Desarrollo de software para CubeSat
Apéndices

Simón González - Pablo Yaniero

7 de Marzo de 2014

Contenido

Índice de figuras

Índice de cuadros

Apéndices

1.1. Apéndice A: Aplicaciones de EMS

El microkernel a usar en los componentes del satélite propone un diseño en el que el programa se describe como un conjunto de aplicaciones que ejecutan en forma independiente. El microkernel se encarga de compartir el CPU entre los mismos permitiendo un enfoque modular en el que la lógica de cada aplicación puede ser analizada y verificada en forma separada. La presente sección describe en detalle las aplicaciones del módulo de gestión de energía.

La tabla 1.1 lista las aplicaciones y los elementos con los que cada una interactúa. La columna Entradas indica variables compartidas y recursos que la aplicación lee. La columna Salidas indica variables compartidas y recursos que la aplicación escribe.

Aplicación	**Entradas**	**Salidas**
batteries_manager		tension_batteries beacon_data
beacon_data_collector	REC_EMS_APPLICATION _BEACON_DATA_COLLECTOR	beacon_data user_message user_message_counter
clock	REC_EMS_APPLICATION_CLOCK RESOURCE_IMMP_REQUEST RESOURCE_IMMP_REPLY	
energy_modes_manager	tension_threshold_ascend tension_threshold_descend tension_batteries energy_mode	energy_mode desired_states RESOURCE_IMMP_REQUEST RESOURCE_IMMP_REPLY
i2c_monitor	effective_states	RESOURCE_IMMP_REQUEST RESOURCE_IMMP_REPLY
mcs_on_off_command_processor	REC_EMS_APPLICATION _MCS_ON_OFF_COMMAND_PROCESSOR REC_RESPONSE_MANAGER effective_states	REC_AWAKE_MANAGER desired_states timer_counters timer_callbacks beacon_enable
modules_restorer	modules module_timeouts	retries
morse_beacon	energy_mode user_message	REC_MORSE_COMMUNICATOR
morse_communicator	beacon_data user_message tension_batteries beacon_enable REC_BEACON_SERVICE	user_message user_message_counter morse_unit
mppt	mppt_step	
on_off_manager	desired_states energy_mode REC_AWAKE_MANAGER	effective_states retries module_timeouts beacon_data REC_RESPONSE_MANAGER RESOURCE_IMMP_REQUEST RESOURCE_IMMP_REPLY
params_loader	energy_mode effective_states REC_EMS_APPLICATION _PARAMS_LOADER	tension_threshold_ascend tension_threshold_descend mppt_step RESOURCE_IMMP_REQUEST RESOURCE_IMMP_REPLY

Continúa en la siguiente página

Aplicación	Entradas	Salidas
payload_on_off_command_processor	effective_states REC_EMS_APPLICATION_PAYLOAD - _ON_OFF_COMMAND_PROCESSOR REC_RESPONSE_MANAGER	desired_states timer_counters timer_callbacks RESOURCE_IMMP_REQUEST RESOURCE_IMMP_REPLY REC_AWAKE_MANAGER
roger_receiver	REC_EMS_APPLICATION _ROGER_RECEIVER	REC_MORSE_COMMUNICATOR
telemetry_report	effective_states	RESOURCE_IMMP_REQUEST RESOURCE_IMMP_REPLY
timer_loop	timer_callbacks	timer_counters
immp_rx		REC_EMS_APPLICATION _PAYLOAD_ON_OFF_COMMAND_PROCESSOR REC_EMS_APPLICATION _BEACON_DATA_COLLECTOR REC_EMS_APPLICATION _PARAMS_LOADER REC_EMS_APPLICATION _PARAMS_LOADER_WAKEUP REC_EMS_APPLICATION_CLOCK REC_EMS_APPLICATION _MCS_ON_OFF_COMMAND_PROCESSOR REC_EMS_APPLICATION _ROGER_RECEIVER
immp_tx	RESOURCE_IMMP_REQUEST RESOURCE_IMMP_TX_RESPONSE	RESOURCE_IMMP_REPLY

Tabla 1.1. Listado de aplicaciones de EMS.

A continuación se enumeran las aplicaciones diseñadas para el EMS brindando mayor detalle de las responsabilidades que cada una cumple y su funcionamiento.

1.1.1. MPPT (mppt)

Maximiza la transferencia de potencia entre los paneles solares y el resto del sistema.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Encontrar la tensión óptima de trabajo de los paneles solares en cada momento.
- Cambiar entre mppt por software y mppt por hardware según el valor de potencia obtenido.
- Reprogramar los puertos de entrada y salida periódicamente.

`Funcionamiento`

Existen tres instancias independientes de este aplicación, una para cada dispositivo MPPT. La aplicación busca maximizar la potencia generada por los paneles solares buscando el valor de tensión óptimo en dicha curva. Para ello se aplica el algoritmo de "Hill Climbing":

1. Se fija deltaV positivo.
2. Dada la tensión actual T0 se calcula la potencia actual P0. La potencia se calcula a partir de la intensidad y la tensión por medio de la relación P = V . I.
3. Se elige una nueva tensión T1 tal que T1 = T0 + deltaV, se asigna T1 al MPPT y se determina la potencia para esta nueva tensión, denominada P1.
4. Si Po <= P1, deltaV continúa siendo positivo, sino deltaV se vuelve negativo.
5. Se vuelve al paso 2

Existen dos modos de funcionamiento.

- `Hardware` En modo hardware, la tensión es controlada por un dispositivo mppt implementado por hardware.
- `Software` En modo software, la tensión es controlada por la aplicación.

La aplicación comienza en modo software. En cada iteración se lee el nivel de tensión de trabajo actual y la intensidad de la corriente. A partir de estos valores se calcula la potencia actual. Con los valores de potencia y tensión se define el modo de trabajo. Si esta activo el modo software pero la potencia no alcanza un valor mínimo, se cambia a modo hardware por al menos cinco segundos. Si en cambio el voltaje no alcanza un mínimo suficiente, se pasa a modo hardware por tiempo indeterminado. Una vez en modo hardware, se mantiene en ese modo hasta que el voltaje y la potencia superen los mínimos impuestos, volviendo entonces a modo software.

Si se define el modo de trabajo por software, a continuación se ejecuta el algoritmo de Hill Climbing. Primero la potencia es recalculada a partir de nuevas mediciones de tensión e intensidad. Luego se modifica la tensión de trabajo según el paso establecido y se vuelve a recalcular la potencia. Según sea la relación entre las dos potencias calculadas antes y después del paso, se define la dirección del siguiente.

Pseudocódigo: **mppt**

```
mppt () {
    direction = RIGHT; //1 indica continuar en la direccion actual, -1 invierte la direccion
    currentMode = HARDWARE;//Inicia en modo hardware

    switch2SW(); //Se cambia a modo software
    for (;;) {

        updateMode();
        if(currentMode == SOFTWARE){
            step = mppt_step //mppt_step puede cambiar por orden de tierra
            mpptStep(); //Se modifica la tensión de trabajo
        }

        sleep(0.1);

    }
}

updateMode(){
    tension = hal_read_tension();
    current = hal_read_current();
    power = tension * current;
    if (currentMode == HARDWARE && tension >= INITIAL_TENSION_LEVEL && power >=
        epsilon){
        switch2SW();
    }else if (currentMode == SOFTWARE && power < POWER_EPSILON){
        switch2HW();
        sleep(5);
    }else if(currentMode == SOFTWARE && tension < INITIAL_TENSION_LEVEL){
        switch2HW();
    }
}

mpptStep() {
    step = mppt_step;
    step = (direccion == LEFT) ? step : -step;
    tension = hal_read_tension();
    current = hal_read_current();
    before_power = tension * current;
    updateTension(step); //Cambia la tension de trabajo
    tension = hal_read_tension();
```

```
    current = hal_read_current();
    after_power = tension * current;
    direccion = after_power >= before_power ? 1 : -1;
}
```

1.1.2. Gestor de baterías (batteries_manager)

Lee periódicamente el nivel de tensión de las baterías.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Leer el nivel de tensión y de energía de las baterías periódicamente.
- Reprogramar las protecciones de las baterías periódicamente.
- Reprogramar las protecciones de las baterías en caso que la energía leída esté fuera de rango.

`Funcionamiento`

El estado energético del satélite se determina a partir del nivel de tensión de las baterías. La aplicación lee cada un segundo el nivel de tensión de las baterías mediante un conversor ADC y almacena el valor obtenido en la variable de estado `tension_batteries`. El nivel de energía de cada batería es previamente obtenido a partir de las protecciones de las mismas. Si el nivel de energía leído no se encuentra en el rango esperado se reprograma la protección correspondiente y se descarta la lectura de tensión. Un contador es utilizado para calcular el tiempo aproximado desde la última reprogramación de las protecciones, reprogramando al alcanzar el tiempo requerido. Al final de cada iteración se actualizan los valores de tensión y energía en la baliza morse.

Pseudocódigo: **batteries_manager**

```
batteries_manager() {
  seconds_until_reprogram = 0;
  for (;;) {

    //Se lee el nivel de energía de las baterías
    for (i = 0 .. 3) {
      energy_battery[i] = read_energy_battery(i);
      if (energy_battery[i] < ENERGY_MIN || energy_battery[i] > ENERGY_MAX){
        reprogram_battery(i);
      }
    }

    //Se lee el nivel de tensión
    tension = hal_read_tension_batteries();
    if(tension >= TENSION_MIN && tension <= TENSION_MAX){
      set_tension_batteries(tension);
    }

    //Se actualizan los datos de la baliza morse
    update_beacon_data(tension, energy_battery);

    sleep(SLEEP_TIME);

    //Verificar tiempo restante para reprogramar
    seconds_until_reprogram = seconds_until_reprogram + SLEEP_TIME;
    if(seconds_until_reprogram >= REPROGRAM_PERIOD){
      seconds_until_reprogram = 0;
      reprogram_battery(NUMBER_OF_BATTERIES);
    }
```

```
  }
}
```

1.1.3. Gestor de modos de energía (energy_modes_manager)

Determina el modo de energía actual del sistema.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Determinar el modo de energía actual del sistema.
- Verificar el correcto despliegue de las antenas.
- Actualizar el vector de estados deseados según corresponda al modo de energía.

`Funcionamiento`

Al iniciar el sistema, la aplicación inicializa el modo de energía en `DEPLOY_WAIT` y verifica periódicamente que las antenas hayan sido desplegadas. Una vez las antenas estén listas o bien hayan pasado más de 24 horas desde el inicio, se realiza una espera de 15 minutos y se prosigue a pasar a modo `RECOVERY` comenzando entonces a monitorizar el nivel de tensión de las baterías. Cada 30 segundos se controla la tensión presente en la variable de estado `tension_batteries` y se determina el modo de energía, el cual es almacenado en la variable de estado `energy_mode`. En modo `RECOVERY`, cuando la tensión supera un umbral determinado, se cambia al estado `SAFE` declarando en el vector de estados deseados que los módulos I^2C, MCS, COMM1 y COMM2 deben estar encendidos. En modo `SAFE`, cuando la tensión baja por debajo de otro umbral, se vuelve a estado `RECOVERY` y se declara en el vector de estados deseados que los módulos ADCS, PAYLOAD, SBAND1 y SBAND2 deben estar apagados.

Pseudocódigo: **energy_modes_manager**

```
energy_modes_manager() {
  waiting_for_antennas_deployment();

  for (;;)  {
      energy_modes_manager_control();
      sleep(30);
  }
}

waiting_for_antennas_deployment(){
  //Verificar si las antenas fueron desplegadas
  bool deploy_ready = hal_get_antennas_deployed();
  energy_mode = DEPLOY_WAIT;

  //Si las antenas no fueron desplegadas
  if (!deploy_ready){
    secondsInDeploy = 0;
    //Busy waiting esperando por el despliegue de las antenas o
    //hayan pasado 24 horas
    deploy_ready = hal_get_antennas_deployed();
    while (!deploy_ready && secondsInDeploy <= 24 * 60 * 60){
        sleep(30 seconds);
        secondsInDeploy = secondsInDeploy + 30;
        deploy_ready = hal_get_antennas_deployed();
    }
    //Si pasaron las 24 horas sin desplegar las antenas
    if (!deploy_ready){
      //Comunicar evento a MCS
```

```
      send(MCS, EVENT_EMS_MCS_DEPLOY_TIMEOUT)
    }

    //Se aguardan 15 minutos antes de continuar
    sleep(15 * 60);
  }

  energy_mode = RECOVERY;
}

energy_modes_manager_control(){
    if(energy_mode == RECOVERY &&
        tension_batteries  > tension_threshold_ascend{
      //Transición de RECOVERY a SAFE
      energy_mode = SAFE;

      //Encender módulos que deben estar encendidos en SAFE
      desired_state [MODULE_MCS] = ON;
      desired_state [MODULE_I2C] = ON;
      desired_state [MODULE_COMM1] = ON;
      desired_state [MODULE_COMM2] = ON;
  }else if(energy_mode == SAFE
          && tension_batteries < tension_threshold_descend{
    //Transición de SAFE a RECOVERY
    energy_mode = RECOVERY;

    //Apagar módulos sin encendido automático.
    desired_state [MODULE_ADCS] = OFF;
    desired_state [MODULE_PAYLOAD] = OFF;
    desired_state [MODULE_SBAND1] = OFF;
    desired_state [MODULE_SBAND2] = OFF;
}
```

1.1.4. Gestor de encendido y apagado (on_off_manager)

Se encarga de encender y apagar módulos según sea requerido por el vector de estados deseados.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Detectar y registrar los módulos en Falla.
- Encender y apagar módulos según corresponda.
- Mantener el vector de reintentos.

`Funcionamiento`

La aplicación verifica periódicamente el estado de encendido de cada uno de los módulos del sistema. Inicialmente se corrobora si el módulo sufrió un cortocircuito. En caso afirmativo se apaga el módulo y se notifica a control principal. De no detectarse falla, se procede a verificar el estado de encendido actual. Si el módulo se encuentra apagado o en falla, se verifica si debe ser encendido. Para ello se corrobora que el estado deseado sea encendido, que cumpla con las precondiciones de encendido del módulo y que en caso de estar en falla, su contador de reintentos sea positivo. Al encender un módulo, si se detecta un cortocircuito se vuelve a intentar encenderlo pudiendo repetir el proceso la cantidad de veces indicadas en el vector de reintentos. Si antes de agotar los reintentos el módulo se enciende sin problemas, se actualiza el vector de estados efectivos con el nuevo estado. De lo contrario se envía una notificación y el módulo es marcado como en falla. Si el módulo se encuentra originalmente encendido, se verifica si debe

ser apagado. Para ello se corrobora que el estado deseado sea apagado,o que no cumpla con alguna de las precondiciones de encendido del módulo.

