



UNIVERSIDAD DE LA REPÚBLICA
Facultad de Ingeniería
Instituto de Computación – Instituto de Ingeniería Eléctrica

# µKernel Versión 6.14

## MANUAL DEL USUARIO

AUTOR:
GUSTAVO DE MARTINO

TUTORES:
Juan Pechiar – Instituto de Ingeniería Eléctrica
Ariel Sabiguero – Instituto de Computación

TRIBUNAL:
Andrés Aguirre
Federico Rodríguez
Leonardo Steinfeld

MONTEVIDEO – URUGUAY
2013

# Resumen

Este documento describe las características, la configuración y la interfaz del kernel desarrollado en el ámbito del proyecto Antelsat para utilizar en los procesadores MSP430 de un satélite experimental. Incluye además un conjunto de ejemplos que demuestran las funcionalidades.

El desarrollo fue realizado como parte del Proyecto de Grado titulado "Software y Protocolos para Cubesat". La motivación y principales decisiones de diseño se presentan en la documentación de ese proyecto. Este documento se concentra en la herramienta y su uso.

Esta parte del trabajo se presenta por separado entendiendo que puede llegar a ser útil para otros desarrollos y con fines educativos.

# Tabla de Contenido

# Glosario

| | |
|---|---|
| Contexto | Conjunto de datos asociado a un proceso compuesto por los registros del procesador y contenido del stack. |
| kernel | Conjunto de piezas de software fundamentales de un sistema operativo encargadas de gestionar los recursos, esencialmente el procesador y generalmente la memoria y los dispositivos. |
| PCB | Process Control Block: Conjunto de datos que el sistema operativo utiliza para gestionar un proceso particular. Incluye el contexto del proceso cuando está suspendido. |
| RCB | Resource Control Block: Conjunto de datos que el sistema operativo utiliza para gestionar un recurso. Incluye las colas de procesos bloqueados por el recurso. |
| watchdog | Mecanismo de seguridad basado en un temporizador que provoca un reinico del sistema ante una situación anómala. |
| MISRA | Motor Industry Software Reliability Association. |

# Capítulo 1.

## Generalidades

### 1.1 Introducción

Este "kernel" o núcleo de sistema operativo fue construido en el ámbito del desarrollo de software para un satélite experimental. Las características de su diseño están fuertemente relacionadas con los requerimientos de la misión.

A pesar de eso, en la construcción se ha considerado la portabilidad y la capacidad de adaptación a otros requerimientos.

El ámbito de ejecución de este sistema es muy hostil y es altamente probable que exista alteración de datos a nivel de bits tanto en memoria RAM como en los registros de la CPU y sus periféricos. El sistema está por lo tanto diseñado recuperarse de fallos y usa el reinicio como herramienta básica de restauración.

Los sistemas operativos de propósito general suelen gestionar el uso del procesador, la memoria y el almacenamiento, proporcionar una interfaz para el acceso de los programas a los dispositivos, y proveer mecanismos de sincronización, de comunicación entre procesos y de seguridad.

Los sistemas operativos de tiempo real, en cambio, limitan en general su alcance a las funciones más esenciales y se enfocan en gestionar el uso del procesador para asegurar el tiempo de ejecución de las aplicaciones críticas.

Este sistema en particular no puede clasificarse estrictamente dentro de ninguna de esas dos categorías puesto que se trata de un kernel no expropiativo (o colaborativo) que provee las funcionalidades más básicas, de allí que lleve el nombre de µKernel.

### 1.2 Características del µKernel

Este kernel está desarrollado para ser utilizado en un sistema embebido en el que los procesos se ejecutan en forma permanente. No existe por lo tanto el concepto de finalización de un programa. Cada proceso tiene asociado un conjunto de datos de control llamado PCB –por *"Process Control Block"*- y un stack independiente.

Al iniciar un proceso, este recibe una dirección de memoria como parámetro. Esto permite que existan varias instancias del mismo código ejecutando diferentes procesos.

Este kernel provee los mecanismos para sincronizar procesos y transferir datos. Estos mecanismos son provistos a través de entidades llamadas recursos.

Los espacios de transferencia de datos se encuentran en la memoria de los procesos o en la memoria global. El kernel no tiene memoria para retener datos salvo las estructuras básicas para administrar los procesos y recursos.

### 1.2.1 Algoritmo de Planificación

Cuando más de un proceso está en condiciones de ejecutarse, el kernel debe elegir uno de ellos para asignarle el procesador. Para escogerlo emplea un algoritmo de planificación basado en prioridad estática.

A cada proceso se le asigna una prioridad en el momento de la carga. Los procesos de mayor prioridad tienen preferencia sobre los de menor. La prioridad queda definida por el orden de carga, y por lo tanto, no hay dos procesos con la misma prioridad. El proceso que se carga primero, es el que tiene mayor prioridad. Si bien la prioridad de un proceso es fija, el proceso puede relegarse temporalmente a la más baja.

Otra característica importante es la modalidad de expropiación. Este kernel es no expropiativo, lo que significa que los procesos pueden hacer uso del procesador tanto tiempo como necesiten antes de cederlo a otro proceso.

El programador debe definir un tiempo máximo de ejecución para cada proceso en el momento de la carga. Este tiempo se utilizará para configurar un temporizador *"watchdog"* cada vez que se entregue el procesador al proceso.

Si el proceso no entrega el procesador antes de que el tiempo expire, el *"watchdog"* reiniciará la CPU.

## 1.3 Recursos

Los recursos son entidades controladas por el kernel que permiten a las aplicaciones sincronizarse y transferir datos.

Un recurso puede entenderse como el encapsulamiento de un espacio de memoria compartida en un esquema de múltiples productores y un consumidor, sumado a un mecanismo de sincronización para que el consumidor reciba los datos luego de cada acción de un productor.

Cada recurso tiene un conjunto de datos asociado. Este conjunto de datos se conoce como RCB por *"Resource Control Block"*.

Existe un único recurso distinguido que lleva el identificador 0, al que llamamos recurso IDLE.

Todos los recursos son públicos y cualquier proceso puede obtenerlos.

Cuando el sistema inicia, todos los recursos están disponibles con excepción del recurso IDLE, que se inicializa bloqueado y nunca se libera.

Los procesos pueden tomar el control de un recurso utilizando la función lock. Cuando un recurso es bloqueado por un proceso se etiqueta como propiedad de este último. El proceso adquiere entonces derechos que le permiten:

- Definir un espacio de transferencia de datos asociado al recurso utilizando las funciones set_buffer y reset_buffer.
- Bloquearse a la espera del ingreso de datos al recurso utilizando la función read.
- Desbloquear el recurso poniéndolo a disposición de otros procesos utilizando la función unlock.

Cuando un proceso intenta bloquear un recurso que es propiedad de otro, el primero se suspende hasta que el recurso sea liberado. Esto permite implementar exclusión.

No hay límite en la cantidad de procesos bloqueados por un recurso.

Todos los procesos pueden inyectar datos a un recurso usando la función write. La cantidad de información que puede inyectarse en cada operación depende sólo del tamaño del espacio de transferencia que definió el proceso propietario.

El propietario desconoce el origen de los datos recibidos y no existen mecanismos para evitar que varios procesos inyecten datos a un recurso. De ser necesario, el programador debe gestionar la exclusión para lo que puede utilizar otro recurso.

Cuando un proceso intenta inyectar datos a un recurso y este no está listo, el proceso se suspende hasta que el recurso esté en condiciones de recibir los datos.

Existe una versión no bloqueante de write llamada writei, diseñada para que las ISR puedan disparar eventos y entregar datos.

Los recursos proveen un flujo de datos unidireccional dirigido al propietario. Para realizar una comunicación bidireccional deben utilizarse dos recursos, uno en cada sentido.

### 1.3.1 Estados de un recurso

Al inicializar el kernel, todos los recursos salvo IDLE, se configuran en estado *"FREE"*. Usando la función lock, el recurso pasa al estado *"LOCKED"*.

