# A common time base for parallel jobs in a distributed and embedded real-time system

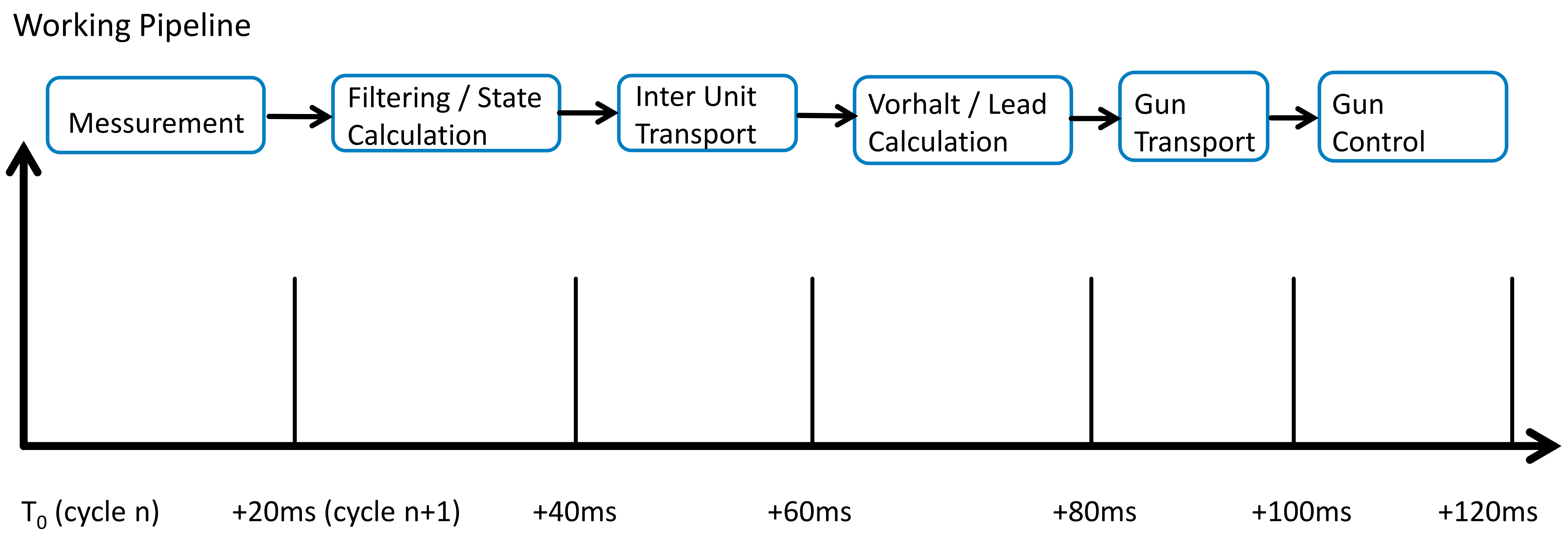
A common time base porgrammed in Ada for a Air Defence System.

## Mission accomplished

## The air defense problem: Calculate a State of an air vehicle.

- A fire control system measures an air vehicle position and calculates a track (velocity and acceleration  State) of that air vehicle for the lead calculation of the gun or the designation of the missiles.
- A common time base is evidently important for a measurement of the position of the air vehicle.
- There are two possible algorithms for the calculation of the velocity and acceleration of an air vehicle :
  - One algorithm measures the positions and the exact time of measurement (time-stamped). The time-stamp mechanism is almost fully a software solution (e.g. NTP).
  - The other way is to use a clock-base cyclic measurement system (like a clocked CPU). The clock-based mechanism is a mixture of hardware and software.
- Both mechanisms have their benefits. We chose both, clock-based for internal time-base data and time-stamped when data leaves our system.

