POWER_LINE_VOLTAGE  120.00           71
F_GEN_OUTPUT               0.90
P_GEN_OUTPUT              10.00
F_GEN_ON_OFF_SW            1.00          277
P_GEN_ON_OFF_SW            1.00          274
  /--------V---------------------------\
  | Function Gen:  |  Pulse:           |
  |        +       |                *  |   F: Function Generator
  |         +      |                *  |   P: Pulse Generator
  |           +    |                *  |   L: Line Voltage
  |            +   |                *  |   A: Amplitude of Function Gen.
  |              + |                *  |
  |+               |*                  |   VALUE: Off = 0.0; On = 1.0
  |  +             |*                  |          Voltage, Amplitude: float
  |   +            |*                  |
  |     +          |*                  |   Enter F,P,L,A space VALUE:
  |      +         |*                  |
  |        +       |                *  |
  |         +      |                *  |
  |           +    |                *  |
  |            +   |                *  |
  |                                    |
```

FIG. 3. Sample output of procedure SIMULATE which displays probe readings and signal levels for the Illustrative Example. The outputs of the function and pulse generators are displayed graphically and scroll upward to show the changes with time. The "cycle count" indicates the number of cycles completed by each running task and procedure SIMULATE.

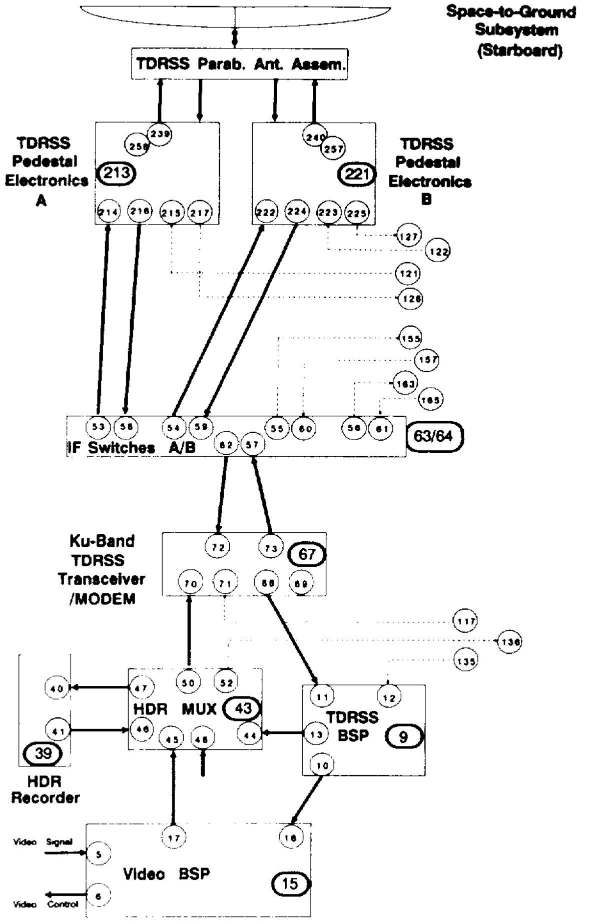


FIG. 4. Block diagram of the starboard space-to-ground subsystem of the Communications and Tracking System of Space Station Freedom. The numbers are sensor identifiers. BSP: baseband signal processor; HDR: high data rate recorder; TDRSS: Tracking and Data Relay Satellite System.

34 tasks for the total system (which includes the port system and contingency communications equipment not shown here). At present, the simulator transforms signal levels and sets sensor readings to typical values. Once the detailed electronic characteristics for this equipment (under development) are established, they can be incorporated into the corresponding tasks.

The simulator has been compiled under a variety of ADA compilers (including Alsys, DEC, Meridian, and Verdix), and runs under MS-DOS on PC's (80286 and 80386 CPU's), and DEC VMS and Ultrix operating systems. Although it is unlikely that all of the communications equipment on the Space Station would be ON simultaneously, the current simulator has been run in that state with no major problems. (As more tasks are switched "ON," they do slow execution somewhat, especially on an IBM PC-AT.)

Additional refinements of the model presented here are possible, most especially in the area of information hiding. One of the major problems with a blackboard approach is that signal information is not only visible to all of the equipment tasks, but accessible (i.e., modifiable) as well. It is thus possible for the Function Generator to get access to the output value of the Pulse Generator even though it has no electrical connection to that output. However, from the code in Table II, it can be seen that one would deliberately have to associate a local variable with an improper global variable in order to accomplish such a connection. There are ADA constructs that can be used to prevent any unwarranted access to normally inaccessible signal levels,[6] but the level of abstraction and programming complexity required would obscure what is basically a simple concept and implementation, and these were deemed beyond the scope and intent of the present article.

## IV. CONCLUSIONS

One of the unique aspects of the ADA programming language is the ability to do logically "parallel" processing using the task construct. This is especially useful in simulating concurrently running equipment. Satisfactory results are readily obtainable for situations where transient states can be ignored (e.g., where we are not concerned with the output of the pulse generator during the transitions between its minimum and maximum voltage states). When the latter are important, timing considerations can greatly increase the complexity of the problem. Real-time simulations, which require timing and interrupt considerations, constitute a further challenge in ADA,[7–10] which is not considered here.