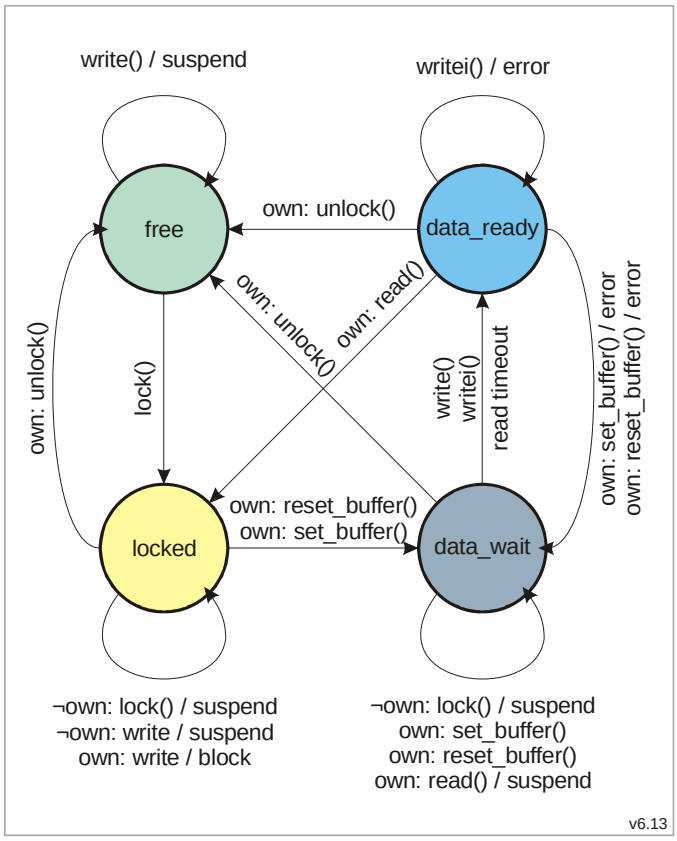


Figura 1 – Estados de un recurso

El propietario del recurso puede entonces establecer un buffer para recibir datos a través del recurso. Una vez definido el buffer, el recurso pasa a estado *"DATA_WAIT"*.

Cuando otro proceso inyecta datos al recurso o si el propietario ejecuta un read con tiempo límite distinto de cero y este se alcanza, el recurso pasa a estado *"DATA_READY"*.

Cuando el propietario obtiene los datos usando la función read, el recurso pasa nuevamente a estado *"LOCKED"*.

El propietario puede liberar el recurso en cualquier momento usando la función unlock. Si ningún proceso está esperando para bloquear el recurso, este pasará a estado *"FREE"*, de lo contrario se mantendrá en estado *"LOCKED"* cambiando de propietario.

## 1.4 Procesos

Un proceso es una entidad controlada por el kernel que representa a una instancia de ejecución de una aplicación.

Cada proceso tiene asociada una estructura de datos en memoria llamada PCB por *"Process Control Block"*, que contiene los datos básicos para administrar la ejecución.

Los procesos se crean en estado *"READY"* durante la inicialización del kernel. Este estado cambia en función de las decisiones del despachador y las acciones realizadas por el proceso durante su ejecución. En particular, los procesos no cambian de estado por acción del tiempo.

### 1.4.1 Estados de un proceso

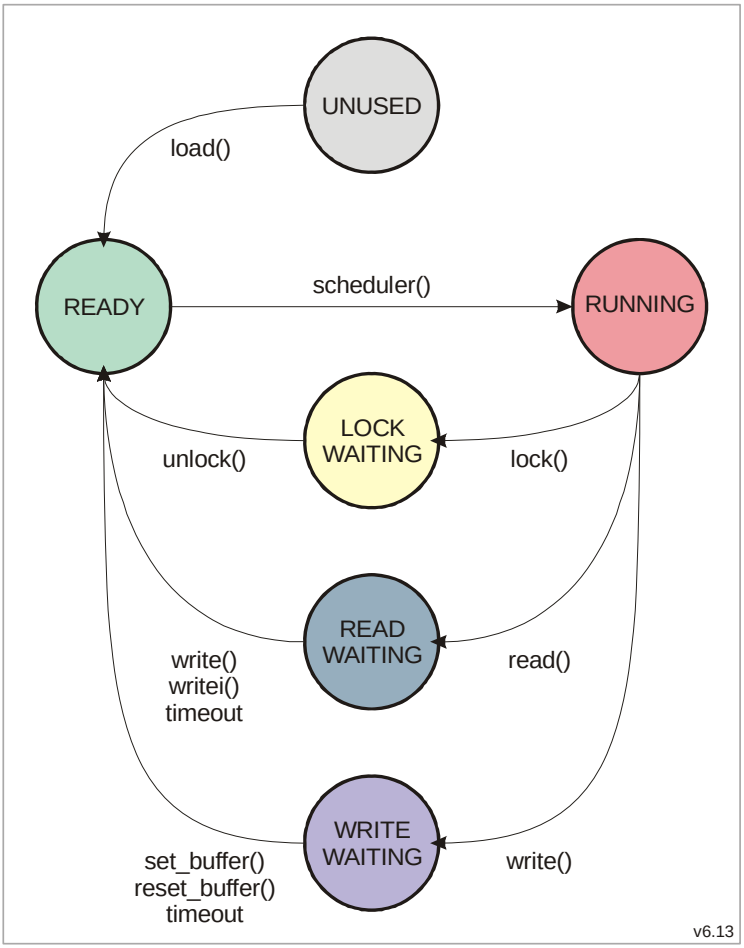


Figura 2 – Estados de un proceso

En una tabla de PCBs se registra el estado de cada posible proceso. Una vez inicializado el kernel, todos los procesos están marcados como "UNUSED".

Luego de ejecutar la función load, el contexto queda preparado para la ejecución y el estado del proceso para a ser *"READY"*.

Cuando un proceso está en ejecución, el PCB correspondiente indica como estado *"RUNNING"*.

Durante la ejecución de un proceso, este puede suspenderse. Existen tres razones por las que esto puede suceder. El proceso puede esperar para:

- Bloquear un recurso (lock).

- Obtener datos o sincronizarse (read)
- Entregar datos o sincronizarse (write)

Cada uno de estos bloqueos tiene un estado asociado en el PCB del proceso. Los estados son respectivamente:

- LOCK_WAITING
- READ_WAITING
- WRITE_WAITING

Cuando la condición que determina el bloqueo se extingue, ya sea por la acción de otros proceso o porque se alcance un tiempo límite, el proceso vuelve al estado *"READY"*.

## 1.5 Requerimientos de memoria

El espacio de memoria RAM requerido por el kernel para cada proceso es de nueve o diez palabras del procesador dependiendo si se usa o no la opción "Verificar la consistencia de datos". El mayor requerimiento de memoria se genera por el stack ya que cada proceso tiene su propio espacio de memoria a tales efectos.

El PCB almacena sólo el stack pointer del proceso suspendido. El resto del contexto se almacena en el mismo espacio extra del stack que suele proveerse para permitir el servicio de las interrupciones.

En el escenario objetivo, con un procesador de 20 bits y 12 registros de propósito general, los procesos que hacen uso mínimo de variables locales funcionan correctamente con 128 bytes de stack.

El espacio de memoria RAM requerido por el kernel para cada recurso es de seis palabras del procesador.

Además de los procesos, el kernel usa un espacio de memoria igual a cuatro palabras del procesador para alojar sus datos, mas una adicional si se configura la opción *"Control del estado del kernel"*.

## 1.6 Estados del kernel

El kernel puede estar en uno de tres estados posibles:

- No inicializado
- Inicializado
- Ejecutando

Cuando el kernel no está inicializado, la única función que puede ejecutarse es kernel_init.

Una vez inicializados los datos, puede ejecutarse la función load tantas veces como programas deseen cargarse.

La ejecución de la función run, pasa el kernel a estado ejecutando. A partir de este momento pueden llamarse todas las restantes funciones excepto las ya mencionadas.

## 1.7 Estructuras de datos del kernel

Los RCB se agrupan en una tabla, que puede tener tantos elementos como el programador entienda necesarios. Cada RCB tiene los siguientes datos:

- El estado del recurso.
- La identificación del proceso que ha bloqueado el recurso (propietario).
- Una cola de procesos suspendidos a la espera de bloquear el recurso.
- Una cola de procesos suspendidos a la espera de escribir datos al recurso.
- La dirección del espacio de transferencia asignada por el proceso propietario del recurso.
- El tamaño del espacio de transferencia.