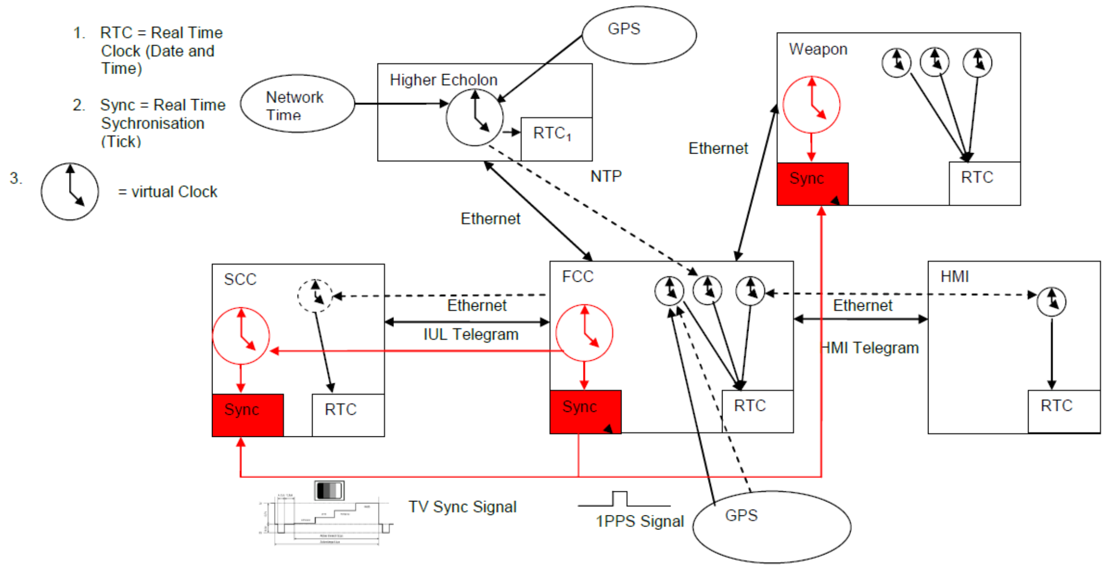
## The air defense problem: System Diagram.



## Hard- and Software Architecture

### The three Ada topics in this presentation

1. RTC: Real-Time Clock (Date and Time)
2. Synchronisation: Real-Time Synchronisation (Task synchronisation (and schedular) and Tick Counter)
3. Virtual Clocks.



## Real Time Clock (RTC)

- The hardware RTC is used on every computer node (similar to personal computer RTC module). The used CPU boards in the Fire Control Computer node (FCC) and the Sensor Control Computer node (SCC) didn’t have battery buffered RTC module (because of embedded CPU hardware).
- The GUI computer node (HMI) is the only CPU board which has a buffered RTC module.
- The RTC clock is accessed from Ada Programs via the predefined package Ada.Calendar. But there is a large disadvantage for embedded systems: there is no time/date set method in Ada.Calendar! That’s makes sense for main frame or personal computer system with multiple user accounts, but not for embedded system with anonymous user (or God).
- We solved that problem with a new package Adjustable_Calendar (It was developed in Ada83 more than 25 years ago. Today, it would be a sub-package of Ada.Calendar → Ada.Calendar.Adjustable (e.g.)).

# Adjustable_Calendar

```
WITH Ada.Calendar;

PACKAGE adjustable_calendar IS

   -- ---------------------------------------------------------
   -- Extended Calendar Time format                         --
   -- ---------------------------------------------------------
   SUBTYPE hour_number IS Integer RANGE 0 .. 23;
   SUBTYPE minute_number IS Integer RANGE 0 .. 59;
   SUBTYPE second_number IS Integer RANGE 0 .. 59;
   SUBTYPE msec_number IS Integer RANGE 0 .. 999;

   TYPE extended_time IS RECORD
      year   : Ada.Calendar.Year_Number;
      month  : Ada.Calendar.Month_Number;
      day    : Ada.Calendar.Day_Number;
      hour   : hour_number;
      minute : minute_number;
      second : second_number;
      msec   : msec_number;
   END RECORD;

   -- ---------------------------------------------------------
   -- Convert the adjustable Time to the predefined Time    --
   -- ---------------------------------------------------------
   FUNCTION to_calendar_time (item : time) RETURN Ada.Calendar.Time;

   -- ---------------------------------------------------------
   -- Convert the predefined Time to the adjustable Time    --
   -- ---------------------------------------------------------
   FUNCTION to_adjustable_time (item : Ada.Calendar.Time) RETURN time;

   -- ---------------------------------------------------------
   -- Returns the current time                              --
   -- ---------------------------------------------------------
   FUNCTION clock RETURN time;

   -- ---------------------------------------------------------
   -- Sets the current time                                 --
   -- ---------------------------------------------------------
   PROCEDURE set_clock (item : IN time);

END adjustable_calendar;
```

## Adjustable_Calendar (mechanism)

- The Adjustable_Calendar mechanism is quite simple and fully programmed in Ada!
  - The set time method calculates the difference between the underlying time (Ada.Calendar) and the set time. This difference is stored in the Object and will be used every time when clock is called to calculate the time!
  - Ada.Calendar can be set too BUT that needs compiler specific interfaces or definition (same for checking time representation). It was chosen a subpackge definition to encapsulate that internals!

```
FUNCTION clock (obj : IN object) RETURN time IS
BEGIN
   RETURN Ada.Calendar.Clock + obj.time_diff;
END clock;
PROCEDURE set_clock
     (obj           : IN OUT object;
      operator_time : IN     time)  IS
BEGIN
   obj.time_diff := operator_time - Ada.Calendar.Clock;
END set_clock;
```