Pseudocódigo: **on_off_manager**

```
on_off_manager() {

  //Aguardar hasta que las antenas hayan sido desplegadas
  while (energy_mode == DEPLOY_WAIT) {
          sleep(1);
  }

  lock(REC_AWAKE_MANAGER);
  set_buffer (REC_AWAKE_MANAGER, 0);

  for (;;)  {

    ret_val  = read(REC_AWAKE_MANAGER, TIMEOUT_GESTOR);
    timeout = (ret_val == ERROR_TIMEOUT);
    reset_buffer (REC_AWAKE_MANAGER);

    previous_mcs_state =  effective_state [MODULE_MCS];

    //Verificar  cada módulo y encenderlo/apagarlo si corresponde
    for modulo in {I2C, MSC, COM1, COM2, ADCS, PAYLOAD, SBAND1, SBAND2} {
        verify_module(modulo);
    }

    current_mcs_state =  effective_state [MODULE_MCS];
    if (previous_mcs_state == OFF && current_mcs_state == ON) {
     /* Si MCS fue encendido, solicitar a params_loader enviar pedido de
     actualización de parámetros a MCS. */
     params_loader_request_parameters();
    }

    if (!timeout) {
      write(REC_RESPONSE_MANAGER, 0);
    }

  }

  // Nunca se libera REC_AWAKE_MANAGER (no es necesario)
  unlock(REC_AWAKE_MANAGER);
}


verify_module(module) {
  if ( effective_state [module] == OFF || effective_state[module] == FAIL) {
    //Encender si corresponde
    turn_on_if_must(module);
  } else if ( effective_state [module] == ON) {
    //Apagar si corresponde
    turn_off_if_must (module);
  }
}

turn_on_if_must(module) {
  //Verificar  si  el  módulo debe ser encendido
  if (must_turn_on(module) {
```

```
    //El módulo debe ser encendido

    //Encender el módulo y verificar si esta en cortocircuito.
    //En caso de estarlo reintentar encenderlo el numero de veces
    //indicado en retries.
     short_circuit = true;
    while (retries[module] > 0 && short_circuit) {
      //Encender módulo
      power_enable(module);

       retries[module]--;
      sleep (WAIT_TIME);
       short_circuit = fault(module);
    }

    if (! short_circuit ) {
      //Si no fallo al encender se restablece el número de reintentos
       retries[module] = 5;
       effective_states [module] = ON;
    } else {
      //En caso de falla se avisa a MCS
       effective_states [module] = FAIL;
      module_timeouts[module] = TIMEOUT;
       send_failure_notification_to_mcs (MCS, module);
    }
  }
}


// Devuelve true sii realmente apagó
turn_off_if_must (module) {
  //Verificar si el módulo debe ser apagado
  if (must_turn_off(module)) {
    if (module == MCS) {
       //MCS debe ser notificado antes de ser apagado.
       send_turn_off_notification_to_mcs (MCS);
       sleep(1);
    }

    //Apagar módulo
    power_disable(module);
     effective_states [module] = OFF;
    return true;
  }

  return false;
}

bool must_turn_on(module) {
  //Verificar precondiciones de encendido
  if (! verify_turn_on_preconditions(module)) {
    //No se cumplen las precondiciones
    return false;
  }

  if ( effective_state [module] == FAIL
      && retries[module] == 0) {
    //El módulo se encuentra en falla y no tiene mas reintentos
```

```
    return false;
  }

  if (automatic_turn_on(module)) {
    //Se cumple una condición de encendido automático del módulo
    //entonces se ignora el estado deseado.
    return true;
  }

  //El estado deseado determina si se debe encender o no.
  return (desired_state[module] == ON)? true : false;
}

bool must_turn_off(module) {

  //Verificar si el módulo esta en cortocircuito
  if (fault(module)) {

    if (module != MODULE_MCS) {
      //MCS debe ser notificado
      send_short_circuit_notification_to_mcs (module);
    }

    //Los modulos en falla deben ser apagados.
    return true;
  }

  //Verificar las precondiciones de apagado.
  if ( verify_turn_off_preconditions (module)) {
    //No puede ser apagado.
    return false;
  }

  //Verificar las precondiciones de encendido, si alguna deja de cumplirse
  //se debe apagar
  if (verify_turn_on_preconditions(module) == false) {
    return true;
  }

  //El estado deseado es apagado
  return (desired_state[module] == OFF)? true : false;
}
```

1.1.5. Cargador de parámetros (params_loader)

Recibe y carga en memoria los parámetros de funcionamiento del módulo provistos por MCS.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Escuchar por mensajes provenientes de MCS conteniendo una actualización de parámetros.
- Almacenar los parámetros recibidos en las variables de estado que corresponda.

`Funcionamiento`

La aplicación se mantiene a la espera de la recepción de un mensaje desde MCS. En caso de comprobarse que un mensaje recibido es una actualización de parámetros, los datos recibidos son almacenados en las

variables de estado correspondientes y se procede a aguardar por la llegada del próximo mensaje. De no ser el mensaje esperado, el mismo se descarta.

Pseudocódigo: **params_loader**

```
params_loader() {
  lock(REC_EMS_APPLICATION_PARAMS_LOADER);

  for (;;) {
    params_loader_control();
  }

}

void params_loader_control() {

  //Escuchar por un mensaje de actualización de parámetros
  response = receive_parameters_response();

  if(response.success){
    //Actualizar las variables de estado con los nuevos valores para los parámetros
    update_params(response);
  }
}

receive_parameters_response() {
  message.succes = FALSE
  set_buffer (REC_EMS_APPLICATION_PARAMS_LOADER, buffer);

  message_id = read(REC_EMS_APPLICATION_PARAMS_LOADER);
  if(message_id == MESSAGE_MCS_EMS_PARAMETER_RESPONSE){
    message.tension_threshold_ascend = buffer.tension_threshold_ascend;
    message.tension_threshold_descend = buffer.tension_threshold_descend;
    message.mppt_step = buffer.mppt_step;
    message.success = TRUE;
  }

  return message;
}

update_params(params) {
  tension_threshold_ascend = params.tension_threshold_ascend;
  tension_threshold_descend = params.tension_threshold_descend;
  mppt_step = params.mppt_step;
}


//Llamada por on_off_manager cuando MCS es encendido.
params_loader_request_parameters() {
  if(get_energy_mode() == SAFE && get_effective_state(MODULE_MCS) == ON){
    // Solicitar parámetros a MCS
    send(MCS, MESSAGE_EMS_MCS_PARAMETER_REQUEST);
  }
}
```

1.1.6. Procesador de comandos de encendido y apagado (mcs_on_off_command_processor)

Recibe los mensajes provenientes de MCS relativos al encendido y apagado de módulos.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Escuchar por mensajes provenientes de MCS conteniendo comandos de encendido y apagado.
- Interpretar el comando recibido e iniciar la acción solicitada.

`Funcionamiento`

La aplicación escucha por mensajes IMMP provenientes de MCS. Al recibir un mensaje se determina si pertenece a una de cuatro categorías y se procede a ejecutar cada una según sea el caso.

- Modificación del vector de estados deseados. Ante este mensaje primero se valida que el módulo y estado destino sean validos. Únicamente se acepta encender o apagar los módulos COMM1, COMM2, ADCS y PAYLOAD. Ningún módulo puede ser puesto en estado falla. En caso de ser valido, el vector de estados deseados es modificado con el nuevo valor recibido para el módulo y se notifica del cambio a la aplicación `on_off_manager`. Se aguarda entonces a que el `on_off_manager`efectúe una iteración en respuesta a la notificación, a fin de que encienda o apague el módulo destino antes de volver para escuchar el siguiente mensaje.
- Deshabilitar la portadora morse. En este caso se procede deshabilitar el uso de la portadora a través de la variable de estado `beacon_enable`. Un temporizador es entonces encendido para asegurar que la portadora vuelva a encenderse tras un tiempo prudencial.
- Habilitar la portadora morse. Se modifica la variable de estado `beacon_enable` para indicar que la portadora esta ahora habilitada.
- Reiniciar el satélite. Se accede directamente al `hal` para apagar y prender todo el satélite.

Pseudocódigo: **mcs_on_off_command_processor**

```
mcs_on_off_command_processor() {

   ret_val  = lock(REC_EMS_APPLICATION_MCS_ON_OFF_COMMAND_PROCESSOR);

   set_buffer (REC_EMS_APPLICATION_MCS_ON_OFF_COMMAND_PROCESSOR, input);

   for (;;) {
     //Leer y procesar mensages de MCS
     wait_and_process_message(input);
   }
}

wait_and_process_message(buffer) {
   msg = read(REC_EMS_APPLICATION_MCS_ON_OFF_COMMAND_PROCESSOR);

   if (msg.header.message_id == MESSAGE_MCS_EMS_SET_DESIRED_STATE) {
     //Cambiar el estado deseado de un módulo
      process_set_desired_state (msg);
   } else if (msg.header.message_id == MESSAGE_MCS_EMS_SYSTEM_REBOOT) {
     //Reiniciar el saté lite
     hal_reboot();
   } else if (msg.header.message_id == MESSAGE_MCS_EMS_SILENCE_BEACON) {
     //Silenciar el transmisor de la baliza morse
     process_silence_beacon(msg);
   } else if (msg.header.message_id == MESSAGE_MCS_EMS_RESTORE_BEACON) {
     //Restaurar el transmisor de la baliza morse
     process_restore_beacon(msg);
   }

   reset_buffer (REC_EMS_APPLICATION_MCS_ON_OFF_COMMAND_PROCESSOR);
}
```

```
process_set_desired_state (msg) {
  module_t module = msg->target_module;

  desired_state  = msg->desired_state;

  previous_state  =  effective_state [module];
  desired_state [module] = desired_state;

  // notificar  al  on_off_manager para que procese el cambio de estado deseado
  synch_with_manager();

  new_state =  effective_state [module];

  if  (module == MODULE_COMM1 && previous_state == ON && new_state == OFF) {
    /* COMM1 ha sido apagado. Iniciar el temporizador para encenderlo luego de 72 horas */
    set_timer_counter(TIMER_COMM1_RESTORE, COMM_RESTORE_TIMEOUT);
    set_timer_callback (TIMER_COMM1_RESTORE, comm1_restore_callback);
  }

  if  (module == MODULE_COMM2 && previous_state == ON && new_state == OFF) {
    /* COMM1 ha sido apagado. Iniciar el temporizador para encenderlo luego de 72 horas */
    set_timer_counter(TIMER_COMM2_RESTORE, COMM_RESTORE_TIMEOUT);
    set_timer_callback (TIMER_COMM2_RESTORE, comm2_restore_callback);
  }

  if  (module == MODULE_COMM1 && previous_state == OFF && new_state == ON) {
    /* COMM1 ha sido encendido, apagar temporizador */
    set_timer_counter(TIMER_COMM1_RESTORE, 0);
    set_timer_callback (TIMER_COMM1_RESTORE, 0);
  }

  if  (module == MODULE_COMM2 && previous_state == OFF && new_state == ON) {
    /* COMM2 ha sido encendido, apagar temporizador */
    set_timer_counter(TIMER_COMM2_RESTORE, 0);
    set_timer_callback (TIMER_COMM2_RESTORE, 0);
  }
}

process_silence_beacon(msg) {
  bool previous_state = get_beacon_enable();
  set_beacon_enable(FALSE);

  if  (previous_state  == TRUE) {
    set_timer_counter(TIMER_BEACON_RESTORE, BEACON_RESTORE_TIMEOUT);
    set_timer_callback (TIMER_BEACON_RESTORE, restore_beacon);
  }
}

process_restore_beacon(msg) {
  set_beacon_enable(TRUE);

  set_timer_counter(TIMER_BEACON_RESTORE, 0);
  set_timer_callback (TIMER_BEACON_RESTORE, 0);
}

synch_with_manager() {
  lock(REC_RESPONSE_MANAGER);
  write(REC_AWAKE_MANAGER);
```

```
   set_buffer (REC_RESPONSE_MANAGER);
   read(REC_RESPONSE_MANAGER);
   ret_val  = unlock(REC_RESPONSE_MANAGER);

}

comm1_restore_callback() {
   set_desired_state (MODULE_COMM1, ON);
   synch_with_manager();
}

comm2_restore_callback() {
   set_desired_state (MODULE_COMM2, ON);
   synch_with_manager();
}

restore_beacon() {
   set_beacon_enable(TRUE);
}
```

1.1.7. Procesador de comandos de encendido y apagado del Payload (payload_on_off_command_processor)

Recibe las solicitudes de encendido y apagado de los transmisores SBAND1 y SBAND2 provenientes del módulo PAYLOAD.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Escuchar por mensajes provenientes de PAYLOAD conteniendo comandos de encendido y apagado de los transmisores SBAND1 y SBAND2.
- Interpretar el comando recibido e iniciar la acción solicitada.

`Funcionamiento`

La aplicación aguarda por la llegada de un mensaje proveniente de PAYLOAD. Tras verificar que el módulo y el comando sea válido, se procede a corroborar el estado actual del transmisor destino. Si el comando es de encendido y el transmisor se encuentra apagado, se actualiza el vector de estados deseados y se notifica al `on_off_manager` del cambio, aguardando a que este encienda el transmisor antes de continuar. Si el transmisor es efectivamente encendido, se inicia un temporizador que lo apagará tras 100 segundos y finalmente se envía a PAYLOAD un mensaje confirmando el encendido del transmisor. De lo contrario un mensaje a PAYLOAD indicando error es enviado. En caso que el transmisor estuviera en falla, se envía a PAYLOAD un mensaje informando de que el transmisor no pudo ser encendido. Si en cambio ya estaba encendido, simplemente se reinicia el temporizador y se envía el mensaje de éxito. En el caso que el comando sea de apagado, se procede a modificar el vector de estados deseados y se invalida cualquier temporizador de apagado pendiente.

Pseudocódigo: **payload_on_off_command_processor**

```
payload_on_off_command_processor() {

   ret_val  = lock(REC_EMS_APPLICATION_PAYLOAD_ON_OFF_COMMAND_PROCESSOR);

   for (;;) {
      //Recibir mensaje desde el Payload
      message = receive_payload_message();
      payload_on_off_command_processor_control(message);
   }
```

```
}

payload_on_off_command_processor_control(payload_message_t message){

  if(message.success == TRUE && message.command == TURN_ON){
    //Verificar si el modulo es un transimsor de banda S
    if(message.module == MODULE_SBAND1 || message.module == MODULE_SBAND2){

      if( effective_state [message.module]) == ON){

        //El transmisor ya esta encendido, reiniciar el temporizador
        timer = get_sband_timer(message.module);
        if (timer != TIMER_ERROR) {
          set_timer_counter(timer, 100);
        }

        //Enviar confirmacion de encendido
        send_confirmation_ok(message.module);
      }else if( effective_state [message.module] == FAIL){
        //No se pueden encender módulos en falla
        send_confirmation_not_ok(message.module);
      }else if( effective_state [message.module] == OFF){
        desired_state [message.module] = ON;

        //Espera a que on_off_manager procese el cambio de estado deseado
        error = wait_for_on_off_manager_action();

        if(error == ERROR_NO_ERROR){
          //Verificar si el modulo fue encendido
          if( effective_state [message.module] == ON){
            //Registrar temporizador para apagar el modulo tras 100 segundos
            timer_t timer = get_sband_timer(message.module);
            if (timer != TIMER_ERROR) {
              set_timer_counter(timer, 100);
              timer_callback_t callback = get_sband_callback(message.module);
              if(callback){
                set_timer_callback (timer, callback);
              }
            }

            //Enviar confirmacion de encendido a MCS
            send_confirmation_ok(message.module);
          }else{
            //Se restaura el valor de estado deseado OFF
            desired_state [message.module] = OFF;

            //Se informa que el modulo no fue encendido
            send_confirmation_not_ok(message.module);
          }
        }else{
          //Se restaura el valor de estado deseado OFF
          set_desired_state (message.module, OFF);

          //Se informa que el modulo no fue encendido
          send_confirmation_not_ok(message.module);
        }
```

```
      }
    }

  } else if (message.success == TRUE && message.command == TURN_OFF) {
    desired_state[message.module] = OFF;
    timer = get_sband_timer(message.module);
    if (timer != TIMER_ERROR) {
      set_timer_callback(timer, 0);
      set_timer_counter(timer, 0);
    }
  }
}
```

1.1.8. Bucle temporizador (timer_loop)

Aplicación utilitaria que permite ejecutar un bloque arbitrario de código dentro de cierto tiempo.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Brindar una interfaz para registrar temporizadores que ejecuten un método al cumplir el periodo de tiempo deseado.

`Funcionamiento`

Para utilizar un temporizador, las aplicaciones deben registrar la función a ejecutar en la variable de estado `timer_callbacks`, e inicializar la entrada correspondiente (mismo índice) de `timer_counters`. La aplicación `timer_loop` recorre el arreglo `timer_counters` cada un segundo, decrementando cualquier entrada que sea positiva. Cuando una entrada se hace 0, se invoca el elemento de `timer_callbacks` en el mismo índice. La ejecución se realiza una vez. Si la aplicación necesita que el callback se ejecute nuevamente, ésta debe registrarlo una vez más.

Pseudocódigo: **timer_loop**

```
timer_loop() {
  for (;;) {
    sleep(1);

    for (t = 0; t < NUMBER_OF_TIMERS; t++) {
      counter = get_timer_counter(t);

      //Decrementar valor del timer cada 1 segundo
      if (counter > 0) {
        counter--;
        set_timer_counter(t, counter);

        //Al alcanzar el valor cero ejecutar el callback
        if (counter == 0) {
          timer_callback_t callback = get_timer_callback(t);
          if (callback) {
            callback();
          }
        }
      }
    }
  }
}
```

1.1.9. Restaurador de módulos (modules_restorer)

Restablece el contador de reintentos de las aplicaciones en falla.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Incrementar en uno el contador de reintentos de una aplicación en falla tras el tiempo indicado.

`Funcionamiento`

Cuando un módulo falla, el gestor de energía debe marcarlo y establecer un timeout en minutos en la entrada correspondiente del vector de timeouts (`modules_timeouts`). La aplicación recorre cada 60 segundos el vector de timeouts decrementando en uno cada entrada positiva. Cuando el valor llega a cero se reasigna el valor uno a la entrada del vector de reintentos del módulo que corresponda.

Pseudocódigo: **modules_restorer**

```
modules_restorer() {
    for (,,) {
        foreach(module) {
            if ( retries [module] == 0 && module_timeouts[module] > 0 ) {
                //Cada un minuto decrementar el contador
                module_timeouts[module]--;

                //Cuando el contador llega a cero se restaura un intento.
                if (module_timeouts[module] == 0 ) {
                    retries [module] = 1;
                }
            }
        }

        sleep(60);
    }
}
```

1.1.10. Recolector de datos de baliza (beacon_data_collector)

Escucha los reportes de los módulos MCS, COMM1, COMM2 y ADCS conteniendo datos para la baliza morse.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Escuchar por mensajes desde los módulos MCS, COMM1, COMM2 y ADCS.
- Almacenar los datos recibidos en la variable de estado `beacon_data`.