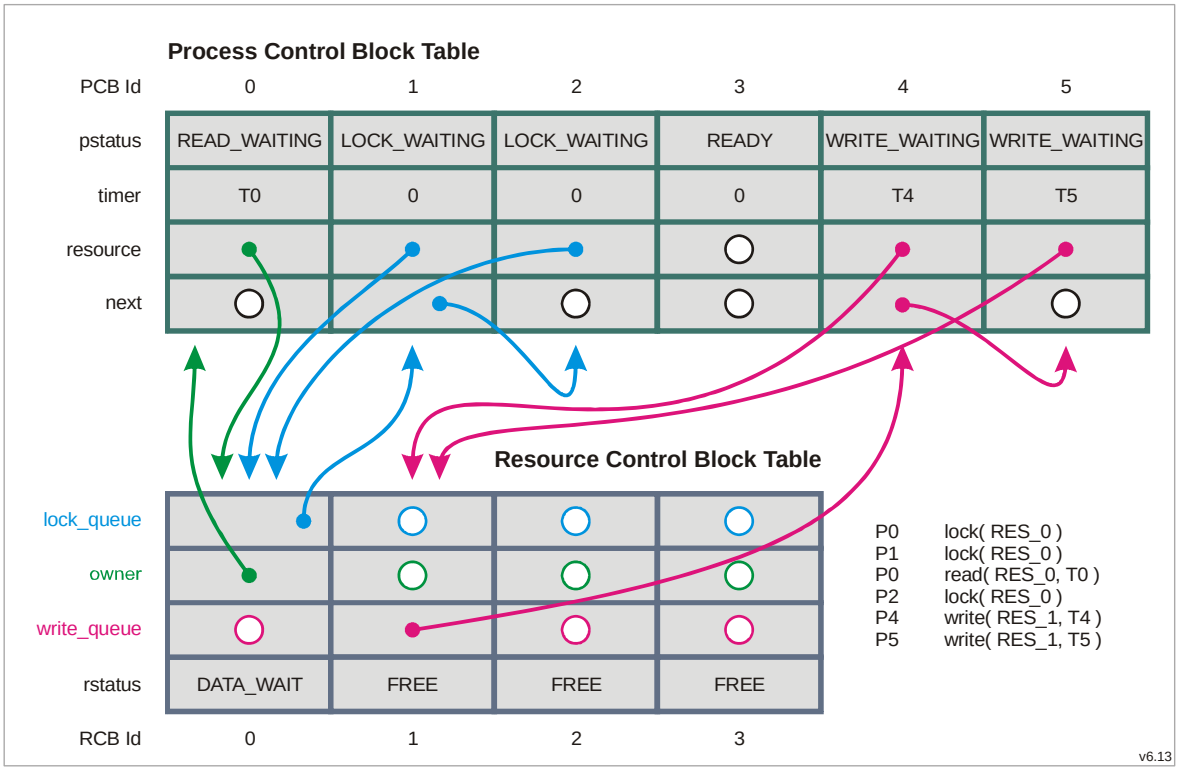


Figura 3 – Estructuras de datos del kernel

Los PCB se agrupan en una tabla que puede tener tantos elementos como procesos se necesiten. Cada PCB tiene los siguientes datos:

- El estado del proceso.
- El recurso por el que espera, sólo cuando está suspendido.
- El siguiente proceso bloqueado por el recurso.
- El tiempo de espera remanente cuando el proceso está suspendido.
- El resultado de la última operación.
- La ubicación del *"stack"*.
- El valor del *"stack pointer"* del proceso suspendido
- El final del espacio reservado para el stack

- La configuración para el temporizador *"watchdog"*.
- El CRC16 de los datos del *"stack"*.

En la figura 3 se muestran la tabla de PCB, la tabla de RCB y sus conexiones. Nótese que las estructuras son de tamaño fijo y a pesar de esto, permiten alojar tantas colas como dos veces la cantidad de recursos, donde cada una puede tener tantos elementos como la cantidad de procesos del sistema. Esta estrategia se basa en que un proceso puede estar en una única cola a la vez.

Por otra parte, el kernel maneja algunas variables como la cantidad de procesos cargados -llamada *"process_count"*-, la identificación del proceso en curso -llamada *"pid"*-, la cantidad de intervalos de tiempo pendientes de procesar -llamada "tics"- y la dirección del stack pointer del despachador.

## 1.8 El despachador

Cada vez que un proceso se bloque el despachador toma el control de la CPU. Este realiza una serie de acciones.

Si está configurado, realiza el control de desbordamiento del stack del proceso que está dejando el procesador.

Si está configurado, calcula y almacena el CRC de la porción utilizada del stack del proceso que está dejando el procesador.

Posteriormente, por cada tic registrado, decrementa el contador de cada proceso suspendido cuyo temporizador sea mayor a cero.

Cuando uno de estos temporizadores llega a cero, pone el proceso en estado "READY", establece el valor de salida en *"TIMEOUT"* y lo retira de la cola de espera correspondiente.

Una vez que se han revisado todos los procesos, el kernel busca el primero en estado "READY" en orden de prioridad.

Si no encuentra ninguno, busca procesos en la cola de bloqueo del recurso IDLE.

Se aun así no encuentra un proceso al que entregarle el procesador, programa el temporizador *"watchdog"* con el valor configurado para bajo consumo de energía y pasa el procesador al estado LPM (Low Power Mode).

Si encontró algún proceso en estado "READY", lo cambia a estado "RUNNING", programa el temporizador *"watchdog"* con el valor definido para ese proceso y le entrega el control restaurando su contexto.

Si está configurado, antes de programar el temporizador *"watchdog",* controla el CRC de la porción utilizada del "stack".

## 1.9 Temporización

El kernel requiere que el hardware incremente periódicamente una variable global –llamada *"tics"*–. Esta variable es utilizada para todas las temporizaciones del kernel.

Para minimizar el consumo de energía debe maximizarse este período.

Nombramos "tic" al tiempo transcurrido entre dos incrementos de este contador.

## 1.10 Stack

El kernel utiliza el stack definido por el linker (stack del main). Los espacios de memoria para los stack de los procesos pueden estar definidos tanto como variables globales como en variables locales del main.

Si se usa el entorno de desarrollo provisto por el fabricante del microcontrolador [8], es conveniente que se utilice la segunda opción ya la herramienta es capaz de verificar y reportar si el stack pointer está apuntado fuera del rango predefinido.

# Capítulo 2.

## Configuración

### 2.1 Archivo de configuración

La configuración del kernel se realiza mediante un archivo de nombre “ukernel_config.h”.

Este archivo debe contener algunas definiciones obligatorias y puede contener otras opcionales. A continuación se describe cada una de esas definiciones.

### 2.2 Cantidad de recursos

Es necesario definir la cantidad de recursos que el kernel puede utilizar con el valor de la variable del precompilador *“MAX_RESOURCE_COUNT”*. Esta variable se utiliza para definir el tamaño de la tabla de RCB y para determinar si el identificador de un recurso es válido.

Para definir por ejemplo 17 recursos, se debe incluir en el archivo de configuración la declaración:

```
#define MAX_RESOURCE_COUNT 17
```

### 2.3 Cantidad de procesos

Es necesario definir la cantidad de procesos que el kernel puede ejecutar con el valor de la variable del precompilador *“MAX_PROGRAM_COUNT”*. Esta variable se utiliza para definir el tamaño de la tabla de PCB y para limitar la cantidad de procesos declarados con la función load.

Para definir por ejemplo 12 procesos, se debe incluir en el archivo de configuración la declaración:

```
#define MAX_PROGRAM_COUNT 12
```

### 2.4 Configuración para el watchdog en LPM

Es necesario definir el parámetro con el que se configurará el temporizador “watchdog” cuando el kernel decida pasar a modo de bajo consumo de energía. Este valor debe definirse en la variable del precompilador “LMP_WATCHDOG_CONFIG”.

La operación del watchdog es externa al kernel y debe ser provista por el programador, por lo que los valores de la configuración dependen de esa implementación.