## A task synchronizing and task scheduler with hardware signal (interface)

- The synchronization with a hardware signal is done by a special hardware and some driver packages written in Ada!
  - The special hardware board generates an interrupt whenever a synchronization signals is recognized (every 20ms)!
  - A low-level driver set-up the board and waits on the synchronization interrupts.
  - A higher level driver provides a high level language method for waiting on that synchronization event:

```
PROCEDURE wait(for_max_timeout : in Duration; success : OUT Boolean);
```

  - The task scheduler provides a blocking and non-blocking method for other task to wait on a synchronization event (or on a multiple of a synchronization event (every second/forth/tenth/50th event).

```
TYPE ticks IS (tick_20ms, tick_40ms, tick_80ms, tick_200ms, tick_1s);
PROCEDURE wait_for (tick : IN ticks);
PROCEDURE wait_for (tick : IN ticks; success : OUT Boolean); -- timeout was set in the initialize method
```

  - The task scheduler also provides a blocking and non-blocking method for other task to wait on a phased synchronization event (the caller will be blocked until some definable milliseconds or microseconds after synchronization events). Up to 5 phases could be defined because of the hardware limitation.

```
TYPE phase_number IS (phase1, phase2, phase3, phase4, phase5);
TYPE phase_record IS RECORD
   time    : Duration := 0.0;
   timeout : Duration := 0.0;
   used : Boolean := False;
END RECORD;
PROCEDURE wait_for (phase : IN phase_number);
PROCEDURE wait_for (phase : IN phase_number; success : OUT Boolean);
```

## A task synchronizing and task scheduler with hardware signal (task 1)

```
TASK BODY a_tick_controller IS
   PROCEDURE rtc_activation IS SEPARATE;
   PROCEDURE await_tick IS SEPARATE;
BEGIN
   forever:
   LOOP
      SELECT
         ACCEPT start DO
            rtc_activation;
         END start;
      OR
         TERMINATE;
      END SELECT;

      control_cycle:
      LOOP
         BEGIN
            SELECT
               ACCEPT stop DO
                  rtc_shutdown;
               END stop;
               EXIT control_cycle;
```

```
ELSE
  await_tick;
  the_cycle_count := rtc.current_cycle;
  cycle_count.set (cycle_count => the_cycle_count);

  WHILE a_tick_20ms'count > 0 LOOP
      ACCEPT a_tick_20ms;
  END LOOP;
  --
  -- Releases all tasks waiting for a 20 ms tick

  IF the_cycle_count MOD 2 = 0 THEN
      WHILE a_tick_40ms'count > 0 LOOP
          ACCEPT a_tick_40ms;
      END LOOP;
      --
      -- Releases all tasks waiting for a 40 ms tick

      IF the_cycle_count MOD 4 = 0 THEN
          WHILE a_tick_80ms'count > 0 LOOP
             ACCEPT a_tick_80ms;
          END LOOP;
```

## A task synchronizing and task scheduler with hardware signal (task 2)

```
IF the_cycle_count MOD 4 = 0 THEN
    WHILE a_tick_80ms'count > 0 LOOP
       ACCEPT a_tick_80ms;
    END LOOP;
     --
     -- Releases all tasks waiting for a 80 ms tick

END IF;    -- cycle_count MOD 4  = 0

IF the_cycle_count MOD 10 = 0 THEN
    WHILE a_tick_200ms'count > 0 LOOP
       ACCEPT a_tick_200ms;
     END LOOP;
     --
     -- Releases all tasks waiting for a 200 ms tick

END IF;    -- cycle_count MOD 10 = 0
```

```
         IF the_cycle_count MOD 50 = 0 THEN
            WHILE a_tick_1s'count > 0 LOOP
               ACCEPT a_tick_1s;
            END LOOP;
            --
            -- Releases all tasks waiting for a 1 s tick
         END IF;   -- cycle_count MOD 50 = 0
      END IF;      -- cycle_count MOD 2  = 0
    END SELECT; -- Accepts stop or waits for tick
  EXCEPTION
    WHEN error: OTHERS =>
      error_message_io.report_exception
        (caller => "Tick_Controller control_cycle loop",
         error => error);
  END;
 END LOOP control_cycle;
END LOOP forever;
EXCEPTION
   WHEN error: OTHERS =>
      error_message_io.report_exception
         (caller => "Tick_Controller", error => error);
END a_tick_controller;
```