`Funcionamiento`

Al llegar un mensaje la aplicación valida el identificador de mensaje y determina a partir de él, el módulo de origen. Los datos recibidos son copiados entonces a las entradas correspondientes de la variable de estado `beacon_data`.

Pseudocódigo: **beacon_data_collector**

```
beacon_data_collector() {
    lock(REC_EMS_APPLICATION_BEACON_DATA_COLLECTOR);
    for (;;) {
        //Recibir datos de baliza de otros módulos
        msg = receive_message();
```

```
        switch(msg.src){
            case COMM1_DATA:
                update_comm1_beacon_data(msg.data.comm1_data);
                break;
            case COMM2_DATA:
                update_comm2_beacon_data(msg.data.comm2_data);
                break;
            case MCS_DATA:
                update_mcs_beacon_data(msg.data.mcs_data);
                break;
            case ADCS_DATA:
                update_adcs_beacon_data(msg.data.adcs_data);
                break;
            case USER_MESSAGE_DATA:
                update_usr_msg_beacon_data(msg.data.user_message_data);
                break;
        }
    }
}
```

1.1.11. Baliza morse (morse_beacon)

Controla la frecuencia de envío y conjunto de datos de la baliza morse.

`Responsabilidades`

Las responsabilidades de la aplicación son:m

- Determinar según el modo de energía la frecuencia de envío de la baliza morse.
- Determinar según el modo de energía el conjunto de datos que debe ser enviado.

`Funcionamiento`

Cada 60 segundos verifica el modo de energía actual y el tiempo pasado desde la última baliza enviada. En caso de estar en modo `RECOVERY` y haber pasado más de 5 minutos desde el último envío, se le solicita a la aplicación `morse_communicator` que envíe una baliza simple. En caso de estar en modo `SAFE`, se le indica a la aplicación `morse_communicator` que debe enviar una baliza completa en cada iteración. Sin embargo, en caso de que se haya recibido un mensaje de usuario, en lugar de una baliza completa se le indica que se debe enviar una baliza extendida.

Pseudocódigo: **morse_beacon**

```
morse_beacon(parameters) {
  count = 0;

  for (;;) {
    sleep(60);
    count += 60;

    if (energy_mode == RECOVERY && count >= 300) {
      count = 0;
      write(REC_MORSE_COMMUNICATOR, SEND_BASIC_DATA);
    } else if (energy_mode == SAFE && count >= 60) {
      count = 0;

      user_message = get_user_message();
      if (user_message.lenght > 0 && get_user_message_counter() > 0) {
        error_t ret_val = write(REC_MORSE_COMMUNICATOR, SEND_EXTENDED_DATA);
      } else {
```

```
        error_t  ret_val  = write(REC_MORSE_COMMUNICATOR, SEND_COMPLETE_DATA);
      }
    }
  }
}
```

1.1.12. Receptor de Rogers (roger_receiver)

Recibe las solicitudes de envío de Rogers a través de la baliza por parte de MCS.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Escuchar por solicitudes de MCS indicando que se recibió un comando desde la Tierra.
- Determinar si un mensaje de Roger debe ser enviado a través de la baliza como respuesta a la solicitud.
- Señalizar a la aplicación `morse_communicator` cuando un Roger deba ser enviado.

`Funcionamiento`

La aplicación aguarda por mensajes de confirmación de recepción de mensajes provenientes de la Tierra por parte de MCS. Ante la recepción de un mensaje, la aplicación notifica a la aplicación `morse_comunicator` que se debe enviar una R (Roger) a través de la baliza. Luego se va a dormir por un periodo de tiempo previo a volver a aguardar por el siguiente mensaje, evitando el envío consecutivo de Rogers.

Pseudocódigo: **roger_receiver**

```
roger_receiver (void* parameters) {
  invalid_messages = 0;
  lock(REC_EMS_APPLICATION_ROGER_RECEIVER, 0);
  set_buffer (REC_EMS_APPLICATION_ROGER_RECEIVER, &buffer);

  for (;;) {
    wait_and_process_message(buffer);
  }

}

wait_and_process_message(buffer) {
  error_t  ret_val  = read(REC_EMS_APPLICATION_ROGER_RECEIVER);
  header = header(buffer);
  if (header->message_id == MESSAGE_MCS_EMS_MESSAGE_ACK) {
    write(REC_MORSE_COMMUNICATOR);

    //No enviar otro roger hasta haber pasado un tiempo prudencial
    sleep(ROGER_SEPARATION_TIME);
  }

  reset_buffer (REC_EMS_APPLICATION_ROGER_RECEIVER);
}
```

1.1.13. Comunicador morse (morse_communicator)

Envía mensajes a la tierra en código morse.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Escuchar por las señalizaciones de envío provenientes de las aplicaciones `roger_receiver` y `morse_beacon`.
- Obtener y armar en caso de ser necesario el mensaje a enviar.
- Enviar el mensaje a través de la baliza utilizando el servicio `beacon_service`.

`Funcionamiento`

Aguarda por mensajes de las aplicaciones `roger_receiver` y `morse_beacon` para iniciar el envío de datos. El mensaje indica que tipo de estructura de datos enviar. La aplicación a partir de esta indicación obtiene la estructura de datos correspondiente e invoca al servicio `beacon_service_send_morse_data` pasando como parámetro dicha estructura. Para enviar un Roger simplemente se invoca el servicio indicando el carácter a enviar (R). La baliza simple y la baliza completa se obtienen de la variable de estado `beacon_data`. En caso de tratarse de una baliza extendida, la aplicación se encarga de concatenar a la baliza completa el último mensaje de usuario recibido.

Pseudocódigo: **morse_communicator**

```
morse_communicator() {

  lock(REC_MORSE_COMMUNICATOR);

  for (;;) {

    wait_for_data(action);
    beacon_service_set_morse_unit(get_morse_unit());

    if(action == SEND_ROGER){
      send_roger();
    }else if(action == SEND_BASIC_DATA){
      send_basic_data();

      //Borrar beacon_data a fin de evitar enviar informacion desactualizada
      beacon_data_init();
    }else if(action == SEND_COMPLETE_DATA){
      send_complete_data();

      //Borrar beacon_data a fin de evitar enviar informacion desactualizada
      beacon_data_init();
    }else if(action == SEND_EXTENDED_DATA){
      send_extended_data();

      //Anular el mensaje de usuario cuando el número de envios fue alcanzado
      decrement_user_message_counter();
      if(get_user_message_counter() <= 0){
        t_morse_user_message empty;
        empty.lenght = 0;
        set_user_message(empty);
      }

      //Borrar beacon_data a fin de evitar enviar informacion desactualizada
      beacon_data_init();

    }

  }

}

send_roger(){
  if(get_beacon_enable() == TRUE){
```

```
    beacon_service_send_morse_data(ROGER_CHAR);
  }
}

send_basic_data(){
  if(get_beacon_enable() == TRUE){
    beacon_service_send_morse_data(get_basic_beacon());
  }
}

send_complete_data(){
  if(get_beacon_enable() == TRUE){
    beacon_service_send_morse_data(get_complete_beacon());
  }
}

send_extended_data(){
  if(get_beacon_enable() == TRUE){
    extended.complete = get_complete_beacon();
    extended.msg[0] = encode_char(0x20); //Delimitar mensaje de usuario con un espacio

    message = get_user_message();
    for(i = 1; i <= USER_MESSAGE_MAX_LENGHT && i <= message.lenght; i = i + 1){
        extended.msg[i] = message.msg[i];
    }

    beacon_service_send_morse_data(extended);

  }
}

wait_for_data(action){
  error_t error;
  error = set_buffer(REC_MORSE_COMMUNICATOR, action);
  //Waits until another application sends a signal
  length = read(REC_MORSE_COMMUNICATOR);
  if(lenght != sizeof(t_beacon_msg)){
    error = ERROR_DATA_LOST;
  }

  return error;
}
```

Servicio de baliza (beacon_service)

El servicio de transmisión del mensaje morse se encuentra implementado por dos módulos, cada uno conteniendo una capa lógica de la solución. La capa superior se haya en el módulo `beacon_service`, la cual pública el método `beacon_service_send_morse_data` utilizado para iniciar el servicio. Este método inicia una máquina de estados interruptiva, donde los estados se encuentran determinados por el manejador de la interrupción de un temporizador, las transiciones son las invocaciones a dicho temporizador, las entradas son los caracteres morse del mensaje y las salidas son mensajes a la capa inferior.

Un mensaje morse es generado mediante el envío de una señal a través de una onda portadora. Los tiempos que la señal se mantiene alta o baja codifican el mensaje. Una unidad de tiempo arbitraria es definida y se siguen un conjunto de reglas:

- Un punto es determinado por una señal alta de un tiempo.

- Una línea es determinada por una señal alta de tres tiempos.
- Entre cada componente (punto o línea) se requiere un silencio de un tiempo.
- Entre cada carácter se requiere un silencio de tres tiempos.
- Entre cada palabra se requiere un silencio de siete tiempos.
- Entre cada mensaje se requiere un silencio de siete tiempos.

La máquina de estados de la capa superior se encarga de descomponer el mensaje en puntos y líneas. A su vez se encarga de generar los silencios entre caracteres, palabras y mensajes. Para ello se utiliza un temporizador del hardware, el cual se configura para interrumpir en un determinado tiempo. Cada interrupción del temporizador invoca un manejador, implementado como un puntero a función que brinda la lógica de cada estado de la máquina. Los silencios son entonces controlados por el temporizador, mientras que la señal es controlada por la capa inferior.

Los caracteres morse son codificados por el módulo `morse_encoder` como una palabra de 16 bits dónde los últimos cuatro bits indican la cantidad de componentes y los primeros doce codifican los mismos, siendo 0 el valor del punto y 1 el valor de la raya.

El método `beacon_service_send_morse_data` tras iniciar el servicio se bloquea mediante una lectura al recurso `REC_BEACON_SERVICE`. Al finalizar, el servicio señaliza al recurso `REC_BEACON_SERVICE` para indicar el fin de la transmisión.

El proceso interruptivo se encuentra determinado por cinco funciones:

`send_morse_message` Un índice que indica el próximo carácter morse a enviar es mantenido globalmente dentro del módulo. A partir de dicho índice la función determina si hay caracteres pendientes de envío. En caso afirmativo se invoca la función `send_morse_character` para enviar el carácter actual. De lo contrario, cuando el índice supera el largo del mensaje, significa que la transmisión ha terminado. En este caso se hace uso del temporizador utilizando como manejador del mismo la función `notify_end_of_transmission` y realizando una espera de 4 tiempos para completar el silencio de fin de mensaje.

`send_morse_character` Otro índice que indica el próximo componente del carácter morse a enviar es mantenido globalmente dentro del módulo. A partir de dicho índice la función determina si hay componentes pendientes de envío. Para ello se utiliza el largo codificado en los últimos 4 bits del carácter morse. Si el índice de componente no supera dicho largo, se determina cuál es el próximo componente a enviar verificando el valor del i-esimo bit de izquierda a derecha, con i siendo el índice de componente. Si vale cero se invoca la función `send_dot` para enviar un punto, y si vale 1 se invoca la función `send_dash` para enviar una línea. Si se determina que todos los componentes fueron enviados, se hace uso del temporizador utilizando como manejador del mismo la función `send_morse_message` y realizando una espera de 3 tiempos para asegurar el silencio de fin de carácter. Un caso especial se da cuando el carácter es un espacio, ya que no tiene componentes. Un silencio de 7 tiempos es generado en este caso.

`send_dot` Esta función invoca a la capa inferior indicando que mantenga la señal en alto durante 1 tiempo y al finalizar invoque a la función `send_morse_character`.

`send_dash` Esta función invoca a la capa inferior indicando que mantenga la señal en alto durante 3 tiempos y al finalizar invoque a la función `send_morse_character`.

`notify_end_of_transmission` Señaliza el recurso `REC_BEACON_SERVICE` con un writei.

La capa inferior por su parte es implementada en el módulo `ems_hal_beacon`, el cual pública el método `hal_beacon_signal_up(duration, callback)` encargado de enviar una señal alta durante el periodo de tiempo indicado como parámetro. Al finalizar la señal, se invoca al manejador indicado como segundo parámetro.

El amplificador y el modulador de la señal deben ser encendidos antes de cada transmisión y apagados al finalizar. Una máquina de estados interruptiva es usada para controlar los tiempos de espera requeridos por los periféricos para su encendido y apagado. Para gestionar estos tiempos se usa nuevamente el temporizador, utilizando como manejador la función `hal_beacon_signal_up_handler`. Es esta función se controlan los cuatro estados:

- **TS_NO_TRANSMISSION** Estado inicial. Enciende el amplificador y el modulador con la señal baja. Luego invoca al temporizador para realizar una espera de 5 milisegundos.
- **TS_TURNING_ON_MODULATOR** Levanta la señal. Luego invoca al temporizador para realizar la espera correspondiente a la duración requerida para la señal.
- **TS_SIGNAL_UP** Apaga el modulador y baja la señal. Luego invoca al temporizador para realizar una espera de 5 milisegundos.
- **TS_TURNING_OFF_MODULATOR** Apaga el amplificador. Luego asigna el manejador indicado como parámetro en el temporizador y realiza una espera de un tiempo, correspondiente al silencio entre componentes.

1.1.14. Reporte de telemetría (telemetry_report)

Recaba y reporta los datos de telemetría al módulo de control principal.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Recolectar periódicamente los datos de telemetría.
- Enviar a MCS la información recolectada.

`Funcionamiento`

Cada 300 segundos la aplicación recolecta los datos requeridos para telemetría utilizando los métodos accesorios del `hal`. Una vez listos, los datos son enviados a MCS.

Pseudocódigo: **telemetry_report**

```
telemetry_report(parameters) {

  for (;;) {
    sleep(300);

    if ( get_effective_state (MODULE_MCS) == ON && !hal_mcs_fault()) {

      refresh_telemetry(content);
      send(MCS, MESSAGE_EMS_MCS_TELEMETRY_REPORT, content);
    }
  }
}

refresh_telemetry(telemetry) {
  telemetry->report.current_cell_x = hal_get_current_cell_x();
  telemetry->report.current_cell_y = hal_get_current_cell_y();
  telemetry->report.current_cell_z = hal_get_current_cell_z();
  telemetry->report.current_sge = hal_get_current_sge();
  telemetry->report.current_beacon = hal_get_current_beacon();
  telemetry->report.current_i2c = hal_get_current_i2c();
  telemetry->report.current_mcs = hal_get_current_mcs();
  telemetry->report.current_comm1 = hal_get_current_comm1();
  telemetry->report.current_comm2 = hal_get_current_comm2();
  telemetry->report.current_adcs = hal_get_current_adcs();
  telemetry->report.current_payload = hal_get_current_payload();
```

```
  telemetry->report.current_sband_1 = hal_get_current_sband_1();
  telemetry->report.current_sband_2 = hal_get_current_sband_2();
  telemetry->report.tension_cell_x =  hal_get_tension_cell_x ();
  telemetry->report.tension_cell_y =  hal_get_tension_cell_y ();
  telemetry->report.tension_cell_z =  hal_get_tension_cell_z ();
  telemetry->report.tension_batteries =  hal_get_tension_batteries ();
  telemetry->report.tension_sge = hal_get_tension_sge();
  telemetry->report.tension_mcs = hal_get_tension_mcs();
  telemetry->report.tension_comm1 = hal_get_tension_comm1();
  telemetry->report.tension_comm2 = hal_get_tension_comm2();
  telemetry->report.tension_adcs = hal_get_tension_adcs();
  telemetry->report.tension_payload = hal_get_tension_payload();
  telemetry->report.tension_sband_1 = hal_get_tension_sband_1();
  telemetry->report.tension_sband_2 = hal_get_tension_sband_2();
  telemetry->report.temperature_sge = hal_get_temperature_sge();
  telemetry->report.output_a_mux1_micro = hal_get_output_a_mux1_micro();
  telemetry->report.output_b_mux1_micro = hal_get_output_b_mux1_micro();
  telemetry->report.output_tension_mppt_x = hal_get_output_tension_mppt_x();
  telemetry->report.output_tension_mppt_y = hal_get_output_tension_mppt_y();
  telemetry->report.output_tension_mppt_z = hal_get_output_tension_mppt_z();
  telemetry->report.antennas_deployed = hal_get_antennas_deployed();
}
```

1.1.15. Reloj (clock)

Recibe las solicitudes provenientes de MCS relacionadas con la hora del sistema.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Actualizar la referencia interna de la hora al recibir actualizaciones de hora desde MCS.
- Enviar el valor de la referencia interna ante una solicitud de MCS.

`Funcionamiento`

La aplicación escucha por mensajes provenientes de MCS.

Si el mensaje es una actualización de hora, se sincroniza el RTC interno con el valor recibido. Si el mensaje es una solicitud de hora, se envía la referencia actual del RTC a MCS.