Para definir por ejemplo el valor "4", debe incluirse en el archivo de configuración la declaración:

```
#define LMP_WATCHDOG_CONFIG 4
```

## 2.5 Control de desbordamiento del stack

Si se define *"CONTROL_STACK_OVERFLOW"*, el kernel inserta valores de control en los extremos del stack.

Al cambiar de contexto, cuando el programa pierde el control, el kernel verifica si estos valores fueron modificados y de ser así genera un reset.

Para incluir el control de desbordamiento el archivo de configuración debe incluir la definición:

```
#define CONTROL_STACK_OVERFLOW
```

## 2.6 Control del estado del kernel

Si se define *"CONTROL_KERNEL_STATUS"*, todas las funciones controlarán el estado del kernel previo a su ejecución y en caso de error devolverán un resultado indicándolo. Las funciones que retornan error_t devolverán *"ERROR_INVALID_STATUS"*. Las funciones que retornan resource_status_t devolverán *"RESOURCE_STATUS_INVALID"*.

Para incluir el control de estado el archivo de configuración debe incluir la definición:

```
#define CONTROL_KERNEL_STATUS
```

## 2.7 Control de consistencia de datos locales

Si se define *"CONTROL_STACK_CONSISTENCY"*, el kernel agrega un valor en el PCB para almacenar el CRC de los datos almacenados en el stack.

Cada vez que el programa delega la CPU y antes de entregarle el control, el kernel calcula el CRC de los datos y si fueron modificados genera un reset.

Cuando se utiliza este control de consistencia los datos no pueden ser modificados mientras el programa está suspendido. Las funciones como read y write, exponen espacios de memoria para otros procesos: read expone buffer y largo de datos, write expone la cantidad de bytes recibidos. Las variables que contienen estos datos no pueden ser locales al proceso.

Para incluir el control de consistencia de datos locales el archivo de configuración debe incluir la definición:

```
#define CONTROL_STACK_CONSISTENCY
```

## 2.8 Múltiples copias del código

Si se define *"KERNEL_COPY"*, el compilador colocará tres copias de cada una de las funciones del kernel.

Esta definición es opcional, pero su inclusión hace obligatorias las definiciones de largo y CRC de cada una de las funciones.

El largo del código es dependiente de las opciones seleccionadas, del procesador y del modo de operación para el que se esté compilando, por lo que debe hacerse una compilación primaria para obtener los valores de tamaño y una ejecución.

Para obtener los valores de CRC pueden calcularse a partir del código generado pero la forma más sencilla es ejecutar el programa con un depurador y tomar nota de los valores calculados en tiempo de ejecución.

Si se incluyó la definición, la función kernel_init verificará el CRC de la primer copia de cada función y lo comparará con el valor definido en la constante correspondiente. Si la comparación no da valores iguales, kernel_init hará lo mismo con la segunda copia. En caso de fallar este control, kernel_init definirá como válida la tercera copia sin controlarla.

Para incluir las copias del código debe incluirse en el archivo de configuración un conjunto de definiciones como las siguientes:

```
#define KERNEL_CODE_LENGTH_LOAD1 120
#define KERNEL_CODE_LENGTH_RUN1 18
#define KERNEL_CODE_LENGTH_SCHEDULER1 438
#define KERNEL_CODE_LENGTH_GET_RESOURCE_STATUS1 28
#define KERNEL_CODE_LENGTH_LOCK1 202
#define KERNEL_CODE_LENGTH_READ1 180
#define KERNEL_CODE_LENGTH_SET_BUFFER1 144
#define KERNEL_CODE_LENGTH_RESET_BUFFER1 138
#define KERNEL_CODE_LENGTH_SLEEP1 94
#define KERNEL_CODE_LENGTH_STOP1 18
#define KERNEL_CODE_LENGTH_TSLEEP1 30
#define KERNEL_CODE_LENGTH_UNLOCK1 116
#define KERNEL_CODE_LENGTH_WRITE1 256
#define KERNEL_CODE_LENGTH_WRITEI1 158

#define KERNEL_CODE_CRC_LOAD1 0x34CF
#define KERNEL_CODE_CRC_RUN1 0x0E47
#define KERNEL_CODE_CRC_SCHEDULER1 0x9260
#define KERNEL_CODE_CRC_GET_RESOURCE_STATUS1 0xE269
#define KERNEL_CODE_CRC_LOCK1 0x6D35
#define KERNEL_CODE_CRC_READ1 0x7AAC
#define KERNEL_CODE_CRC_SET_BUFFER1 0xD490
#define KERNEL_CODE_CRC_RESET_BUFFER1 0xEAD4
#define KERNEL_CODE_CRC_SLEEP1 0xA5BB
#define KERNEL_CODE_CRC_STOP1 0xBAD1
#define KERNEL_CODE_CRC_TSLEEP1 0x3BCF
#define KERNEL_CODE_CRC_UNLOCK1 0xBCA5
#define KERNEL_CODE_CRC_WRITE1 0x6D77
#define KERNEL_CODE_CRC_WRITEI1 0x6AEA
```

```
#define KERNEL_CODE_LENGTH_LOAD2 120
#define KERNEL_CODE_LENGTH_RUN2 18
#define KERNEL_CODE_LENGTH_SCHEDULER2 438
#define KERNEL_CODE_LENGTH_GET_RESOURCE_STATUS2 28
#define KERNEL_CODE_LENGTH_LOCK2 202
#define KERNEL_CODE_LENGTH_READ2 180
#define KERNEL_CODE_LENGTH_SET_BUFFER2 144
#define KERNEL_CODE_LENGTH_RESET_BUFFER2 138
#define KERNEL_CODE_LENGTH_SLEEP2 94
#define KERNEL_CODE_LENGTH_STOP2 18
#define KERNEL_CODE_LENGTH_TSLEEP2 30
#define KERNEL_CODE_LENGTH_UNLOCK2 116
#define KERNEL_CODE_LENGTH_WRITE2 256
#define KERNEL_CODE_LENGTH_WRITEI2 158

#define KERNEL_CODE_CRC_LOAD2 0x34CF
#define KERNEL_CODE_CRC_RUN2 0x0E47
#define KERNEL_CODE_CRC_SCHEDULER2 0x9260
#define KERNEL_CODE_CRC_GET_RESOURCE_STATUS2 0xE269
#define KERNEL_CODE_CRC_LOCK2 0x6D35
#define KERNEL_CODE_CRC_READ2 0x7AAC
#define KERNEL_CODE_CRC_SET_BUFFER2 0xD490
#define KERNEL_CODE_CRC_RESET_BUFFER2 0xEAD4
#define KERNEL_CODE_CRC_SLEEP2 0xA5BB
#define KERNEL_CODE_CRC_STOP2 0xBAD1
#define KERNEL_CODE_CRC_TSLEEP2 0x3BCF
#define KERNEL_CODE_CRC_UNLOCK2 0xBCA5
#define KERNEL_CODE_CRC_WRITE2 0x6D77
#define KERNEL_CODE_CRC_WRITEI2 0x6AEA
```

## 2.9 Nombres de los recursos

No es estrictamente necesario pero resulta muy recomendable incluir la definición de los nombres de los recursos en el archivo de configuración.

Para asignarle por ejemplo el nombre *"RESOURCE_AX25_RX_INPUT"* al recurso número 5 el archivo de configuración debe incluir la definición:

```
#define RESOURCE_AX25_RX_INPUT 5
```

# Capítulo 3.

## API

### 3.1 Introducción

La API del kernel provee dos tipos de funciones. Un conjunto para la inicialización del sistema y otro para el uso de las aplicaciones.

Las funciones de inicialización son:

- kernel_init — Para inicializar las estructuras de datos.
- load — Para cargar programas.
- run — Para transferir el control al despachador.

Las funciones para uso de las aplicaciones son:

- get_resource_status — Para conocer el estado de un recurso.
- lock — Para tomar el control de un recurso.
- unlock — Para liberar un recurso.
- set_buffer — Para establecer el espacio de transferencia asociado a un recurso.
- reset_buffer — Para restablecer el espacio de transferencia.
- read — Para recibir datos a través de un recurso.
- write — Para entregar datos a través de un recurso.
- writei — Para entregar datos a través de un recurso durante una ISR.
- tsleep — Para suspender el proceso un tiempo expresado en tics.
- sleep — Para suspender el proceso un tiempo expresado en segundos.
- stop — Para detener permanentemente un proceso.