## ACKNOWLEDGMENT

This work was done while the first author held a National Research Council–NASA Senior Research Associateship.

## REFERENCES

1. G. Gordon, *System Simulation* (Prentice-Hall, Englewood Cliffs, NJ, 1978), 2nd ed.
2. B. W. Marsden, Software-Prac. Exper. **14**, 659 (1984).
3. A. Hac, Software-Prac. Exper. **14**, 696 (1984).
4. G. Booch, *Software Engineering With Ada* (Benjamin/Cummings, Menlo Park, CA, 1987), 2nd ed., Chap. 16.
5. W. K. Erickson, Proc. AIAA/ACM/NASA/IEEE Computers Aerospace V Conference, Long Beach, CA (October 1985), p. 33.
6. G. Booch, IEEE Trans. Software Eng. **SE-12**, 211 (1986).
7. J. D. Laird, R. L. Victa, M. R. Koppes, and B. A. Burton, Proc. AIAA/ACM/NASA/IEEE Computers Aerospace V Conference, Long Beach, CA (October 1985), p. 285.
8. M. Narotam, C. Layton, and J. Slish, EDN (20 August 1987), p. 133.
9. H. Falk, Comput. Des. **27**, 55 (April 1988).
10. G. Chitwood, Def. Comput. **1**, 32 (July–August 1988).

# Modeling superconducting networks containing Josephson junctions by means of PC-based circuit simulation software

James A. Blackburn
*Department of Physics and Computing, Wilfrid Laurier University, Waterloo, Ontario N2L 3C5, Canada*

H. J. T. Smith
*Department of Physics, University of Waterloo, Waterloo, Ontario N2L 3C5, Canada*

*(Received 30 January 1990; accepted 18 May 1990)*

Software packages are now available with which complex analog electronic circuits can be simulated on desktop computers. Using Micro Cap III it is demonstrated that the modeling capabilities of such software can be extended to include *superconducting* networks by means of an appropriate equivalent circuit for a Josephson junction.

## INTRODUCTION

Superconducting circuits, containing Josephson devices, inductances, capacitors, and resistors, have many important practical applications.[1,2] These include SQUID magnetometers, high-speed superconducting computer elements, and voltage standards. The usual procedure for predicting the behavior of such circuits has been to solve the corresponding sets of nonlinear differential equations numerically. However, as we shall demonstrate, *superconducting electronics* can be included within the modeling capabilities of presently available circuit simulation software, and this provides a powerful and flexible alternative method of analysis.

With the advent of computer-aided engineering (CAE) software, analog circuits can be simulated on a computer before a hardware prototype is constructed. A well-known mainframe oriented software package is SPICE,[3] which was developed at UC Berkeley in the 1970s. The appearance of high-performance personal computers based on the 80386, and most recently 80486, chips has made CAE simulation of relatively large circuits feasible on desktop machines. Micro Cap III[4] (which was selected for the present work) is a leading simulation package for use on PC's. It has an extensive library of standard devices. Each specific component, such as a 2N2222 transistor or an LM741 op–amp, is modeled so as to replicate accurately that device's static and dynamic characteristics. As will be shown, this library can be extended by creating an equivalent circuit for a Josephson junction.

## I. JOSEPHSON JUNCTION SIMULATION

The circuit for simulating a current-biased noncapacitive Josephson device is shown in Fig. 1. The principal elements are an operational amplifier (op–amp) and a voltage-controlled oscillator (VCO). MicroCap III does not provide a VCO in its component library, and so a separate macro, described below, was designed for this purpose.

The VCO shown schematically in Fig. 2 contains three separate submacros:

(1) SPDT—a voltage-controlled switch set to act as a zero crossing detector; this is formed by combining two voltage-controlled single-pole/single-throw switches provided within MicroCap III.

(2) $X$—a simple voltage multiplier created from a voltage-controlled voltage source in MicroCap III.

(3) SINECONV—a four-diode triangle-to-sine wave converter[5] as shown in Fig. 3. This type of sine converter possesses the important attribute of not introducing any phase shift in the waveform.

The input voltage to the VCO is applied to PIN 1, and then is passed to the control terminal of the first SPDT (which enables the VCO to handle both positive and negative input voltages) and to the multiplier. The action of the circuit may be followed by assuming for the moment that $V_{in} > 0$. Suppose the present state of the circuit is as indicated in the schematic. One input to the multiplier is 8 V, the other is $V_{in}$. The multiplier output is thus $+8 \times V_{in}$, and so

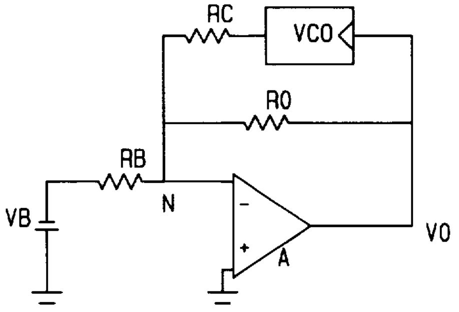


FIG. 1. Circuit for simulating a current-biased resistively shunted Josephson junction. The polarity shown for $V_b$ is required for positive equivalent bias current. Note that in this and subsequent figures, Micro Cap III represents $V_b$ as $VB$, $R_b$ as $RB$, etc.