Pseudocódigo: **clock**

```
clock(parameters) {

  lock(REC_EMS_APPLICATION_CLOCK);

  for (;;) {
    clock_control ();
  }

}

clock_control (){
  msg = receive_clock_message();

  if(msg.success == TRUE){
    if(msg.type == UPDATE_TIME){
      hal_set_time(msg.time);
    }else if(msg.type == REQUEST_TIME){
```

```
      send_time_response();
    }
  }
}

receive_clock_message(){
  message.success = FALSE;

  set_buffer(REC_EMS_APPLICATION_CLOCK, buffer);
  read(REC_EMS_APPLICATION_CLOCK);
  switch(buffer.header.message_id){
    case MESSAGE_MCS_EMS_TIME_UPDATE:
      message.type = UPDATE_TIME;
      message.time = buffer.time_update.time;
      message.success = TRUE;
      break;
    case MESSAGE_MCS_EMS_TIME_REQUEST:
      message.type = REQUEST_TIME;
      message.success = TRUE;
      break;
  }

  return message;
}

send_time_response(){
  time = hal_get_time();

  content.time = time;
  immp_send(MCS, MCS_APPLICATION_1, MESSAGE_EMS_MCS_TIME_RESPONSE, content);
}
```

1.1.16. Monitor I²C (i2c_monitor)

Verifica que el bus I²C funcione correctamente.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Enviar periódicamente un PING al bus I²C.
- Reiniciar el satélite en caso de no obtener respuesta.

`Funcionamiento`

Cada 5 segundos, la aplicación verifica si alguno de los módulos MCS, COMM1 o COMM2 está encendido. Si alguno lo está, le envía un mensaje PING por I²C. Si hay algún error en el envío el mensaje se vuelve a enviar hasta dos veces más. Si se falla las tres veces seguidas, se reinicia todo el satélite (EMS incluído).

Pseudocódigo: **i2c_monitor**

```
i2c_monitor() {
  for (;;) {
    sleep(5);

    //Si el bus i2c esta encendido, verificar que se puedan enviar mensajes a MCS, COMM1 o
        COMM2
    if ( effective_state [MODULE_I2C] == ON && !hal_i2c_fault()) {
      i = 0;
```

```
        modules[0] = MODULE_MCS;
        modules[1] = MODULE_COMM1;
        modules[2] = MODULE_COMM2;

        for (i = 0; i < 3; i++) {
          module = modules[i];
          if ( get_effective_state (module) == ON && !at_fault(module)) {
            check_i2c(module);
            break;
          }
        }
      }
    }
}

at_fault(module) {
  bool ret = FALSE;
  switch (module) {
  case MODULE_MCS:
    ret = hal_mcs_fault();
    break;
  case MODULE_COMM1:
    ret = hal_comm1_fault();
    break;
  case MODULE_COMM2:
    ret = hal_comm2_fault();
    break;
  }

  return ret;
}

check_i2c(module) {

   ret_val = send(module, MCS_APPLICATION_1, MESSAGE_EMS_PING);

   if (ret_val == ERROR_NO_ERROR) {
     return;
   }

   sleep(1);
   ret_val = send(module, MCS_APPLICATION_1, MESSAGE_EMS_PING);
   if (ret_val == ERROR_NO_ERROR) {
     return;
   }

   sleep(1);
   ret_val = send(module, MCS_APPLICATION_1, MESSAGE_EMS_PING);
   if (ret_val == ERROR_NO_ERROR) {
     return;
   }

   /* Si tras 3 repeticiones no se pudo enviar ningun mensaje, entonces el bus no se encuentra
   andando. Se reinicia el saté lite */
   hal_reboot();
}
```

1.2. Apéndice B: Aplicaciones de ADCS

En esta sección se enumeran las distintas aplicaciones a nivel del microkernel que fueron diseñadas, indicando cuál es su responsabilidad y explicando su funcionamiento.

1.2.1. Procesador de Comandos (command_processor)

Esta aplicación se encarga de recibir y procesar los comandos provenientes de otros módulos del satélite, tomando las acciones necesarias como respuesta a los distintos comandos.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Aguardar la recepción de un comando de solicitud de estado desde otro módulo.
- Responder a la solicitud con el estado de actitud según se detalla en la sección de requerimientos.

`Funcionamiento`

La aplicación se bloquea mediante una lectura al recurso reservado por IMMP. Cuando un mensaje es recibido se corrobora que posea el identificador correcto. En caso afirmativo se procede a enviar como respuesta el estado actual de la máquina de estados de actitud, la hora actual mantenida por el módulo y los valores de rotación roll, pitch y yaw obtenidos en la última ejecución del algoritmo de control.

Pseudocódigo: **command_processor**

```
command_processor() {
    lock(RSRC_COMMAND_PROCESSOR);

    for (;;) {

        set_buffer (RSRC_COMMAND_PROCESSOR, command);
        read(RSRC_COMMAND_PROCESSOR, 0);

        if (command.id == MSG_ADCS_STATUS_REQUEST) {
            attitude_status status;
            status.state = get_adcs_state();
            status.time = hal_get_time();
            status.roll = attitude_data.orientation.roll;
            status.yaw = attitude_data.orientation.yaw;
            status.pitch = attitude_data.orientation.pitch;

            send(PAYLOAD, status); //Se envía información de estado al PAYLOAD
        }
    }
}
```

1.2.2. Cargador de Parámetros (params_loader)

Se encarga de solicitar a MCS los parámetros de ACDS y de procesar la recepción de los mismos así como también de las actualizaciones de TLE.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Solicitar a MCS los parámetros del módulo al arranque del mismo.
- Manejar la llegada de una actualización de parámetros, almacenándolos internamente y haciendo visible la posesión de nuevos parámetros.

- Manejar la llegada de una actualización de TLE, almacenándolo internamente y haciendo visible la posesión del nuevo TLE.

`Funcionamiento`

Al inicio de la aplicación, se envía a MCS un mensaje solicitando una actualización de parámetros. Luego, la aplicación se bloquea mediante una lectura al recurso reservado por IMMP. Cuando un mensaje es recibido se corrobora que posea el identificador correcto. Si este corresponde a una actualización de parámetros, se los almacena en la variable de estado attitude_parameters y se enciende la bandera parameters_received. Si corresponde a una actualización de TLE, se los almacena en la variable de estado attitude_tle y se enciende la bandera parameters_tle.

Pseudocódigo: **params_loader**

```
params_loader() {
    lock(RSRC_PARAMS_LOADER);

    intentos = 0;
    do {
        send(MCS, parameters_request); //Solicita parámetros a MCS

        set_buffer (RSRC_PARAMS_LOADER, adcs_params);
        error = read(RSRC_PARAMS_LOADER, TIMEOUT); //Aguarda la respuesta de MCS

        if(error == ERROR_NO_ERROR){
            if(adcs_params.message_id == MSG_ADCS_PARAMS_RESPONSE) {
                set_attitude_parameters(adcs_params.parameters);
                set_parameters_received(TRUE);
            } else if (adcs_params.message_id == MSG_ADCS_TIME_UPDATE) {
                set_attitude_tle (adcs_params.tle);
                set_tle_received (TRUE);
            }
        } else {
            sleep (SLEEP_TIME);
        }

        intentos++;
    } while(error != ERROR_NO_ERROR && intentos < MAX_INTENTOS);


    for (;;) {

        set_buffer (RSRC_PARAMS_LOADER, adcs_params);
        error = read(RSRC_PARAMS_LOADER, TIMEOUT); //Aguarda la respuesta de MCS

        if(error == ERROR_NO_ERROR){
            if(adcs_params.message_id == MSG_ADCS_PARAMS_RESPONSE) {
                set_attitude_parameters(adcs_params.parameters);
                set_parameters_received(TRUE);
            } else if (adcs_params.message_id == MSG_ADCS_TIME_UPDATE) {
                set_attitude_tle (adcs_params.tle);
                set_tle_received (TRUE);
            }
        }
    }

}
```

1.2.3. Reloj (clock)

Se encarga de mantener la hora del sistema interna a ADCS, recibiendo las actualizaciones provenientes de MCS e impactando las mismas en el oscilador interno.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Enviar periódicamente a MCS un mensaje requiriendo una actualización de hora.
- Aguardar la llegada de una actualización de hora desde MCS.
- Actualizar la referencia del oscilador interno con el valor provisto por MCS.

`Funcionamiento`

La aplicación se bloquea mediante una lectura al recurso reservado por IMMP. Cuando un mensaje es recibido se corrobora que posea el identificador correcto. Si se trata de una actualización de hora se corrobora que la hora recibida sea estable, es decir, este sincronizada con la hora de la Tierra. De ser estable, el RTC es actualizado con el valor recibido, en caso contrario se envía el módulo al estado `CRASH`. Si ocurre un timeout durante la espera del mensaje desde MCS, se le envía una solicitud de actualización de hora.

Pseudocódigo: **clock**

```
clock() {
    lock(RSRC_CLOCK);

    for (;;) {
        set_buffer (RSRC_CLOCK, time_update);
        error = read(RSRC_CLOCK, NO_TIME_UPDATE_TIMEOUT);
        if(error == TIMEOUT) {
            send(MCS, time_update_request); //Solicita una actualización de hora
        } else if () {
            if(time_update.time & 0x80000000) {//La hora recibida es insegura
                adcs_state = CRASH;
            } else {
                hal_set_time(time_update.time);
            }
        }
    }
}
```

1.2.4. Reporte de Telemetría (telemetry_report)

Se encarga de recolectar datos de telemetría y enviarlos periódicamente al MCS.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Relevar los datos requeridos por telemetría.
- Enviar los datos relevados a MCS periódicamente.

`Funcionamiento`

Tras realizar una espera de 5 minutos, la aplicación lee las variables de estado y copia en una estructura los datos de telemetría. Los valores leídos de las variables de estado se corresponden con el último juego de datos de entrada y la última salida de los algoritmos de detección y control de actitud. Una vez llena la estructura, la misma es enviada a MCS y se reanuda la espera.

Pseudocódigo: **telemetry_report**

```
telemetry_report() {
    refresh_telemetry(); //Obtiene los datos de telemetría de varias fuentes

    loop_counter = 0;
    for (;;) {

        telemetry_data = collect_telemetry();
        send(MCS, telemetry_data);
        sleep(SLEEP_TIME);
    }
}

refresh_telemetry() {
    telemetry->report.adcs_mode = build_mode(get_adcs_state());
    telemetry->report.error_vector = build_error_vector();
    telemetry->report.gyro_x_measurement = get_raw_angular_velocity_x();
    telemetry->report.gyro_y_measurement = get_raw_angular_velocity_y();
    telemetry->report.gyro_z_measurement = get_raw_angular_velocity_z();
    telemetry->report.gyro_temp_x_measurement = adcs_hal_mainproc_gyro_x_temp();
    telemetry->report.gyro_temp_y_measurement = adcs_hal_mainproc_gyro_x_temp();
    telemetry->report.gyro_temp_z_measurement = adcs_hal_mainproc_gyro_x_temp();
    telemetry->report.ss_plus_x_measurement = (int)get_sun_plus_x();
    telemetry->report.ss_plus_y_measurement = (int)get_sun_plus_y();
    telemetry->report.ss_plus_z_measurement = (int)get_sun_plus_z();
    telemetry->report.ss_minus_x_measurement = (int)get_sun_minus_x();
    telemetry->report.ss_minus_y_measurement = (int)get_sun_minus_y();
    telemetry->report.ss_minus_z_measurement = (int)get_sun_minus_z();
    telemetry->report.mm_x_measurement = (int)get_magnetic_field_x();
    telemetry->report.mm_y_measurement = (int)get_magnetic_field_y();
    telemetry->report.mm_z_measurement = (int)get_magnetic_field_z();
    telemetry->report.temp_adcs_msp = adcs_hal_mainproc_get_temperature() *
        TEMP_ADCS_MSP_CALIBRATION;
    telemetry->report.on_off_flags = get_on_off_flags();
    telemetry->report.mt_x_current_cmd = (int)(get_current_x() * CURRENT_CALIBRATION);
    telemetry->report.mt_y_current_cmd = (int)(get_current_y() * CURRENT_CALIBRATION);
    telemetry->report.mt_z_current_cmd = (int)(get_current_z() * CURRENT_CALIBRATION);
    telemetry->report.angular_rate_est_x = (int)(get_ang_vel_x() *
        ANGULAR_RATE_CALIBRATION);
    telemetry->report.angular_rate_est_y = (int)(get_ang_vel_y() *
        ANGULAR_RATE_CALIBRATION);
    telemetry->report.angular_rate_est_z = (int)(get_ang_vel_z() *
        ANGULAR_RATE_CALIBRATION);
    telemetry->report.roll_angle_est = (int)( get_roll() * RPY_CALIBRATION);
    telemetry->report.pitch_angle_est = (int)(get_pitch() * RPY_CALIBRATION);
    telemetry->report.yaw_angle_est = (int)(get_yaw() * RPY_CALIBRATION);
    telemetry->report.position_x = get_position_x();
    telemetry->report.position_y = get_position_y();
    telemetry->report.position_z = get_position_z();
    telemetry->report.velocity_x = get_velocity_x();
    telemetry->report.velocity_y = get_velocity_y();
    telemetry->report.velocity_z = get_velocity_z();
    telemetry->report.sun_model_x = get_sun_model_x();
    telemetry->report.sun_model_y = get_sun_model_y();
    telemetry->report.sun_model_z = get_sun_model_z();
    telemetry->report.magnetic_field_model_x = get_magnetic_field_model_x();
    telemetry->report.magnetic_field_model_y = get_magnetic_field_model_y();
    telemetry->report.magnetic_field_model_z = get_magnetic_field_model_z();
```

```
    telemetry->report.v_sun_est_x = get_v_sun_est_x();
    telemetry->report.v_sun_est_y = get_v_sun_est_y();
    telemetry->report.v_sun_est_z = get_v_sun_est_z();
    }
```

1.2.5. Reporte de Baliza (beacon_report)

Se encarga de recolectar los datos a ser enviados por la baliza, codificarlos como números del 0 al 9 y enviarlos al EMS, multiplexando en el tiempo los datos a ser enviados.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Recolectar los datos requeridos por EMS
- Enviar a EMS los datos recolectados multiplexados en el tiempo.

`Funcionamiento`

Se mantiene un índice a una tabla que indica cual grupo de datos debe ser enviado. Por ejemplo el índice cero se corresponde con el estado actual de actitud, el uno con el estado de falla de tres sensores de sol, el dos con los restantes tres sensores, etc. En cada iteración se envía a EMS el conjunto de datos indicado por el índice y se incrementa en uno su valor. La aplicación realiza una espera de 5 minutos antes de enviar el siguiente grupo de datos.

Pseudocódigo: **beacon_report**

```
beacon_report() {
    loop_counter = 0;
    for (;;)  {
        if(adcs_state == CRASH) {
            loop_counter = loop_counter >= 8 ? 0 : loop_counter;
        } else {
            loop_counter = loop_counter >= 7 ? 0 : loop_counter;
        }

        switch(loop_counter) {
            case 0: //Reporta el estado de la máquina de estados
                beacon_data.aplicacion_destino =
                        EMS_APPLICATION_BEACON_DATA_COLLECTOR;
                beacon_data.modulo_destino = EMS;
                beacon_data.stat = STATE;
                beacon_data.value = adcs_state;

            break;
            case 1: //Reporta el estado de falla de los fotodiodos x+, x-, y+
                beacon_data.aplicacion_destino =
                        EMS_APPLICATION_BEACON_DATA_COLLECTOR;
                beacon_data.modulo_destino = EMS;
                beacon_data.stat = SS1;
                beacon_data.value = build_value( fault_ss_plus_x , fault_ss_minus_x , fault_ss_plus_y );

            break;
            case 2: //Reporta el estado de falla de los fotodiodos y-, z+, z-
                beacon_data.aplicacion_destino =
                        EMS_APPLICATION_BEACON_DATA_COLLECTOR;
                beacon_data.modulo_destino = EMS;
                beacon_data.stat = SS2;
                beacon_data.value = build_value(fault_ss_minus_y , fault_ss_plus_z , fault_ss_minus_z )
                        ;
```

```
            break;
            case 3: //Reporta el estado de falla de los magnetorquers
                beacon_data.aplicacion_destino =
                        EMS_APPLICATION_BEACON_DATA_COLLECTOR;
                beacon_data.modulo_destino = EMS;
                beacon_data.stat = MT;
                beacon_data.value = build_value(fault_mt_x, fault_mt_y, fault_mt_z);

            break;
            case 4: //Reporta el estado de la máquina de estados nuevamente
                beacon_data.aplicacion_destino =
                        EMS_APPLICATION_BEACON_DATA_COLLECTOR;
                beacon_data.modulo_destino = EMS;
                beacon_data.stat = STATE;
                beacon_data.value = adcs_state;

            break;
            case 5: //Reporta el estado de falla de los giroscopios
                beacon_data.aplicacion_destino =
                        EMS_APPLICATION_BEACON_DATA_COLLECTOR;
                beacon_data.modulo_destino = EMS;
                beacon_data.stat = GY;
                beacon_data.value = build_value(fault_gy_x, fault_gy_y, fault_gy_z);

            break;
            case 6: //Reporta el estado de falla de los magnetometros
                beacon_data.aplicacion_destino =
                        EMS_APPLICATION_BEACON_DATA_COLLECTOR;
                beacon_data.modulo_destino = EMS;
                beacon_data.stat = MM;
                beacon_data.value = build_value(0, 0, fault_mm);

            break;
            case 7: //Reporta la causa de entrada al estado crash
                beacon_data.aplicacion_destino =
                        EMS_APPLICATION_BEACON_DATA_COLLECTOR;
                beacon_data.modulo_destino = EMS;
                beacon_data.stat = CRASH_REASON;
                beacon_data.value = encode_crash_reason();

            break;
        }

        send(EMS, beacon_data); //Envía los datos a EMS
        loop_counter++;
        sleep(SLEEP_TIME);
    }
}

build_value(val_2, val_1, val_0){
    value = 0;
    if(val_0){
        value += 1;
    }

    if(val_1){
        value += 2;
```

```
    }

    if(val_2){
        value += 4;
    }

    return value;
}
```

1.2.6. Bucle de Temporización (timer_loop)

Se encarga de actualizar una serie de contadores que ofician de timers proveyendo un servicio al resto de las aplicaciones para temporizar la ejecución de un método. La precisión es a nivel de segundos y esta atada a la precisión del cambio de contexto del kernel.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Decrementar una serie de contadores reservados.
- Ejecutar el método asociado a un contador que alcanza el valor cero.