### 3.2 Funciones de inicialización

#### 3.2.1 kernel_init

```
error_t kernel_init( void )
```

#### 3.2.1.1 Descripción

La función kernel_init controla la integridad del código[1] e inicializa los datos del kernel. Esta función no inicializa el hardware y debe ejecutarse antes de cargar los programas con load.

Su ejecución cambia el estado del kernel a *"INITIALIZED"*.

#### 3.2.1.2 Parámetros

*Ninguno*

#### 3.2.1.3 Retorna

| | |
|---|---|
| *NO_ERROR* | Si el proceso fue exitoso. |
| *INVALID_STATUS* | Si el kernel ya se había inicializado. |

### 3.2.2 load

```
error_t load  (  void (*program) (void*), void* parameters, stack_t* stack,
                 const size_t stack_length, const unsigned short wd_config )
```

#### 3.2.2.1 Descripción

La función load inicializa el contexto del proceso. Cuando este inicie, recibirá como parámetro el valor de *"parameters"* y el stack estará ubicado en las direcciones más altas del intervalo definido por *"stack"* y *"stack_length"*.

#### 3.2.2.2 Parámetros

| | |
|---|---|
| *program* | Punto de entrada del programa a ejecutar. |
| *parameters* | Dirección de memoria que se entregará al programa como único parámetro. |
| *stack* | Dirección más baja del espacio de memoria RAM reservado para el stack del proceso. |
| *stack_lenght* | Tamaño del stack expresado en palabras (stack_t). |
| *wd_config* | Valor con el que se configurará el temporizador *watchdog* cada vez que se asigne el procesador al proceso. |

#### 3.2.2.3 Retorna

| | |
|---|---|
| *NO_ERROR* | Si el proceso fue exitoso. |

[1] Opcionalmente. Ver en la sección "Múltiples copias del código" del Capítulo 2.

| | |
|---|---|
| *TOO_MANY_PROGS* | Si se ha excedido la cantidad máxima de programas[2]. |
| *INVALID_STATUS* | Si no se realizó la inicialización o ya se pasó a modo ejecución. |

#### 3.2.3 run

```
error_t run ( void )
```

##### 3.2.3.1 Descripción

Ejecuta el despachador, retorna solo en caso de error. Su ejecución cambia el estado del kernel a *"RUNNING"*

##### 3.2.3.2 Parámetros

*Ninguno*

##### 3.2.3.3 Retorna

| | |
|---|---|
| *INVALID_STATUS* | Si no se realizó la inicialización o ya se pasó a modo ejecución. |
| *NO_PROGS* | Si no se cargaron programas. |

### 3.3 Funciones para uso de las aplicaciones

#### 3.3.1 get_resource_status

```
resource_status_t get_resource_status ( const resource_t resource )
```

##### 3.3.1.1 Descripción

La función get_resource_status informa acerca del estado de un recurso

##### 3.3.1.2 Parámetros

| | |
|---|---|
| *resource* | Identificador del recurso que se desea investigar |

##### 3.3.1.3 Retorna

El estado del recurso o "*INVALID*" si el identificador del recurso no es válido.

---

[2] Ver en la sección "Cantidad de procesos" del Capítulo 2.

### 3.3.2 lock

```
error_t lock ( const resource_t resource, const time_t timeout )
```

#### 3.3.2.1 Descripción

La función lock intenta obtener el control sobre un recurso. Si el recurso está bloqueado por otro proceso, el proceso llamador se agrega a la cola de los que requieren el recurso y se suspende hasta que este esté disponible o hasta que se alcance el tiempo indicado en *"timeout"*.

La ejecución con salida "*NO_ERROR*" cambia el estado del recurso a "LOCKED".

#### 3.3.2.2 Parámetros

| | |
|---|---|
| *resource* | Identificador del recurso. |
| *timeout* | Tiempo máximo de espera expresado en tics. Si timeout es 0 la espera será hasta que el recurso esté disponible. |

#### 3.3.2.3 Retorna

| | |
|---|---|
| *NO_ERROR* | Si el proceso resultó exitoso. |
| *INVALID_STATUS* | Si el kernel no se ha inicializado. |
| *INVALID_RESOURCE* | Si el recurso no es válido. |
| *TIMEOUT* | Si expiró el tiempo. |
| *ALREADY_LOCKED* | Si ese proceso ya tiene bloqueado el recurso. |

### 3.3.3 unlock

```
error_t unlock ( const resource_t resource )
```

#### 3.3.3.1 Descripción

La función unlock libera un recurso previamente bloqueado para que otros procesos puedan utilizarlo. Nunca bloquea. Si existían procesos bloqueados a la espera por el recurso, unlock le asigna el recurso al primero en la cola.

#### 3.3.3.2 Parámetros

| | |
|---|---|
| *resource* | Identificador del recurso a liberar |

#### 3.3.3.3 Retorna

| | |
|---|---|
| *NO_ERROR* | Si el proceso resultó exitoso. |
| *INVALID_STATUS* | Si el kernel no se ha inicializado. |

| | |
|---|---|
| *INVALID_RESOURCE* | Si el recurso no es válido. |
| *NOT_LOCKED* | Si el recurso no estaba bloqueado. |
| *INVALID_OWNER* | Si el recurso está bloqueado por otro proceso. |

### 3.3.4 set_buffer

```
error_t set_buffer ( const resource_t resource, void* buffer,
                     const size_t buffer_length )
```

#### 3.3.4.1 Descripción

La función set_buffer define el espacio de transferencia de datos asociado al recurso referenciado. El recurso debe haber sido bloqueado por el proceso llamador. Si había datos sin leer en el buffer, reporta el error y define el buffer con los nuevos parámetros.

#### 3.3.4.2 Parámetros

| | |
|---|---|
| *resource* | Identificador del recurso al que se quiere asignar el espacio de transferencia. |
| *buffer* | Dirección del espacio de transferencia. |
| *buffer_length* | Tamaño del espacio de transferencia expresado en bytes. |

#### 3.3.4.3 Retorna

| | |
|---|---|
| *NO_ERROR* | Si el proceso resultó exitoso. |
| *INVALID_STATUS* | Si el kernel no se ha inicializado. |
| *INVALID_RESOURCE* | Si el recurso no es válido. |
| *NOT_LOCKED* | Si el recurso no estaba bloqueado. |
| *INVALID_OWNER* | Si el recurso está bloqueado por otro proceso. |
| *ERROR_DATA_LOST* | Si al establecer el buffer se perdieron datos. |

### 3.3.5 reset_buffer

```
error_t reset_buffer ( const resource_t resource )
```

#### 3.3.5.1 Descripción

La función reset_buffer restablece el espacio de transferencia de datos asociado a un recurso. El recurso debe haber sido bloqueado por el proceso llamador. Si había datos sin leer en el buffer, reporta el error y define el buffer con los nuevos parámetros.

Si el proceso ya había ejecutado set_buffer, reset_buffer restablecerá el mismo buffer. Si la función se llama luego de lock, se establecerá un espacio de transferencia nulo y de largo cero, apto sólo para sincronización.

#### 3.3.5.2 Parámetros

| | |
|---|---|
| *resource* | Identificador del recurso al que se quiere asignar el espacio de transferencia. |

#### 3.3.5.3 Retorna

| | |
|---|---|
| *NO_ERROR* | Si el proceso resultó exitoso. |
| *INVALID_STATUS* | Si el kernel no se ha inicializado. |
| *INVALID_RESOURCE* | Si el recurso no es válido. |
| *NOT_LOCKED* | Si el recurso no estaba bloqueado. |
| *INVALID_OWNER* | Si el recurso está bloqueado por otro proceso. |
| *ERROR_DATA_LOST* | Si al establecer el buffer se perdieron datos. |

### 3.3.6 read

```
error_t read ( const resource_t resource, size_t* data_length,
               const time_t timeout )
```

#### 3.3.6.1 Descripción

La función read bloquea el proceso hasta que el recurso tenga datos listos para leer o se alcance el tiempo máximo de espera.