## Await Hardware Tick (interrupt handler)

- The method await tick could be seen as a simple wait on an entry which will be set by an interrupt routine.
- There are much checks and test in reality to ensure that no cycle is lost or to detect such a lost.
- It is a very easy job to write a interrupt routine in Ada! You just have to write a protected type and assign a interrupt:

```
PROTECTED TYPE prot_interrupt (int_id : Ada.Interrupts.Interrupt_ID) IS
   ENTRY wait_for_interrupt ; -- Entry for client(s) waiting until expected interrupt is received.
   PROCEDURE interrupt_procedure; -- This procedure is only "attached" to the interrupt vector (below).
   PRAGMA ATTACH_HANDLER (interrupt_procedure, int_id); -- Assignment of the interrupt vector.
PRIVATE
   interrupt_received : Boolean := False;
END prot_interrupt;

PROTECTED BODY prot_interrupt IS
   ENTRY wait_for_interrupt WHEN interrupt_received IS
   BEGIN
      interrupt_received := False;
   END wait_for_interrupt;
   PROCEDURE interrupt_procedure IS
   BEGIN
      interrupt_received := True;
   END interrupt_procedure;
 END prot_interrupt;
```

## Await Hardware Tick (calling interrupt handler)

- The interrupt handler can be called like every protected type:
  - Blocking
  - Timed
  - Conditional
- We are calling such interrupt routine either blocking or timed. The tick interrupt handler is called timed/non blocking because we don't want that our whole system stands if we have an interrupt problem:

```
PROCEDURE wait_for_interrupt (timeout   : IN duration := -1.0;
                              timed_out : OUT boolean) IS
BEGIN
   IF timeout < 0.0 THEN
      interrupt_handler.wait_for_interrupt;
   ELSE
     SELECT
        interrupt_handler.wait_for_interrupt;
        timed_out := false;
     OR
        DELAY local_timeout;
        timed_out := true;
     END SELECT;
   END IF;
END wait_for_interrupt;
```

## Representation Clauses (1)

- Representation Clauses are very useful Ada constructs for low level programming.
- There are representation clauses for size, alignment, value,  and record layout  of a type/object:

```
TYPE Medium IS RANGE 0 .. 65_000;
FOR Medium'Size USE 2*8;
FOR Medium'Alignment USE 2;


TYPE Short IS DELATA 0.01 RANGE -100.0 .. 100.0;
FOR Short'Size USE 15;


TYPE Mix_Code IS (Add, Sub, Mul, Lda, Sta, Stz);
FOR Mix_Code USE (Add => 1, Sub => 2, Mul => 3, Lda => 8, Sta => 24, Stz => 33);
```

## Representation Clauses (2)

```
TYPE cfg_timer_sync IS RECORD
     reserved_4                : bit_field (1 .. 3);
     sync_1_in_sel             : sync_1_in_selection;
     sync_1pps_falling_edge    : Boolean;
     sync_dt60_falling_edge    : Boolean;
     sync_50_falling_edge      : Boolean;
     lock_phase_lange_window   : Boolean;
     phase_filter_enable       : Boolean;
     temp_comp                 : Boolean;
     check_phase_and_f_delta   : Boolean;
     tvs_direct                : Boolean;
     source                    : Integer RANGE 0 .. 7;
  END RECORD;
```

```
FOR cfg_timer_sync USE RECORD AT MOD 2;
     reserved_4              AT 0 RANGE 0 .. 2;
     sync_1_in_sel           AT 0 RANGE 3 .. 4;
     sync_1pps_falling_edge  AT 0 RANGE 5 .. 5;
     sync_dt60_falling_edge  AT 0 RANGE 6 .. 6;
     sync_50_falling_edge    AT 0 RANGE 7 .. 7;
     lock_phase_lange_window AT 0 RANGE 8 .. 8;
     phase_filter_enable     AT 0 RANGE 9 .. 9;
     temp_comp               AT 0 RANGE 10 .. 10;
     check_phase_and_f_delta AT 0 RANGE 11 .. 11;
     tvs_direct              AT 0 RANGE 12 .. 12;
     source                  AT 0 RANGE 13 .. 15;
  END RECORD;
  FOR cfg_timer_sync'size USE 16;

IF a_cfg.sync_50_falling_edge THEN
   a_cfg.phase_filter_enable := True;
END IF;

a_cfg.phase_filter_enable := a_cfg.sync_50_falling_edge;
```

## Representation Clauses (3)

- A comparison with C/C++:
- Ada let you access direct the defined fields! There is no need for error prone  shift and mask operation!
- The compiler checks the layout for consistency!
- In Ada the definition needs time but not the use of it!
- The layout is driven directly from the hardware (registers). That’s a good documentation too.

## Questions?