`Funcionamiento`

Cada un segundo, se recorre el arreglo contenido en la variable de estado `timer_counters` y se decrementa en uno todos los contadores con un valor mayor a cero. Si un contador pasa de 1 a 0, se ejecuta la función almacenada en el mismo índice del arreglo `timer_callbacks`.

Pseudocódigo: **timer_loop**

```
timer_loop() {
  for (;;) {
    sleep(SLEEP_TIME);

    for (t = 0; t < NUMBER_OF_TIMERS; t++) { //Para cada timer
      counter = get_timer_counter(t);
      if (counter > 0) {
        counter--;
        set_timer_counter(t, counter); //Actualiza el contador del timer

        if (counter == 0) { //Si el contador llega a cero ejecuta el callback
          timer_callback_t callback = get_timer_callback(t);
          if (callback) {
            callback();
          }
        }
      }
    }
  }
}
```

1.2.7. Bucle de Control (control_loop)

El bucle de control es la aplicación encargada de llevar a cabo el ciclo de control. Mantiene internamente la máquina de estados que indica la actitud del satélite y es responsable de solicitar el cómputo de los algoritmos al coprocesador.

`Responsabilidades`

Las responsabilidades de la aplicación son:

- Llevar control del ciclo de control, siendo el encargado de gestionar las etapas de sensado, cómputo y actuación.
- Mantener la máquina de estados que indica la actitud del satélite.
- Gestionar la comunicación con el coprocesador.

`Funcionamiento`

El funcionamiento del bucle de control puede ser entendido como una secuencia de etapas en un bucle infinito en las que se gestionan los diferentes servicios de los que es responsable. A más bajo nivel, se trata de dos bucles anidados. El bucle exterior se encarga de la asegurar el estado del coprocesador, verificando el estado de la comunicación, enviando los parámetros iniciales de funcionamiento y reiniciándolo en caso de falla. El bucle interno por su parte mantiene las etapas del ciclo de vida y la gestión de la máquina de estados. A continuación se listan las distintas etapas:

- **Recepción de parámetros y TLE:** Debido a que no es posible determinar con antelación cual va a a ser la orbita exacta del satélite una vez lanzado, no es viable mantener un conjunto de parámetros e información de la orbita (TLE) por defecto que permitan realizar cálculos precisos sin comunicación previa con la Tierra. Por lo tanto, durante esta etapa inicial, el bucle de control realiza una espera activa aguardando por la llegada de dichos datos. La comunicación con el módulo de control principal (encargado de realizar las comunicaciones con la Tierra) no es responsabilidad de esta aplicación. A fin de determinar cuando el ADCS ha recibido un mensaje conteniendo los nuevos parámetros o conteniendo un TLE actualizado, se consultan las variables de estado `parameters_received` para parámetros y `tle_received` para el TLE. Las mismas toman el valor `true` con la llegada de su respectivo mensaje y es responsabilidad del bucle de control volverlas al valor `false` una vez leídas, para poder detectar la llegada de futuros mensajes. La espera activa se realiza hasta que ambas variables valgan `true`, sin un tiempo límite para la misma. Esta etapa es de inicialización y solo se ejecuta una vez.
- **Verificación de comunicación con coprocesador:** Al ser encendido el módulo ADCS, el coprocesador no es capaz de responder inmediatamente a la recepción de mensajes provenientes del procesador principal, debiendo realizar una serie de inicializaciones previas. A fin de coordinar ambos procesadores para que se puedan intercambiar mensajes, el bucle principal previo al envío de datos y solicitudes al coprocesador, verifica que este sea capaz de responder a los mensajes enviados. Para ello se utiliza el mensaje PING del protocolo SPI_RPC. Si se obtiene un mensaje PONG como respuesta la comunicación esta establecida y se continua con la ejecución, de lo contrario se entra en una espera activa reiterando el envío de PINGs hasta que un PONG sea obtenido o se agote el tiempo de espera (Timeout). En caso de timeout se procede a reiniciar el coprocesador. Esta etapa pertenece al bucle exterior.
- **Envío de parámetros y TLE al coprocesador** Una vez obtenidos los parámetros y el TLE, y verificado que existe comunicación, se procede a enviar estos datos al coprocesador. El envío de parámetros se realiza mediante el mensaje SET_PARAMS, el cual espera como respuesta el mensaje PARAMS_RECEIVED conteniendo un código de confirmación. En caso de falla en el envío/recepción de alguno de estos mensajes o se obtenga un código de confirmación invalido, no se retorna error inmediatamente, sino que se cuenta con un sistema de reintentos con espera incremental entre ellos. Si luego de alcanzado el número máximo de reintentos no se obtiene una respuesta positiva, se considera que el enlace con el coprocesador se ha visto dañado, por lo que se procede a reiniciar el coprocesador. El envío de TLE se realiza en forma análoga mediante los mensajes SET_TLE y TLE_RESPONSE. Esta etapa pertenece al bucle exterior.
- **Verificación de actualización de parámetros o TLE** Luego de inicializado el coprocesador se cambia la máquina de estados a estado DETUMBLING y se procede con el bucle interior. El primer paso es verificar si desde el último envío de parámetros o TLE al coprocesador se recibieron actualizaciones de los mismos desde la Tierra. En caso afirmativo es necesario transmitir los datos nuevamente. La verificación vuelve a hacer uso de las variables de ambiente `parameters_received` y `tle_received` análogamente a la etapa de "Recepción de parámetros y TLE", con la diferencia que no se efectúa una espera activa, sino que tan solo se verifica su valor una vez. Por su parte, el envío se realiza en la misma forma que se explicó en la etapa "Envío de parámetros y TLE al coprocesador". Aunque a primera vista puede parecer redundante, esta etapa es necesaria para

mantener actualizado el coprocesador con los últimos datos provenientes de la Tierra mientras se este ejecutando el bucle interior.

- **Obtención de datos de sensado** En esta etapa se aguarda por una señalización sobre el recurso RSRC_SENSOR_DATA_READY indicando que un nuevo conjunto de lecturas realizadas sobre los sensores está disponible. Al recibir la señalización, se asume que los datos se encuentran impactados en la variable de ambiente `attitude_input`.
- **Cómputo de algoritmos de determinación y control** Los datos de sensado son utilizados como entrada por los algoritmos de determinación y control que han de ser ejecutados en el coprocesador. El mensaje CONTROL es transmitido conteniendo los datos leídos en la etapa anterior. El coprocesador al recibir este mensaje procede a computar los algoritmos y una vez finalizado envía el mensaje CONTROL_RESPONSE con los datos de salida, entre los que se encuentra las corrientes que deben ser impactadas en los actuadores. El bucle principal mientras tanto aguarda la llegada del mensaje CONTROL_RESPONSE. Una vez obtenida la respuesta, los datos de salida son almacenados en la variable de ambiente `attitude_output`. Al igual que con el envío de parámetros, en caso de falla en la comunicación, se reintenta el envío del mensaje CONTROL con tiempos de espera incrementales entre cada mensaje. Una vez alcanzado el número máximo de reintentos se sale del bucle interior y se reinicia el coprocesador.
- **Actualización de datos de actuación** Se señaliza el recurso RSRC_CONTROL_READY indicando que nuevos valores para los actuadores están disponibles.
- **Actualización de estado** En base a la salida obtenida de los algoritmos se actualiza la máquina de estados. Al ser la última etapa del bucle interior, la siguiente es "Verificación de actualización de parámetros o TLE".
- **Reinicio de coprocesador** Se envía una señal de reinicio al coprocesador. Siendo la última etapa del bucle exterior, luego de esta etapa se vuelve a la "Verificación de comunicación con coprocesador".

El diagrama de flujo de la Figura 1.1 muestra las distintas etapas dentro de ambos bucles.

Pseudocódigo: **control_loop**

```
control_loop () {
    lock(RSRC_SENSOR_DATA_READY);
     set_buffer (RSRC_SENSOR_DATA_READY,0);

    while(!(get_parameters_received() && get_tle_received())){
        sleep(INIT_DATA_WAIT_TIME);
    }

    set_parameters_received(FALSE);
     set_tle_received (FALSE);

    for (;;) {
        coproc_loop();
        reset ();
    }

    coproc_loop() {
        ok = check_communication();
        if (!ok) return;
        ok = send_parameters_to_coprocessor();
        if (!ok) return;
        ok = send_tle_to_coprocessor ();
        if (!ok) return;

        //Al ingresar a detumble se inicia el timer que controla el timeout del estado
        adcs_state = DETUMBLE;
        set_timer(detumbling_timeout, 6000);
```

```
        send_status_change_event(adcs_state);

        no_error = TRUE;
        while(no_error) {
            no_error = control_loop_control();
        }
    }
}

control_loop_control() {

    if(adcs_state == DETUMBLE || adcs_state == NOMINAL || adcs_state == NOMINAL_OK) {
        //Si se recibieron nuevos parametros o tle, enviarlos al coprocesador.
        if(get_parameters_received()){
            set_parameters_received(FALSE);
            success = send_parameters_to_coprocessor();
            if(!success) return FALSE;
        }

        if( get_tle_received ()) {
            set_tle_received (FALSE);
            success = send_tle_to_coprocessor();
            if(!success) return FALSE;
        }

        write(RSRC_SENSOR_DATA_SYNC);//Indica que el proceso se encuentra listo para recibir
            datos
        read(RSRC_SENSOR_DATA_READY);//Lee los datos recolectados de los sensores por otra
            aplicación

        control_done = control();//La corriente se escribe en variable global

        if(control_done) {
            write(RSRC_CONTROL_DONE);
        }
    } else {
        sleep(SLEEP_TIME);
    }

    //Se actualiza la máquina de estados
    switch(adcs_state) {
        case DETUMBLE:
            if (control_algorithm != 0) {
                    adcs_state = NOMINAL;
                    set_timer(nominal_timeout, 6000);
                    send_status_change_event(adcs_state);
            }
            break;
        case NOMINAL:
            if (control_algorithm == 0) {
                adcs_state = DETUMBLING;
                set_timer(detumbling_timeout, 6000);
                send_status_change_event(adcs_state);
            }else{
                if(nominal_ok_flag == true) {
                    adcs_state = NOMINAL_OK;
                    set_timer(0, 0);
                    send_status_change_event(adcs_state);
```

```
                }
            }
            break;
        case NOMINAL_OK:
            if (control_algorithm == 0) {
                adcs_state = DETUMBLING;
                set_timer(detumbling_timeout, 6000);
                send_status_change_event(adcs_state);
            }else{
                if(nominal_ok_flag == false) {
                    adcs_state = NOMINAL;
                    set_timer(nominal_timeout, 6000);
                    send_status_change_event(adcs_state);
                }
            }
            break;
        }

        case TIMEOUT_DETUMBLE:
        case TIMEOUT_NOMINAL:
        case CRASH:
    }

}
```

1.2.8. Módulos Adicionales

Uno de los criterios utilizados durante el diseño de la solución fue el evitar la dependencia entre los módulos que definen a las aplicaciones. Con éste propósito se definieron una serie de módulos adicionales que exponen operaciones que requirieran ser utilizadas desde varias aplicaciones o que bien por su complejidad e importancia, hayan requerido ser probadas en forma aislada a la aplicación que las utilizaría. A continuación se brinda más detalle de estos módulos.

Control de Actitud (attitude_control)

Este módulo encapsula la ejecución de los algoritmos de determinación y control de actitud, publicando una única operación encargada de enviar los datos de entrada al coprocesador y aguardar por el mensaje de respuesta conteniendo el resultado de la ejecución de los algoritmos.

Operaciones:

- `control`: Esta operación envía al coprocesador un mensaje SPI_RPC (spi_rpc_master_control) conteniendo los datos de entrada de los algoritmos. Estos datos son obtenidos de la variable de estado attitude_input la cual como precondición debe haber sido actualizada con las últimas lecturas de los sensores previo a la ejecución del método. La operación se bloquea aguardando por la recepción del mensaje de respuesta (spi_rpc_master_control_response), el cual contiene el resultado de la ejecución de los algoritmos. Dicho resultado es escrito en la variable de estado attitude_output antes de retornar la operación. En caso de falla en la comunicación, se aplican una serie de reintentos con una espera progresiva entre ellos durante la cual se libera el procesador.

Controlador de estados de ADCS (adcs_state_controller)

Se encarga de proveer las operaciones necesarias para cambiar el estado actual de la máquina de estados de actitud, manteniendo la coherencia entre el estado actual y los temporizadores que controlan los eventos de Timeout. El principal método expuesto contiene la lógica de decisión que determina el próximo estado a partir de la última salida del algoritmo de control.

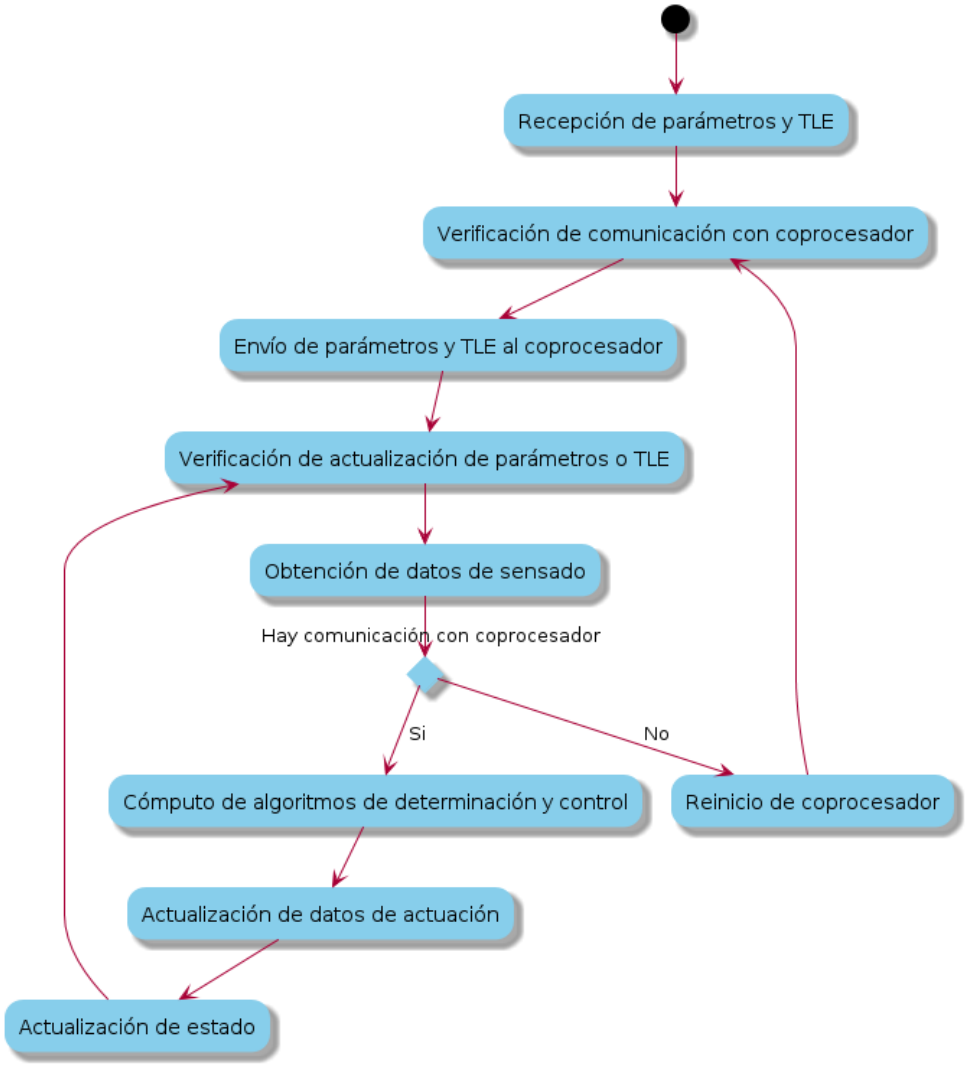

Figura 1.1. Etapas del bucle de control

Operaciones:

- `change_status_init` Inicializa la máquina de estados y los temporizadores que controlan los Timeouts.
- `change_status_timeout_detumble` Cambia al estado `TIMEOUT_DETUMBLE` y apaga los temporizadores.
- `change_status_timeout_nominal` Cambia al estado `TIMEOUT_NOMINAL` y apaga los temporizadores.
- `change_status_nominal` Cambia al estado `NOMINAL` y enciende el temporizador correspondiente.
- `change_status_nominal_ok` Cambia al estado `NOMINAL_OK` y apaga los temporizadores.
- `change_status_detumble` Cambia al estado `DETUMBLE` y enciende el temporizador correspondiente.
- `change_status_crash` Cambia al estado CRASHK indicando la razón de la transición y apaga los temporizadores.
- `state_transition` Define y efectúa la transición al próximo estado utilizando como entrada de la máquina de estados el número de algoritmo de control utilizado y la indicación de alcance de la actitud "nominal ok", ambos provistos en la variable de estados attitude_output. El coprocesador indica tras cada ejecución el algoritmo de control utilizado, siendo este distinto si se encuentra en estado nominal o detumble. A su vez, en caso de alcanzarse una actitud "nominal ok" el coprocesador también informa del hecho. El método infiere el siguiente estado a partir de estos valores y

ejecuta una de las operaciones de cambio de estado enumeradas. La lógica que define a partir de la velocidad y la orientación el estado de actitud se encuentra en el coprocesador.