Esta función permite leer datos de un tamaño mayor al del buffer usando una señalización. El kernel no almacena datos. La transferencia de un flujo de datos se logra con la intervención de ambas partes: la que inyecta los datos y la que los recibe.

#### 3.3.6.2 Parámetros

| | |
|---|---|
| *resource* | Identificador del recurso por el que se desea esperar. |
| *data_length* | Dirección en la que se almacenará el largo de datos recibidos, 0 para ignorar el resultado. |
| *timeout* | Tiempo máximo de espera expresado en *"tics"*, 0 para esperar indefinidamente. |

#### 3.3.6.3 Retorna

| | |
|---|---|
| *NO_ERROR* | Si el proceso resultó exitoso. |
| *INVALID_RESOURCE* | Si el recurso no es válido |
| *NOT_LOCKED* | Si el recurso no estaba bloqueado |
| *INVALID_OWNER* | Si el recurso está bloqueado por otro proceso |
| *BUFFER_FULL* | Si quedan más datos para leer. |

| | |
|---|---|
| *INVALID_BUFFER* | Si no se ha definido el buffer de transferencia. |
| *INVALID_STATUS* | Si el kernel no se ha inicializado. |

### 3.3.7 write

```
error_t write ( const resource_t resource, const void* buffer,
          const size_t data_length, size_t* writed_length,
          const time_t timeout )
```

#### 3.3.7.1 Descripción

La función write bloquea el proceso hasta que el recurso referido se encuentre en estado de espera de datos o se alcance tiempo definido.

Si el largo de datos es mayor que el espacio del buffer definido por el proceso que espera por los datos (el que hizo el set_buffer), se copian tantos bytes como permita el buffer y se reporta el error a los dos procesos.

#### 3.3.7.2 Parámetros

| | |
|---|---|
| *resource* | El identificador del recurso por el que se desea esperar |
| *buffer* | Espacio de transferencia o 0 si no se van a transferir datos. |
| *data_length* | Largo de datos a enviar |
| *writed_length* | Dirección en la que se almacenará el largo de datos entregados, 0 para ignorar el resultado. |
| *timeout* | Tiempo máximo de espera expresado en *"tics"*, 0 para esperar indefinidamente. |

#### 3.3.7.3 Retorna

| | |
|---|---|
| *NO_ERROR* | Si el proceso resultó exitoso |
| *INVALID_RESOURCE* | Si el recurso no es válido |
| *BUFFER_FULL* | Si el espacio del buffer no permitió transferir todos los datos. |
| *INVALID_STATUS* | Si el kernel no se ha inicializado. |

### 3.3.8 writei

```
error_t writei ( const resource_t resource, const void* buffer,
               const size_t data_length, size_t* writed_length )
```

#### 3.3.8.1 Descripción

La función writei transfiere datos al buffer del recurso si este está en estado *"DATA_WAIT"*. Es análoga a la función write pero nunca bloquea. Es ideal para usar en una ISR, tanto para transferir datos como para señalizar.

#### 3.3.8.2 Parámetros

| | |
|---|---|
| *resource* | El identificador del recurso por el que se desea esperar. |
| *buffer* | Espacio de transferencia o 0 si no se van a transferir datos. |
| *data_length* | Largo de datos a enviar. |
| *writed_length* | Dirección en la que se almacenará el largo de datos entregados, 0 para ignorar el resultado. |

#### 3.3.8.3 Retorna

| | |
|---|---|
| *NO_ERROR* | Si el proceso resultó exitoso. |
| *INVALID_RESOURCE* | Si el recurso no es válido. |
| *BUFFER_FULL* | Si el espacio del buffer no permitió transferir todos los datos. |
| *BUFFER_NOT_READY* | Si el recurso no está esperando datos. |
| *INVALID_STATUS* | Si el kernel no se ha inicializado. |

### 3.3.9 tsleep

```
void (*tsleep)( const time_t timeout )
```

#### 3.3.9.1 Descripción

La función tsleep suspende el proceso durante un tiempo. El proceso pasa a estado "READY" después que el kernel detecta tantos intervalos de tiempo como los indicados en el parámetro. Luego el proceso continuará cuando no exista otro proceso listo de mayor prioridad. Esto implica que puede existir una variación del orden de ±1 tics si el procesador no está sobrecargado.

Los procesos pueden relegar su prioridad a la menor posible llamando a esta función con el parámetro 0. Esto provoca una espera por el recurso IDLE.

El tiempo está acotado por el tamaño de la palabra del procesador. Para una palabra de 16 bits y un tiempo de tic de 100 ms, la espera máxima es del orden de 109 minutos.

#### 3.3.9.2 Parámetros

| | |
|---|---|
| *timeout* | Tiempo expresado en *"tics"*. |

#### 3.3.9.3 Retorna

*Nada*

### 3.3.10 sleep

```
void sleep( unsigned long seconds )
```

#### 3.3.10.1 Descripción

La función sleep suspende el proceso durante un tiempo expresado en segundos.

Usar el valor 0 como parámetro tiene el mismo efecto que en tsleep.

El tiempo máximo queda definido por el tamaño del entero largo del sistema (unsigned long). Para entero de 32 bits y un tiempo de tic de 100 ms, la espera máxima es del orden de 49710 días (unos 136 años)

#### 3.3.10.2 Parámetros

*timeout* Tiempo expresado en segundos.

#### 3.3.10.3 Retorna

*Nada.*

### 3.3.11 stop

```
void stop (void) suspende definidamente la ejecución del proceso.
```

#### 3.3.11.1 Descripción

La función stop detiene indefinidamente un proceso.

#### 3.3.11.2 Parámetros

*Ninguno.*

#### 3.3.11.3 Retorna

*Nada.*

# Capítulo 4.

## Implementación del µKernel

### 4.1 Historial de versiones

El kernel fue diseñado y ajustado a lo largo de seis versiones y decenas de revisiones.

Cada una de ellas agregó o ajustó detalles para adaptarse a las necesidades de la misión para la que fue construido o introdujo optimizaciones.

El objetivo primario de este trabajo era satisfacer las necesidades del módulo de control principal de un satélite experimental. Este módulo está basado en un microcontrolador de Texas Instruments MSP430F5438A trabajando con direccionamiento extendido.

Los primeros resultados de su implementación promovieron su uso en los restantes módulos del satélite, algunos de estos basados en el mismo procesador pero operando con direccionamiento de MSP430 estándar, otros basados en el procesador MSP430F6638 del mismo fabricante.

Las distintas versiones se adaptaron a estos procesadores y a las necesidades de los procesos que se ejecutan sobre ellos.

Los casos particulares son:

- Dos implementaciones con altos requerimientos de capacidad de procesamiento como es el caso de un demodulador realizado por software: Módulo de comunicaciones receptor COMM1 del proyecto Antelsat, módulo de comunicaciones receptor y transmisor COMM2 del mismo proyecto.
- Una implementación con altos requerimientos de confiabilidad como es el caso del módulo de control principal (objeto del trabajo inicial).
- Una implementación con bajos requerimientos de procesamiento pero una cantidad importante de procesos: Módulo de gestión de energía del proyecto Antelsat.
- Una implementación balanceada: Módulo de control de actitud del proyecto Antelsat.

A continuación se detallan los cambios más relevantes de las revisiones y los últimos ajustes:

### 4.1.1 Versión 1

- Primera versión para MSP430F5438A con direccionamiento de 20 bits.

### 4.1.2 Versión 2

- La cola de procesos a la espera de bloquear un recurso se integra a la estructura de PCB para minimizar el conjunto de datos del kernel.
- Primera versión independiente del procesador (para ser compatible con MSP430F6638)
- Cambios en el nombre de las funciones de la API

### 4.1.3 Versión 3

- Se agrega la función writei y la cola de procesos a la espera por write integrada con la cola de procesos bloqueados.

### 4.1.4 Versión 4

- Se agrega el recurso IDLE y con él el cambio dinámico de prioridad de los procesos.
- Se agrega el concepto de propietario de un recurso para restringir las operaciones.

### 4.1.5 Versión 5

- Primera versión compatible con direccionamiento MSP430 estándar.
- Se modifican las funciones para que puedan recibir valores nulos en lugar de punteros y así evitar parámetros cuando las funciones se usan para sincronización.
- Las funciones de librería tsleep y sleep se integran al kernel.
- Se agrega el control de desbordamiento del stack.

#### 4.1.5.1 Revisión 7

- Se optimiza el código cambiando los múltiples accesos a elementos indexados por punteros para minimizar el tamaño del código y el tiempo de ejecución.

#### 4.1.5.2 Revisión 10

- Se agrega la opción de utilizar múltiples copias del código.

### 4.1.6 Versión 6

- El stack del kernel pasa a estar sobre el stack del *"main"* para minimizar el consumo de memoria.
- Se agrega el estado RUNNING a los procesos para facilitar la depuración de programas.
- Desaparece el CRC del código en load y estructuras del kernel.
- Desaparece el largo del programa en load y estructuras del kernel.

- Se habilita el control de CRC de datos (previsto pero nunca utilizado).

#### 4.1.6.1 Revisión 12

- Se cambian las firmas para que los datos que no requieren ser modificados sean requeridos como constantes.
- Se elimina el estado de los procesos STOPPED

#### 4.1.6.2 Revisión 13

- La definición de los errores se mueven del archivo *"common.h"* a *"ukernel.h"*.
- Se hace opcional el control de estado del kernel.

#### 4.1.6.3 Revisión 14

- Se modifican las constantes utilizadas por kernel_init para permitir que la copias de código tengan distinto largo y CRC, es decir, para que puedan ser diferentes.

## 4.2 Diseño y construcción

Este kernel fue diseñado en a base conceptos propuestos por tres sistemas operativos de tiempo real:

- FreeRTOS [1]
- µCOS/II Kernel [2]
- SYS/BIOS [7]

La primera versión fue diseñada con la intención de sustituir una cola de tareas. La presente, para dar soporte a requerimientos de diferentes desarrollos sobre varias plataformas de hardware.

La implementación se realizó utilizando las herramientas de verificación de reglas de construcción para sistemas críticos de MISRA [10] y ULP (Ultra-Low Power) Advisor de Texas Instruments [11].

El proceso de verificación se realizó en etapas tempranas sobre un simulador, posteriormente sobre placas de desarrollo utilizando los procesadores a bajas frecuencias y por último en los diferentes ambientes de producción.

## 4.3 Archivos y funciones externas

El código del kernel se encuentra en un archivo de nombre *"ukernel.c"*.

Las definiciones de las funciones, valores de retorno y tipos asociados se encuentran en un archivo de nombre *"ukernel.h"*. Este último requiere un archivo de nombre *"ukernel_config.h"* que el programador debe proveer con la configuración.

De esta forma, incluyendo únicamente *"ukernel.h"* se puede tener acceso a las funciones del kernel y a los valores de configuración.

El kernel requiere de las funciones externas de las que se provee una implementación para el procesador MSP430F5438A en el archivo "MSP430F5438A_hal.c".

- memcpy2: para copiar una secuencia de bytes desde una dirección a otra.
- min_size: para obtener el mínimo de dos elementos de tipo size_t.
- low_power_mode: para pasar al procesador a modo de bajo consumo de energía.
- set_watchdog: para configurar el temprizador "watchdog".
- reset: para reiniciar el procesador.

Además, el kernel requiere un conjunto de funciones de bajo nivel para realizar las tareas específicas de inicialización y cambio de contexto. Estas funciones son:

- init_program_stack: para inicializar el stack de un programa previo a su ejecución.
- resume: para restaurar el contexto de un programa y entregarle el procesador.
- task_change: para salvar el estado del procesador y entregar el control al despachador.
- get_suspend_sp: para conocer el valor del stack pointer cuando el proceso se suspenda.
- cast_ptr: para convertir la dirección de memoria de un programa en un entero.

Estas funciones son estrictamente dependientes de la arquitectura y están escritas en el lenguaje Assembler del procesador.

Se proveen dos versiones, ambas para MSP430. Una para utilizar con el núcleo en modo estándar (direccionamiento de 16 bits) y una para utilizar en modo extendido (direccionamiento de 20 bits).

Estas implementaciones se encuentran respectivamente en los archivos *"msp430_ukernel_ll.s43"* y *"msp430X_ukernel_ll.s43"*.

El pasaje de parámetros entre las funciones escritas en C y las funciones escritas en Assembler se realiza de acuerdo a lo definido en la sección "Assembler language interface" del manual del compilador para MSP430 [9].

# Capítulo 5.

## Código de ejemplo

### 5.1 Elementos comunes

Los ejemplos presentados en esta sección están diseñados sólo para demostrar el uso de las funciones de µKernel. Cada uno de ellos requiere un archivo de configuración "ukernel_config.h". Para simplificar la cantidad de documentos, se ha construido una configuración común a todos.

Por otra parte, los ejemplos necesitan algunos servicios del hardware, como es el caso del temporizador del kernel. Los servicios requeridos por todos los ejemplos se han agrupado en una implementación para el procesador MSP430F5438A en el archivo "MSP430F5438A_hal.c" y su declaración en el archivo "MSP430F5438A_hal.h".

Todos los ejemplos han sido probados en un simulador y ejecutados sobre un procesador montado en una placa de desarrollo.

En la carpeta de nombre "iar" se incluye la definición de un espacio de trabajo para "IAR Embedded Workbench for MSP430 5.52" en el archivo "sample.eww" que contiene la definición de cuatro proyectos: "sample01", "sample02", "sample03" y "sample04" cuyos detalles se encuentran en los archivos con esos nombres y extensiones "dep" y "ewd". En esos proyectos se definen los detalles requeridos para compilar y ejecutar los ejemplos.

## 5.2 Ejemplo 1: Carga y ejecución de un proceso

Este ejemplo muestra un programa muy simple que ejecuta un único proceso y la forma más básica de utilizar las funciones load y run.

```
/**
     Universidad de la República - Facultad de Ingeniería
     Instituto de Computación - Instituto de Ingeniería Eléctrica

     @file      sample01.c
     @version   6.13
     @date      17/11/2013
     @author         Gustavo De Martino

     @brief          Código de ejemplo del uso de µKernel v 6.13
                     Carga y ejecución de un proceso
*/
#include <stdio.h>
#include "ukernel.h"
#define PROCESO_STACK_WORDS 64
#define WATCHDOG_CONFIG 4

/**  Este proceso escribe un texto en la terminal i/o y se bloquea
     permanentemente */
void proceso11 (void* parameters)
{
     printf("Hello world\n");
     stop();
}

int main( void )
{
     /* Los tamaños del stack están definidos en palabras */
     stack_t proceso11_stack [PROCESO_STACK_WORDS];

     /* Instala la interrupción del timer para el kernel */
     hardware_init();

     /* Inicializa el kernel */
     kernel_init ();

     /* Prepara el contexto del programa */
     load( proceso11, 0, proceso11_stack, PROCESO_STACK_WORDS,
          WATCHDOG_CONFIG);

     /* Entrega el control al despachador para que pase a modo de
ejecución */
     run();
     return 0;
}
```