Comunicador de Parámetros (parameters_communicator)

Se encarga de realizar la comunicación con el coprocesador a fin de transmitir el conjunto de parámetros requeridos para los algoritmos de determinación y control, así como también transmitir la información de TLE y verificar la correcta comunicación con el coprocesador. Todas las operaciones de comunicación mantienen el mismo sistema de reintentos que la operación `control` del módulo `attitude_control`.

Operaciones:

- `check_communication` Envía al coprocesador un mensaje SPI_RPC (spi_rpc_master_send_ping_msg) y aguarda por el mensaje de respuesta (spi_rpc_master_check_pong_msg). Este mensaje tiene el único propósito de confirmar que el canal de comunicación se encuentra en buen estado y el coprocesador se encuentra listo para responder a los subsiguientes mensajes. La operación retorna si se tuvo éxito en recibir la respuesta.
- `send_parameters_to_coprocessor` Comunica al coprocesador el conjunto de parámetros para los algoritmos de determinación y control. Para ello se utiliza un mensaje SPI_RPC (spi_rpc_master_set_params) conteniendo una estructura con todos los parámetros. Una confirmación es recibida desde el coprocesador mediante otro mensaje (spi_rpc_master_params_received).
- `send_tle_to_coprocessor` Comunica al coprocesador la última actualización de TLE. Para ello se utiliza un mensaje SPI_RPC (spi_rpc_master_set_tle) conteniendo una estructura con la información orbital. Una confirmación es recibida desde el coprocesador mediante otro mensaje (spi_rpc_master_tle_received).

1.3. Apéndice C: spimp - Máquinas de estado

A continuación se detalla la implementación de las máquinas de estados correspondientes a las operaciones del protocolo **spimp**. Los byte leídos del bus son usados como entradas de la máquina de estados y los bytes escritos son la salida. Para cada estado se detalla:

1. Nombre del estado actual
2. Byte recibido y lista de condiciones a cumplir.
3. Byte a ser escrito, modificación del valor de variables internas y valor de la salida de error de la operación en caso de ser determinada.
4. Próximo estado.
5. Comentarios pertinentes a la transición.

A su vez la siguiente convenciones son utilizadas:

- — indica que se recibe o envía un byte con un valor arbitrario distinto a cualquier valor ya mencionado para el estado actual.
- *length* es el largo del mensaje a transmitir.
- *error* es la salida de error de la operación.
- *index* es el valor del índice en el buffer de envío o recepción en el paso actual.
- *buffer* es el buffer de recepción del mensaje.
- sizeof(*buffer*) es el tamaño del buffer de recepción del mensaje.
- *byte* es un valor leído del bus.
- *data* es un valor leído del bus correspondiente a un byte del mensaje.

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp STATE RECEIVE START	—	MASTER RECEIVE START	spimp STATE RECEIVE START	El maestro comienza la transferencia
spimp STATE RECEIVE START	—	GET LENGTH	spimp STATE RECEIVE GET LENGTH	Se solicita el largo
spimp STATE RECEIVE GET LENGTH	MASTER RECEIVE ACK	GARBAGE	spimp STATE RECEIVE WAIT LENGTH	Se recibe confirmación del esclavo aceptando la operación. Se procede a enviar un byte arbitrario para poder obtener el largo del mensaje a recibir.
spimp STATE RECEIVE GET LENGTH	—	*error*=SPI ERROR SLAVE NOT READY	spimp STATE RECEIVE DONE	No se recibe confirmación del esclavo. Se asume que el esclavo no se encuentra listo para realizar una transacción y se finaliza indicando error.
spimp STATE RECEIVE WAIT LENGTH	*byte*=*length* — *length*==0	*byte*	spimp STATE RECEIVE END	El largo del mensaje a recibir es cero. Se procede a esperar el byte de fin de mensaje pues no se recibiram datos.
spimp STATE RECEIVE WAIT LENGTH	*byte*=*length* — *length* >sizeof(*buffer*)	sizeof(*buffer*) — *error*=SPI ERROR OVERFLOW	spimp STATE RECEIVE WAIT ERROR ACK	El largo del mensaje a recibir es mayor al largo del buffer de entrada. Se retorna el largo del buffer de entrada y se indica el error. Se procede a aguardar por la confirmación de la recepción del error.
spimp STATE RECEIVE WAIT LENGTH	*byte*=*length* — *length* <sizeof(*buffer*)	*byte*	spimp STATE RECEIVE DATA	El largo del mensaje a recibir es menor al largo del buffer de entrada. Se retorna el largo como confirmación. Se procede a aguardar por la llegada de los datos.
spimp STATE RECEIVE DATA	*byte*=*data* — *index* <sizeof(*buffer*) — *index* + 1 <*length*	*buffer[index]*=*byte* — *index*++ — *byte*	spimp STATE RECEIVE DATA	El byte recibido es colocado en el buffer y enviado como confirmación. Se incrementa el índice que indica en que posición del buffer colocar el próximo byte.
spimp STATE RECEIVE DATA	*byte*=*data* — *index* <sizeof(*buffer*) — *index* + 1 == *length*	*buffer[index]*=*byte* — *index*++ — *byte*	spimp STATE RECEIVE END	Se completó la transmisión de datos. Se procede a esperar el byte de fin de mensaje.

Continúa en la siguiente página

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp STATE RECEIVE DATA	*byte*=*data* — *index* <sizeof(*buffer*) — *index* + 1 >*length*	*error*=SPI ERROR FATAL	spimp STATE RECEIVE DONE	Situación inesperada en que a pesar del control, el índice desborda el tamaño de mensaje indicado. Se aborta la transmisión.
spimp STATE RECEIVE DATA	*byte*=*data* — *index* >= sizeof(*buffer*)	*error*=SPI ERROR FATAL	spimp STATE RECEIVE DONE	Situación inesperada en que el índice desborda el tamaño del buffer. Se aborta la transmisión.
spimp STATE RECEIVE END	MASTER RECEIVE END	MASTER RECEIVE END ACK	spimp STATE RECEIVE WAIT CONFIRMA-TION	Se recibe el byte de finalización. Se envía confirmación del mismo y se procede a aguardar por el resultado de la transmisión.
spimp STATE RECEIVE END	—	MASTER RECEIVE ERROR — *error*=SPI ERROR INVALID DATA	spimp STATE RECEIVE ERROR	Falla al recibir el byte de finalización. Se envía informe del error y se pasa al estado de error.
spimp STATE RECEIVE ERROR	—	MASTER RECEIVE ERROR	spimp STATE RECEIVE WAIT ERROR ACK	Se envía informe de error. Se procede a aguardar la confirmación del mismo.
spimp STATE RECEIVE WAIT CONFIRMA-TION	MASTER RECEIVE CON-FIRMATION OK	MASTER RECEIVE DONE	spimp STATE RECEIVE WAIT IS DONE	La transmisión fue exitosa. Se envía byte de cierre de transmisión y se procede a esperar el cierre del emisor.
spimp STATE RECEIVE WAIT CONFIRMA-TION	—	MASTER RECEIVE — *error*=ERROR SPI ERROR INVALID DATA	spimp STATE RECEIVE WAIT ERROR ACK	La transmisión falló. Se envía byte de error y se procede a esperar su confirmación.
spimp STATE RECEIVE WAIT ERROR ACK	—	—	spimp STATE RECEIVE DONE	Se finaliza la transmisión tras el error.
spimp STATE RECEIVE WAIT IS DONE	MASTER RECEIVE IS DONE	*error*=SPI ERROR NO ERROR	spimp STATE RECEIVE DONE	El emisor cierra la transmisión indicando éxito. Se finaliza la transmisión.
spimp STATE RECEIVE WAIT IS DONE	—	*error*=SPI ERROR INVALID END	spimp STATE RECEIVE DONE	El emisor cierra la transmisión indicando falla. Se finaliza la transmisión.

Tabla 1.2. Máquina de estados del Master Receive

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp STATE SEND START	—	MASTER SEND START	spimp STATE SEND START	El maestro comienza la transferencia.

Continúa en la siguiente página

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp STATE SEND START	—	SET LENGTH	spimp STATE SEND SET LENGTH	Se envía el byte SET LENGTH para poder recibir la respuesta del esclavo.
spimp STATE SEND SET LENGTH	OK	*length*	spimp STATE SEND WAIT LENGTH ACK	Se recibe confirmación del esclavo aceptando la operación. Se procede a enviar el largo del mensaje a transmitir.
spimp STATE SEND SET LENGTH	—	*error*=SPI ERROR SLAVE NOT READY	spimp STATE SEND DONE	No se recibe confirmación del esclavo. Se asume que el esclavo no se encuentra listo para realizar una transacción y se finaliza indicando error.
spimp STATE SEND WAIT LENGTH ACK	*length*==0	MASTER SEND END	spimp STATE SEND END	El largo del mensaje a enviar es cero. Se procede a enviar el byte de fin de mensaje pues no se enviarán datos.
spimp STATE SEND WAIT LENGTH ACK	*length*>0	*buffer[index]*	spimp STATE SEND DATA	El largo del mensaje a enviar es mayor a cero. Se envía el primer byte.
spimp STATE SEND DATA	*index*==0 — *byte* <length	*error*=SPI ERROR OVERFLOW	spimp STATE SEND DONE	La confirmación del largo indica que éste es superior al tamaño del buffer de recepción. Se indica el error y se finaliza la transmisión.
spimp STATE SEND DATA	*index*==0 — *byte* >length	*error*=SPI ERROR INVALID DATA	spimp STATE SEND DONE	La confirmación del largo no es correcta. Se indica el error y se finaliza la transmisión.
spimp STATE SEND DATA	*index*==0 — *byte*==length — *index* + 1 <*length*	*buffer[index+1]* — *index*++	spimp STATE SEND DATA	La confirmación del largo es correcta. Se envía el siguiente byte.
spimp STATE SEND DATA	*index*==0 — *byte*==length — *index* + 1 == *length*	*MASTER SEND END* — *index*++	spimp STATE SEND END	La confirmación del largo es correcta y ya no hay mas bytes para envíar. Se envía el byte de fin de mensaje.
spimp STATE SEND DATA	*index* <length — *byte* != *buffer[index-1]* — *index* + 1 <*length*	*buffer[index+1]* — *index*++ — *valid*=false	spimp STATE SEND DATA	La confirmación del byte anterior es incorrecta y aún hay mas bytes para envíar. Se envía el siguiente byte y se marca el mensaje como invalido.

Continúa en la siguiente página

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp STATE SEND DATA	*index* <length — *byte* != *buffer[index-1]* — *index* + 1 == *length*	MASTER SEND END — *index*++ — *valid*=false	spimp STATE SEND END	La confirmación del byte anterior es incorrecta y ya no hay mas bytes para envíar. Se envía el byte de finalización de mensaje y se marca el mensaje como invalido.
spimp STATE SEND DATA	*index* <length — *byte* == *buffer[index-1]* — *index* + 1 <*length*	*buffer[index+1]* — *index*++	spimp STATE SEND DATA	La confirmación del byte anterior es correcta y aún hay mas bytes para envíar. Se envía el siguiente byte.
spimp STATE SEND DATA	*index* <length — *byte* == *buffer[index-1]* — *index* + 1 == *length*	MASTER SEND END — *index*++	spimp STATE SEND END	La confirmación del byte anterior es correcta y ya no hay mas bytes para envíar. Se envía el byte de finalización de mensaje.
spimp STATE SEND DATA	*index* >length	MASTER SEND END — *index*++ — *valid*=false	spimp STATE SEND END	El indice se sale de sus márgenes. Se envía el byte de finalización de mensaje y se marca el mensaje como invalido.
spimp STATE SEND END	*index*==0 — *byte*!=length	MASTER SEND CONFIRMA-TION INVALID — *error*=SPI ERROR INVALID DATA — *valid*=false	spimp STATE SEND ERROR	La confirmación del largo es erronea. Se envía el error.
spimp STATE SEND END	*index*==0 — *byte*==length	MASTER SEND CONFIRMA-TION OK	spimp STATE SEND WAIT END ACK	La confirmación del largo es correcta. Se envía el informe de éxito.
spimp STATE SEND END	*index*==length — *byte* != *buffer[index-1]*	MASTER SEND CONFIRMA-TION INVALID — *error*=SPI ERROR INVALID DATA — *valid*=false	spimp STATE SEND ERROR	La confirmación del byte anterior es erronea. Se envía el error.

Continúa en la siguiente página

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp STATE SEND END	*index*==length — *byte* == *buffer[index-1]* — *valid*==false	MASTER SEND CONFIRMATION INVALID — *error*=SPI ERROR INVALID DATA — *valid*=false	spimp STATE SEND ERROR	La confirmación de algún byte anterior fue erronea. Se envía el error.
spimp STATE SEND END	*index*==length — *byte* == *buffer[index-1]* — *valid*==true	MASTER SEND CONFIRMATION OK	spimp STATE SEND WAIT END ACK	La confirmación del byte anterior es correcta. Se envía el informe de éxito.
spimp STATE SEND END	*index*!=0 — *index*!=length	MASTER SEND CONFIRMATION INVALID — *error*=SPI ERROR INVALID DATA — *valid*=false	spimp STATE SEND ERROR	El indice tiene un valor no esperado. Se envía el error.
spimp STATE SEND ERROR	—	MASTER SEND ERROR	spimp STATE SEND WAIT ERROR ACK	Se envía informe de error. Se procede a aguardar la confirmación del mismo.
spimp STATE SEND WAIT END ACK	MASTER SEND END ACK	MASTER SEND IS DONE	spimp STATE SEND WAIT DONE	La transmisión fue exitosa. Se envía byte de cierre de transmisión y se procede a esperar el cierre del emisor.
spimp STATE SEND WAIT END ACK	—	MASTER SEND ERROR — *error*=SPI ERROR INVALID DATA	spimp STATE SEND WAIT ERROR ACK	La transmisión falló. Se envía byte de error y se procede a esperar su confirmación.
spimp STATE SEND WAIT ERROR ACK	—	—	spimp STATE SEND DONE	Se finaliza la transmisión tras el error.
spimp STATE SEND WAIT DONE	MASTER SEND DONE	*error*=SPI ERROR NO ERROR	spimp STATE SEND DONE	El emisor cierra la transmisión indicando éxito. Se finaliza la transmisión.
spimp STATE SEND WAIT DONE	—	*error*=SPI ERROR INVALID END	spimp STATE SEND DONE	El emisor cierra la transmisión indicando falla. Se finaliza la transmisión.

Tabla 1.3. Máquina de estados del Master Send

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp SLAVE STATE SLAVE RECEIVE START	MASTER SEND START	OK	spimp SLAVE STATE SET LENGTH WAIT	Se recibe el byte de comienzo de la transferencia del maestro.
spimp SLAVE STATE SLAVE RECEIVE START	—	*error*=SPI ERROR INVALID DATA	spimp SLAVE STATE DONE	El byte recibido es inválido. Se finaliza la operación.
spimp SLAVE STATE SET LENGTH WAIT	SET LENGTH	SET LENGTH ACK	spimp SLAVE STATE LENGTH WAIT	Se recibe confirmación del maestro aceptando la operación. Se procede a informar que se esta esperando el largo del mensaje a recibir.
spimp SLAVE STATE SET LENGTH WAIT	—	*error*=SPI ERROR INVALID DATA	spimp SLAVE STATE DONE	No se recibe confirmación del maestro. Se finaliza indicando error.
spimp SLAVE STATE LENGTH WAIT	*byte=length* — *length*==0	*byte*	spimp SLAVE STATE SLAVE RECEIVE END	El largo del mensaje a recibir es cero. Se procede a esperar el byte de fin de mensaje pues no se recibirán datos.
spimp SLAVE STATE LENGTH WAIT	*byte=length* — *length* >sizeof(*buffer*)	sizeof(*buffer*) — *error*=SPI ERROR OVERFLOW	spimp SLAVE STATE ERROR ACK WAIT	El largo del mensaje a recibir es mayor al largo del buffer de entrada. Se retorna el largo del buffer de entrada y se indica el error. Se procede a aguardar por la confirmación de la recepción del error.
spimp SLAVE STATE LENGTH WAIT	*byte=length* — *length* <sizeof(*buffer*)	*byte*	spimp SLAVE STATE DATA RX	El largo del mensaje a recibir es menor al largo del buffer de entrada. Se retorna el largo como confirmación. Se procede a aguardar por la llegada de los datos.
spimp SLAVE STATE DATA RX	*byte=data* — *index* <sizeof(*buffer*) — *index* + 1 <*length*	*buffer[index]=byte* — *index++* — *byte*	spimp SLAVE STATE DATA RX	El byte recibido es colocado en el buffer y enviado como confirmación. Se incrementa el índice que indica en que posición del buffer colocar el próximo byte.
spimp SLAVE STATE DATA RX	*byte=data* — *index* <sizeof(*buffer*) — *index* + 1 == *length*	*buffer[index]=byte* — *index++* — *byte*	spimp SLAVE STATE SLAVE RECEIVE END	Se completó la transmisión de datos. Se procede a esperar el byte de fin de mensaje.

Continúa en la siguiente página

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp SLAVE STATE DATA RX	*byte*=*data* — *index* <sizeof(*buffer*) — *index* + 1 >*length*	*error*=SPI ERROR FATAL	spimp SLAVE STATE DONE	Situación inesperada en que a pesar del control, el índice desborda el tamaño de mensaje indicado. Se aborta la transmisión.
spimp SLAVE STATE DATA RX	*byte*=*data* — *index* >= sizeof(*buffer*)	*error*=SPI ERROR FATAL	spimp SLAVE STATE DONE	Situación inesperada en que el índice desborda el tamaño del buffer. Se aborta la transmisión.
spimp SLAVE STATE SLAVE RECEIVE END	MASTER SEND END	MASTER SEND END ACK	spimp SLAVE STATE CONFIRMATION WAIT	Se recibe el byte de finalización. Se envía confirmación del mismo y se procede a aguardar por el resultado de la transmisión.
spimp SLAVE STATE SLAVE RECEIVE END	—	MASTER SEND ERROR — *error*=SPI ERROR INVALID DATA	spimp SLAVE STATE ERROR	Falla al recibir el byte de finalización. Se envía informe del error y se pasa al estado de error.
spimp SLAVE STATE ERROR	—	MASTER SEND ERROR	spimp SLAVE STATE ERROR ACK WAIT	Se envía informe de error. Se procede a aguardar la confirmación del mismo.
spimp SLAVE STATE CONFIRMATION WAIT	MASTER SEND CONFIRMATION OK	MASTER SEND DONE	spimp SLAVE STATE IS DONE WAIT	La transmisión fue exitosa. Se envía byte de cierre de transmisión y se procede a esperar el cierre del emisor.
spimp SLAVE STATE CONFIRMATION WAIT:	—	MASTER SEND ERROR — *error*=ERROR SPI ERROR INVALID DATA	spimp SLAVE STATE ERROR ACK WAIT	La transmisión falló. Se envía byte de error y se procede a esperar su confirmación.
spimp SLAVE STATE ERROR ACK WAIT	—	—	spimp SLAVE STATE DONE	Se finaliza la transmisión tras el error.
spimp SLAVE STATE IS DONE WAIT	MASTER SEND IS DONE	*error*=SPI ERROR NO ERROR	spimp SLAVE STATE DONE	El emisor cierra la transmisión indicando éxito. Se finaliza la transmisión.
spimp SLAVE STATE IS DONE WAIT	—	*error*=SPI ERROR INVALID END	spimp SLAVE STATE DONE	El emisor cierra la transmisión indicando falla. Se finaliza la transmisión.

Tabla 1.4. Máquina de estados del Slave Receive

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp SLAVE STATE SLAVE SEND START	MASTER RECEIVE START	MASTER RECEIVE ACK	spimp SLAVE STATE GET LENGTH WAIT	El maestro indica que esta pronto para iniciar la transferencia. Se confirma la recepción de la misma.
spimp SLAVE STATE SLAVE SEND START	—	*error*=SPI ERROR INVALID DATA	spimp SLAVE STATE DONE	El byte recibido del maestro no se corresponde con la operación. Se finaliza con error.
spimp SLAVE STATE GET LENGTH WAIT	GET LENGTH	*length*	spimp SLAVE STATE SEND LENGTH	Se recibe la solicitud del largo por parte del maestro. Se procede a enviar el largo del mensaje a transmitir.
spimp SLAVE STATE GET LENGTH WAIT	—	*error*=SPI ERROR INVALID DATA	spimp SLAVE STATE DONE	No se recibe solicitud del maestro. Se finaliza indicando error.
spimp SLAVE STATE SEND LENGTH	*length*==0	MASTER RECEIVE END	spimp SLAVE STATE SLAVE SEND END	El largo del mensaje a enviar es cero. Se procede a enviar el byte de fin de mensaje pues no se enviarán datos.
spimp SLAVE STATE SEND LENGTH	*length*¿0	*buffer[0]*	spimp SLAVE STATE DATA TX	El largo del mensaje a enviar es mayor a cero. Se envía el primer byte.
spimp SLAVE STATE DATA TX	*index*==0 — *byte* <length	*error*=SPI ERROR OVERFLOW	spimp SLAVE STATE DONE	La confirmación del largo indica que éste es superior al tamaño del buffer de recepción. Se indica el error y se finaliza la transmisión.
spimp SLAVE STATE DATA TX	*index*==0 — *byte* >length	*error*=SPI ERROR INVALID DATA	spimp SLAVE STATE DONE	La confirmación del largo no es correcta. Se indica el error y se finaliza la transmisión.
spimp SLAVE STATE DATA TX	*index*==0 — *byte*==length — *index* + 1 <*length*	*buffer[index+1]* — *index*++	spimp SLAVE STATE DATA TX	La confirmación del largo es correcta. Se envía el siguiente byte.
spimp SLAVE STATE DATA TX	*index*==0 — *byte*==length — *index* + 1 == *length*	*MASTER RECEIVE END* — *index*++	spimp SLAVE STATE SLAVE SEND END	La confirmación del largo es correcta y ya no hay mas bytes para envíar. Se envía el byte de fin de mensaje.
spimp SLAVE STATE DATA TX	*index* <length — *byte* != *buffer[index-1]* — *index* + 1 <*length*	*buffer[index+1]* — *index*++ — *valid*=false	spimp SLAVE STATE DATA TX	La confirmación del byte anterior es incorrecta y aún hay mas bytes para envíar. Se envía el siguiente byte y se marca el mensaje como invalido.

Continúa en la siguiente página

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp SLAVE STATE DATA TX	*index* <length — *byte* != *buffer[index-1]* — *index* + 1 == *length*	MASTER RECEIVE END — *index*++ — *valid*=false	spimp SLAVE STATE SLAVE SEND END	La confirmación del byte anterior es incorrecta y ya no hay mas bytes para envíar. Se envía el byte de finalización de mensaje y se marca el mensaje como invalido.
spimp SLAVE STATE DATA TX	*index* <length — *byte* == *buffer[index-1]* — *index* + 1 <*length*	*buffer[index+1]* — *index*++	spimp SLAVE STATE DATA TX	La confirmación del byte anterior es correcta y aún hay mas bytes para envíar. Se envía el siguiente byte.
spimp SLAVE STATE DATA TX	*index* <length — *byte* == *buffer[index-1]* — *index* + 1 == *length*	MASTER RECEIVE END — *index*++	spimp SLAVE STATE SLAVE SEND END	La confirmación del byte anterior es correcta y ya no hay mas bytes para envíar. Se envía el byte de finalización de mensaje.
spimp SLAVE STATE DATA TX	*index* >length	MASTER RECEIVE END — *index*++ — *valid*=false	spimp SLAVE STATE SLAVE SEND END	El indice se sale de sus márgenes. Se envía el byte de finalización de mensaje y se marca el mensaje como invalido.
spimp SLAVE STATE SLAVE SEND END	*index*==0 — *byte*!=length	MASTER RECEIVE CONFIRMATION INVALID — *error*=SPI ERROR INVALID DATA — *valid*=false	spimp SLAVE STATE SEND ERROR	La confirmación del largo es erronea. Se envía el error.
spimp SLAVE STATE SLAVE SEND END	*index*==0 — *byte*==length	MASTER RECEIVE CONFIRMATION OK	spimp SLAVE STATE END ACK WAIT	La confirmación del largo es correcta. Se envía el informe de éxito.
spimp SLAVE STATE SLAVE SEND END	*index*==length — *byte* != *buffer[index-1]*	MASTER RECEIVE CONFIRMATION INVALID — *error*=SPI ERROR INVALID DATA — *valid*=false	spimp SLAVE STATE SEND ERROR	La confirmación del byte anterior es erronea. Se envía el error.

Continúa en la siguiente página

Estado Actual	**Recibe Byte Cumpliendo Condición**	**Escribe Byte y/o Finaliza con Error**	**Próximo Estado**	**Comentarios**
spimp SLAVE STATE SLAVE SEND END	*index*==length — *byte* == *buffer[index-1]* — *valid*==false	MASTER RECEIVE CONFIRMATION INVALID — *error*=SPI ERROR INVALID DATA — *valid*=false	spimp SLAVE STATE SEND ERROR	La confirmación de algún byte anterior fue erronea. Se envía el error.
spimp SLAVE STATE SLAVE SEND END	*index*==length — *byte* == *buffer[index-1]* — *valid*==true	MASTER RECEIVE CONFIRMATION OK	spimp SLAVE STATE END ACK WAIT	La confirmación del byte anterior es correcta. Se envía el informe de éxito.
spimp SLAVE STATE SLAVE SEND END	*index*!=0 — *index*!=length	MASTER RECEIVE CONFIRMATION INVALID — *error*=SPI ERROR INVALID DATA — *valid*=false	spimp SLAVE STATE SEND ERROR	El indice tiene un valor no esperado. Se envía el error.
spimp SLAVE STATE SEND ERROR	—	MASTER RECEIVE ERROR	spimp SLAVE STATE SEND ERROR ACK WAIT	Se envía informe de error. Se procede a aguardar la confirmación del mismo.
spimp SLAVE STATE END ACK WAIT	MASTER RECEIVE END ACK	MASTER RECEIVE IS DONE	spimp SLAVE STATE DONE WAIT	La transmisión fue exitosa. Se envía byte de cierre de transmisión y se procede a esperar el cierre del emisor.
spimp SLAVE STATE END ACK WAIT	—	MASTER RECEIVE ERROR — *error*=SPI ERROR INVALID DATA	spimp SLAVE STATE SEND ERROR ACK WAIT	La transmisión falló. Se envía byte de error y se procede a esperar su confirmación.
spimp SLAVE STATE SEND ERROR ACK WAIT	—	—	spimp SLAVE STATE DONE	Se finaliza la transmisión tras el error.
spimp SLAVE STATE DONE WAIT	MASTER RECEIVE DONE	*error*=SPI ERROR NO ERROR	spimp SLAVE STATE DONE	El emisor cierra la transmisión indicando éxito. Se finaliza la transmisión.
spimp SLAVE STATE DONE WAIT	—	*error*=SPI ERROR INVALID END	spimp SLAVE STATE DONE	El emisor cierra la transmisión indicando falla. Se finaliza la transmisión.

Tabla 1.5. Máquina de estados del Slave Send

1.3.1. Definición de Valores

Aquí se presentan el valor numérico para cada uno de los bytes definidos por el protocolo utilizado en la implementación del mismo.

Nombre	**Valor**
MASTER_SEND_START	31
SET_LENGTH	21
SET_LENGTH_ACK	22
OK	200
MASTER_SEND_END	23
MASTER_SEND_ERROR	24
MASTER_SEND_CONFIRMATION_OK	25
MASTER_SEND_CONFIRMATION_INVALID	26
MASTER_SEND_END_ACK	27
MASTER_SEND_IS_DONE	28
MASTER_SEND_DONE	29
MASTER_RECEIVE_START	40
GET_LENGTH	41
GARBAGE	42
MASTER_RECEIVE_ACK	43
MASTER_RECEIVE_END	44
MASTER_RECEIVE_ERROR	45
MASTER_RECEIVE_END_ACK	46
MASTER_RECEIVE_CONFIRMATION_OK	47
MASTER_RECEIVE_CONFIRMATION_INVALID	80
MASTER_RECEIVE_DONE	48
MASTER_RECEIVE_IS_DONE	49

Tabla 1.6. Valores asignados a los bytes del **spimp**

1.4. Apéndice D: Tablas de Telemetría

En esta sección se presentan las tablas de datos enviados por telemetría en ambos módulos.

1.4.1. Telemetría de EMS

Variable	**Tipo**	**Tamaño (bits)**	**Unidad**	**Calibracion**	**Notas**
VCELL_Z	uint	16	Adimensionado	N/A	Tension de celdas Z
VCELL_Y	uint	16	Adimensionado	N/A	Tension de celdas Y
VCELL_X	uint	16	Adimensionado	N/A	Tension de celdas X
BAT+_S	uint	16	Adimensionado	N/A	Tension de las baterias
TANZA	uint	16	Adimensionado	N/A	Confirmacion de antenas desplegadas
I_I2C	uint	16	Adimensionado	N/A	Corriente consumida por I2C del satelite. A confirmar.
V_TEMP	uint	16	Adimensionado	N/A	Temperatura del SGE

Continúa en la siguiente página

Variable	Tipo	Tamaño (bits)	Unidad	Calibracion	Notas
I_BCN	uint	16	Adimensionado	N/A	Corriente consumida por BeaCoN. A confirmar./ Si se muestrea mas rapido que el tiempo de punto
VOUT_MPPT_Y_S	uint	16	Adimensionado	N/A	Tension salida de MPPT Y
I_V_CELL_Z	uint	16	Adimensionado	N/A	Corriente de celdas Z
VOUT_MPPT_Z_S	uint	16	Adimensionado	N/A	Tension salida de MPPT Z
V_REG_BS2_S	uint	16	Adimensionado	N/A	Tension de regulador de banda S 2
I_BS2	uint	16	Adimensionado	N/A	Corriente de banda S 2
V_REG_BS1_S	uint	16	Adimensionado	N/A	Tension de regulador banda S 1
I_BS1	uint	16	Adimensionado	N/A	Corriente de banda S 1
I_CA	uint	16	Adimensionado	N/A	Corriente control de actitud
V_REG_CA_S	uint	16	Adimensionado	N/A	Tension regulador control de actitud
I_CANTEL	uint	16	Adimensionado	N/A	Corriente de control de antel
V_REG_CANTEL_S	uint	16	Adimensionado	N/A	Tension de regulador de control de antel
I_V_CELL_Y	uint	16	Adimensionado	N/A	Corriente de celdas Y
VOUT_MPPT_X_S	uint	16	Adimensionado	N/A	Tension salida de MPPT X
I_V_CELL_X	uint	16	Adimensionado	N/A	Corriente de celdas X
V_REG_SGE_S	uint	16	Adimensionado	N/A	Tension de regluador de SGE
I_SGE	uint	16	Adimensionado	N/A	Corriente de SGE
V_REG_CP_S	uint	16	Adimensionado	N/A	Tension de regulador control principal
I_CP	uint	16	Adimensionado	N/A	Corriente de control principal
V_REG_CM1_S	uint	16	Adimensionado	N/A	Tension de regulador de comunicacion 1
I_CM1	uint	16	Adimensionado	N/A	Corriente de comunicacion 1
V_REG_CM2_S	uint	16	Adimensionado	N/A	Tension de regulador de comunicacion 2
DA_MUX1	uint	16	Adimensionado	N/A	Salida A del MUX1 hacia el micro
DB_MUX1	uint	16	Adimensionado	N/A	Salida B del MUX1 hacia el micro
I_CM2	uint	16	Adimensionado	N/A	Corriente de comunicacion 2

Tabla 1.7. Datos enviados por telemetría de EMS

1.4.2. Telemetría de ADCS

Variable	Tipo	Tamaño (bits)	Unidad	Calibracion	Notas
ADCS_MODE	char	8	N/A	Ver sheet con sugerencia	
Error vector	uint	16	N/A	Ver sheet con sugerencia	
Gyro X measurement	uint	16	N/A	Se manda medida cruda del ADC.	
Gyro Y measurement	uint	16	N/A	Se manda medida cruda del ADC.	
Gyro Z measurement	uint	16	N/A	Se manda medida cruda del ADC.	
Gyro X TEMP measurement	uint	16	Celsius	Se manda medida cruda del ADC.	
Gyro Y TEMP measurement	uint	16	Celsius	Se manda medida cruda del ADC.	
Gyro Z TEMP measurement	uint	16	Celsius	Se manda medida cruda del ADC.	
Sun sensor +X measurement	uint	16	mV	Y = (2.5/4095) * X	
Sun sensor -X measurement	uint	16	mV	Y = (2.5/4095) * X	
Sun sensor +Y measurement	uint	16	mV	Y = (2.5/4095) * X	
Sun sensor -Y measurement	uint	16	mV	Y = (2.5/4095) * X	
Sun sensor +Z measurement	uint	16	mV	Y = (2.5/4095) * X	
Sun sensor -Z measurement	uint	16	mV	Y = (2.5/4095) * X	
Magnetometer X measurement	uint	16	uT	y = X / 31.24	
Magnetometer Y measurement	uint	16	uT	y = X / 31.24	
Magnetometer Z measurement	uint	16	uT	y = X / 31.24	
Magnetometer On/Off	booleano	1		N/A	
Temperatura ADCS MSP	int	16		Se manda medida de cruda del ADC.	para parsearla la cuenta deberia ser temp_deg = (int)(((res_ts - 1857) * 666) / 4096);
Magnetotorquers On/Off Flag	booleano	1		On/Off	
Gyro On/Off Flag	booleano	1		On/Off	
Sun sensor On/Off Flag	booleano	1		On/Off	
Magnetotorquer X current command	int	16	mA	Y = X * 0.1	capaz int necesario y mandar uA

Continúa en la siguiente página

Variable	**Tipo**	**Tamaño (bits)**	**Unidad**	**Calibracion**	**Notas**
Magnetotorquer Y current command	int	16	mA	Y = X * 0.1	
Magnetotorquer Z current command	int	16	mA	Y = X * 0.1	
Angular rate estimate X	int	16	rad/s	?	capaz que float/double
Angular rate estimate Y	int	16	rad/s	?	capaz que float/double
Angular rate estimate Z	int	16	rad/s	?	capaz que float/double
Roll Angle estimate	int	16	Deg	Y = X * 0.1	
Pitch Angle estimate	int	16	Deg	Y = X * 0.1	
Yaw Angle estimate	int	16	Deg	Y = X * 0.1	
Position estimate ECI X	float	32	Km		Se asume rango [-7300, 7300] Km.
Position estimate ECI Y	float	32	Km		Se asume rango [-7300, 7300] Km.
Position estimate ECI Z	float	32	Km		Se asume rango [-7300, 7300] Km.
Velocidad ECI X	float	32	Km		Se asume rango [-8,333, 8,333] Km/s.
Velocidad ECI Y	float	32	Km		Se asume rango [-8,333, 8,333] Km/s.
Velocidad ECI Z	float	32	Km		Se asume rango [-8,333, 8,333] Km/s.
Sun vector model X	float	32	Adimensionado		Rango entre [-1, 1].
Sun vector model Y	float	32	Adimensionado		Rango entre [-1, 1].
Sun vector model Z	float	32	Adimensionado		Rango entre [-1, 1].
Magnetic field model X	float	32	nT		Rango entre [-50000, 50000].
Magnetic field model Y	float	32	nT		Rango entre [-50000, 50000].
Magnetic field model Z	float	32	nT		Rango entre [-50000, 50000].
Sun vector estimate X	float	32	Adimensionado		
Sun vector estimate Y	float	32	Adimensionado		
Sun vector estimate Z	float	32	Adimensionado		

Tabla 1.8. Datos enviados por telemetría de ADCS

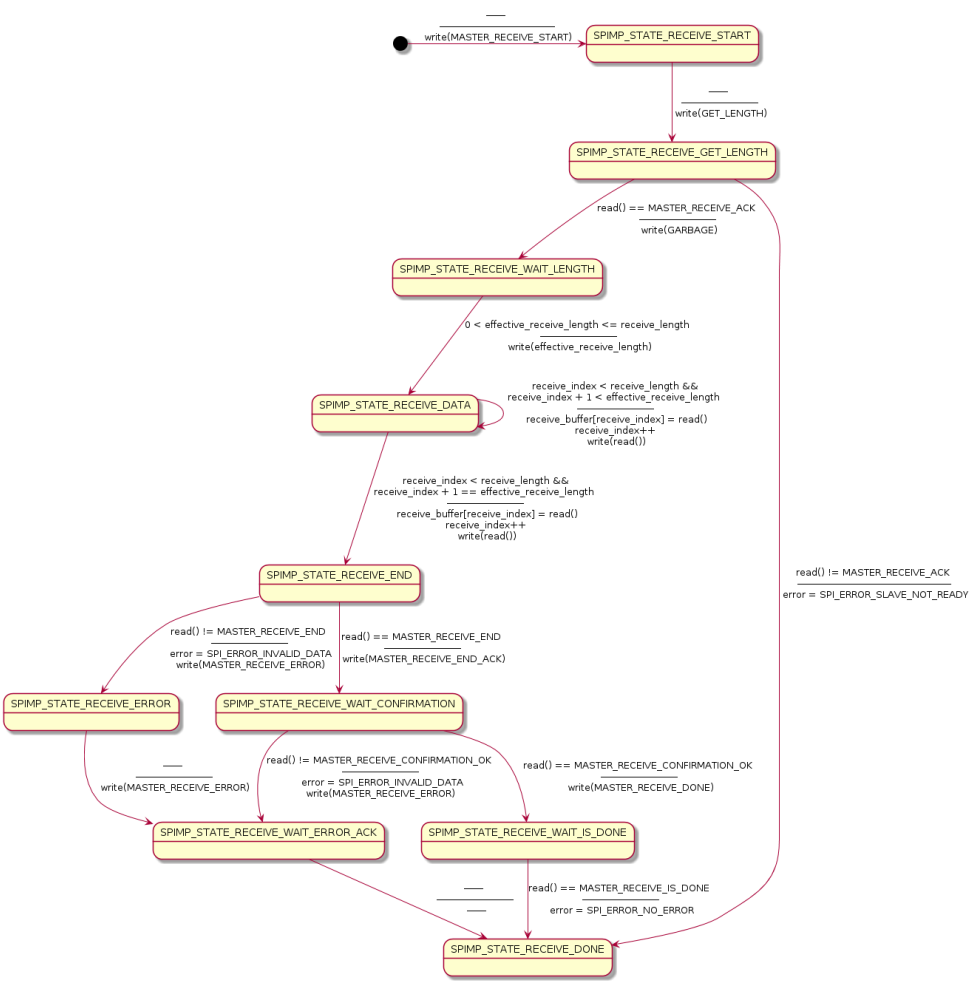

Figura 1.2. Maquina de Estados Simplificada del Master Receive

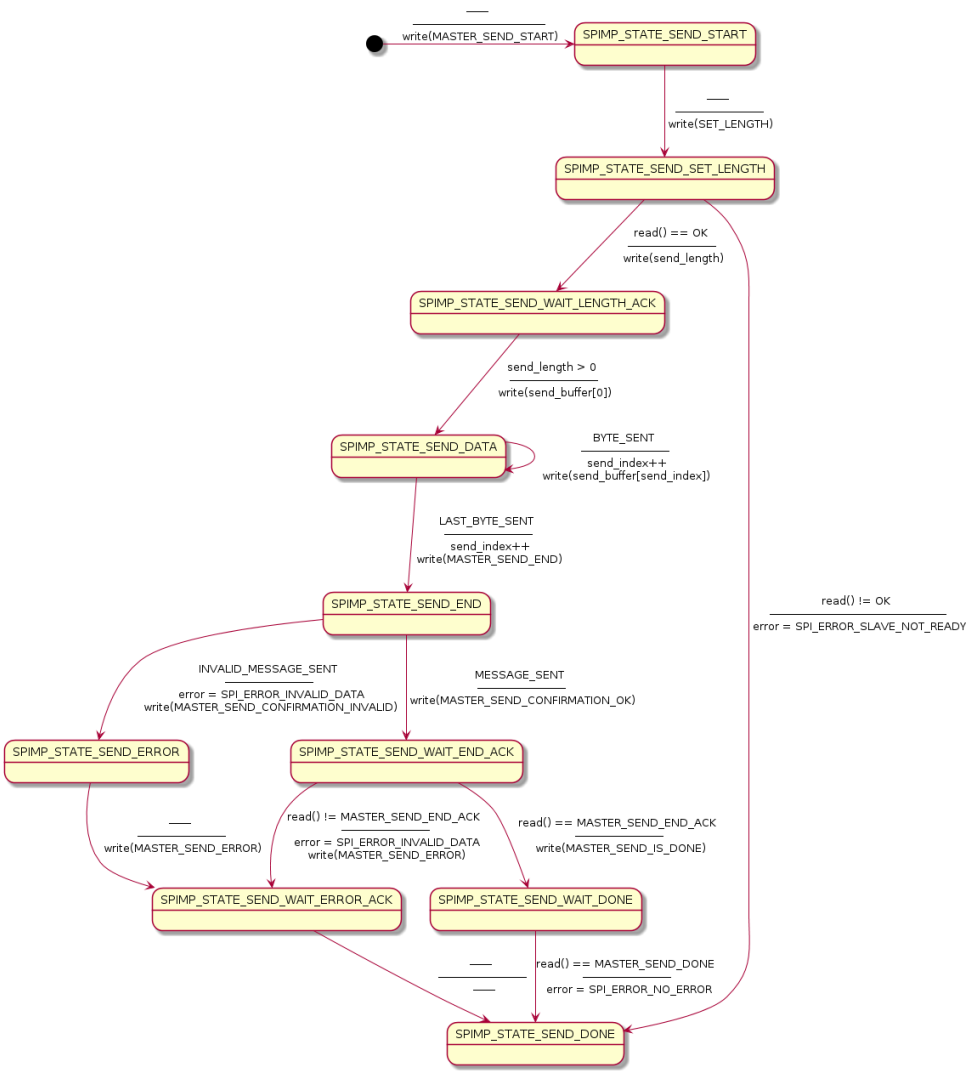

Figura 1.3. Maquina de Estados Simplificada del Master Send

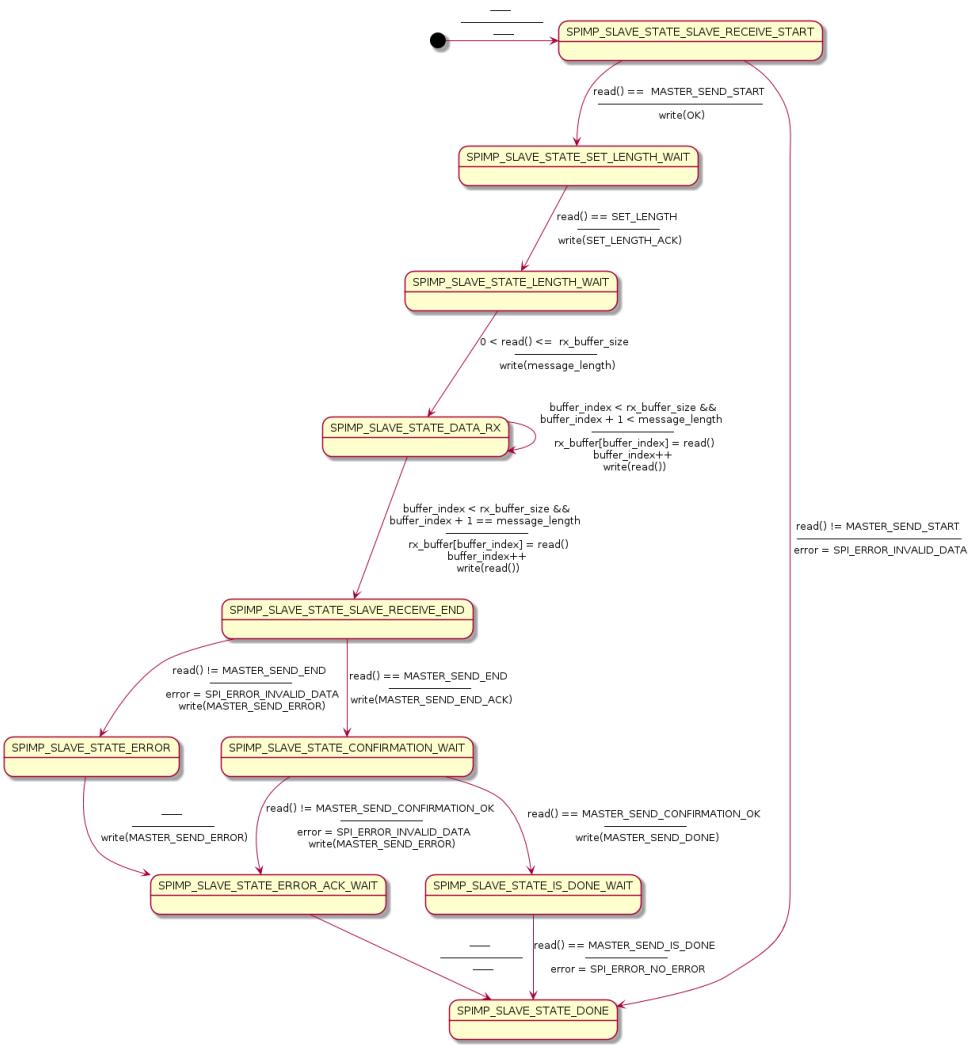

Figura 1.4. Maquina de Estados Simplificada del Slave Receive

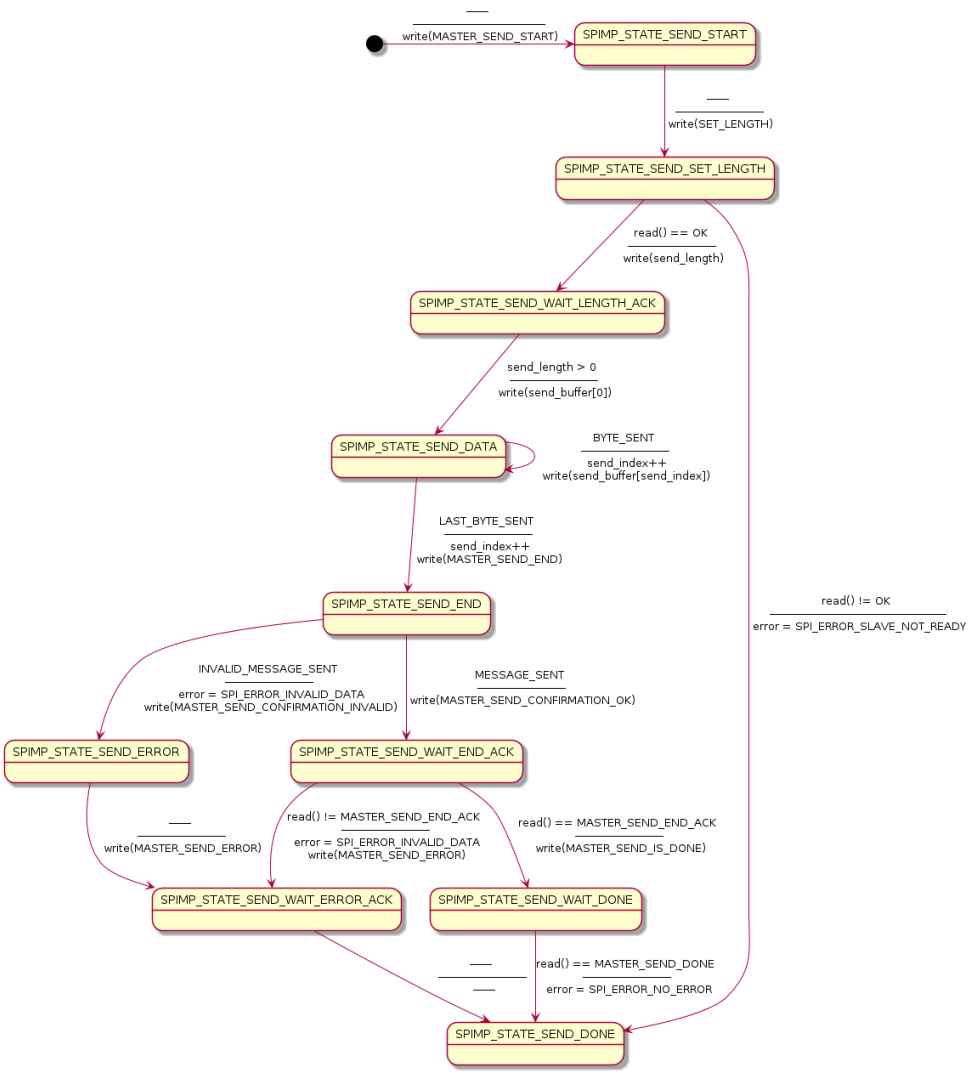

Figura 1.5. Maquina de Estados Simplificada del Slave Send