## 5.3 Ejemplo 2: Sincronización y transferencia de datos

Este ejemplo muestra cómo dos procesos se sincronizan e intercambian datos utilizando las funciones lock, set_buffer, read, write y unlock.

```
/**
     Universidad de la República - Facultad de Ingeniería
     Instituto de Computación - Instituto de Ingeniería Eléctrica

     @file      sample02.c
     @version   6.13
     @date      17/11/2013
     @author          Gustavo De Martino

     @brief           Código de ejemplo del uso de µKernel v 6.13
                      Sincronización y transferencia de datos entre
procesos
*/

#include <stdio.h>
#include "ukernel.h"

#define PROCESO_STACK_WORDS 64
#define WATCHDOG_CONFIG 4

/* Espacio de transferencia fuera del stack si se utiliza el control
de
 consistencia de datos */
int proceso21_input_buffer;

void proceso21 (void* parameters)
{
     /* Bloquea indefinidamente a la espera por el recurso */
     lock( RESOURCE_1, 0);

     /* Define un espacio de transferencia del tamaño de un entero
*/
     set_buffer( RESOURCE_1, &proceso21_input_buffer,
                sizeof(proceso21_input_buffer));

     /* Bloquea indefinidamente a la espera de datos asociados al
recurso
     ignorando el largo de datos recibido */
     read (RESOURCE_1, 0, 0);
     printf("proceso21: dato recibido %d\n",
proceso21_input_buffer);
     unlock(RESOURCE_1);
     stop();
}

void proceso22 (void* parameters)
{
     int dato = 1234;
     /* Bloquea hasta transferir el dato ignorando el largo
transferido */
     write(RESOURCE_1, &dato, sizeof(dato), 0, 0);

     stop();
}


int main( void )
```

```
{
     /* Los tamaños del stack están definidos en palabras */
     stack_t proceso21_stack            [PROCESO_STACK_WORDS];
     stack_t proceso22_stack            [PROCESO_STACK_WORDS];

     /* Instala la interrupción del timer para el kernel */
     hardware_init();

     /* Inicializa el kernel */
     kernel_init ();

     /* Prepara el contexto de los programas */
     load(proceso21, 0, proceso21_stack, PROCESO_STACK_WORDS,
          WATCHDOG_CONFIG );
     load(proceso22, 0, proceso22_stack, PROCESO_STACK_WORDS,
          WATCHDOG_CONFIG );

     /* Entrega el control al despachador para que pase a modo de
ejecución */
     run();

     /* Esto nunca se ejecuta */
     return 0;
}
```

## 5.4 Ejemplo 3: Señalización desde una ISR

Este ejemplo muestra una de las formas de utilizar la función writei desde una ISR para señalizar a un proceso y el uso de la función read con tiempo límite.

```
/**
     Universidad de la República - Facultad de Ingeniería
     Instituto de Computación - Instituto de Ingeniería Eléctrica

     @file      sample03.c
     @version   6.13
     @date      17/11/2013
     @author         Gustavo De Martino

     @brief          Código de ejemplo del uso de µKernel v 6.13
                     Uso de un recurso como elemento de señalización
*/

#include <stdio.h>
#include "ukernel.h"

#define PROCESO_STACK_WORDS 64
#define WATCHDOG_CONFIG 4


/**
 *    ISR del ADC
 */
volatile unsigned int internal_temp;
#pragma vector = ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{
  switch(__even_in_range(ADC12IV,6))
  {
  case  0: break;
  case  2: break;
  case  4: break;
  case  6: /**< Vector  6:  ADC12IFG0 */
     internal_temp = ADC12MEM0;
     /* Señalización de conversión completa sin transferencia de
datos */
     writei(RESOURCE_1, 0, 0, 0);
     __low_power_mode_off_on_exit();
     break;
  }
}

/*   Este proceso configura el ADC para que lea la temperatura
     La ISR señaliza el final de la conversión usando el recurso
RESOURCE_2
     El valor de la temperatura actualizado cada 1 segundo
*/
float temp;

void proceso31 (void* parameters)
{
     error_t error;
     for (;;)
     {
```

```
          /* Bloquea indefinidamente a la espera por el recurso
RESOURCE_2
          utilizado por la ISR de ADC12 para indicar el fin de la
lectura */
          lock( RESOURCE_1, 0);

          /* Sin espacio de transferencia, sólo para señalizar */
          set_buffer( RESOURCE_1, 0, 0);

          /* Configuro el conversor A-D para leer la temperatura
del
               microprocesador */
          adc12_read_temp();

          /* Espero la señalización de la ISR con un tiempo máximo
          de 500 ms */
          error = read (RESOURCE_1, 0, 5);

          /* Apago el ADC */
          adc12_off();

          if (error == ERROR_NO_ERROR )
          {
               /* Uso la curva característica del dispositivo para
obtener
               el valor */
               unsigned int mesure = internal_temp;
               temp = (((float)(mesure + (mesure >>1)))-
2818.048)/10.32192;
          }
          else if (error == ERROR_TIMEOUT )
          {
               /* No se obtuvo la medida en el tiempo esperado */
               printf("proceso5: No se pudo leer la temperatura");
          }
          /* Libero el recurso asociado al ADC */
          unlock(RESOURCE_1);
          /* Espero aprox 1 segundo */
          tsleep(10);
     }
}

int main( void )
{
     stack_t proceso31_stack            [PROCESO_STACK_WORDS];
     hardware_init();
     kernel_init ();
     load( proceso31, 0, proceso31_stack, PROCESO_STACK_WORDS,
          WATCHDOG_CONFIG );
     run();
     return 0;
}
```

## 5.5 Ejemplo 4: Procesos paramétricos

Este ejemplo muestra cómo utilizar un mismo código para ejecutar varios procesos. En este caso se ejecutan cuatro instancias del programa 4.1. El programa 4.2 inicia una secuencia de transferencias de datos entre los procesos con el siguiente flujo de datos.

| Proceso | Lee datos desde | Escribe datos en | Intervalo entre ejecuciones. |
|---|---|---|---|
| 4.1 | **RESOURCE_3** | RESOURCE_1 | 100 ms |
| 4.2 | RESOURCE_2 | RESOURCE_3 | 200 ms |
| 4.3 | **RESOURCE_3** | **RESOURCE_2** | 300 ms |
| 4.4 | RESOURCE_1 | **RESOURCE_2** | 400 ms |

Los procesos 1 y 3 compiten para obtener datos del recurso 3 que son generados por el proceso 2. Los procesos 3 y 4 entregan datos al recurso 2

```
/**
     Universidad de la República - Facultad de Ingeniería
     Instituto de Computación - Instituto de Ingeniería Eléctrica

     @file      sample04.c
     @version   6.13
     @date      17/11/2013
     @author          Gustavo De Martino

     @brief           Código de ejemplo del uso de µKernel v 6.13
                      Varias instancias del mismo código.
                      Varios productores, varios consumidores.
*/
#include <stdio.h>
#include "ukernel.h"

#define PROCESO_STACK_WORDS 64
#define WATCHDOG_CONFIG 4

/* Estructura de datos parámetros de un proceso */
struct process_data{
     int             process_id;
     resource_t read_from;
     resource_t write_to;
     char       buffer[10];
     size_t          length;
};

/* Parámetros de los procesos */
struct process_data p1 = {1, RESOURCE_3, RESOURCE_1, "", 10};
struct process_data p2 = {2, RESOURCE_2, RESOURCE_3, "", 10};
struct process_data p3 = {3, RESOURCE_3, RESOURCE_2, "", 10};
struct process_data p4 = {4, RESOURCE_1, RESOURCE_2, "", 10};

/* Proceso del que se ejecutarán varias instancias */
void proceso41(void* parameters)
{
     struct process_data * par = (struct process_data *)
parameters;
     printf("Iniciando proceso %d \n",par->process_id);
```

```
     printf("Obteniendo datos de recurso %d y entregándolos a
recurso %d\n",
                par->read_from, par->write_to);

     for(;;)
     {
          lock( par->read_from, 0);
          set_buffer( par->read_from, par->buffer, par->length);
          read(par->read_from, &par->length, 0);
          par->buffer[8] = par->process_id + '0';
          printf(par->buffer);
          printf("\n");
          unlock(par->read_from);
          write(par->write_to, par->buffer, par->length, 0, 0);
          tsleep(par->process_id);
     }
}

/* Este proceso inicia la comunicación y se bloquea permanentemente
*/
void proceso42 (void* parameters)
{
     write(RESOURCE_1, "Proceso ?", 10, 0, 0);
     stop();
}

int main( void )
{
     /* Los tamaños del stack están definidos en palabras */
     stack_t proceso411_stack     [PROCESO_STACK_WORDS];
     stack_t proceso412_stack     [PROCESO_STACK_WORDS];
     stack_t proceso413_stack     [PROCESO_STACK_WORDS];
     stack_t proceso414_stack     [PROCESO_STACK_WORDS];
     stack_t proceso42_stack      [PROCESO_STACK_WORDS];

     hardware_init();

     kernel_init ();

     load( proceso41, &p1, proceso411_stack, PROCESO_STACK_WORDS,
          WATCHDOG_CONFIG );
     load( proceso41, &p2, proceso412_stack, PROCESO_STACK_WORDS,
          WATCHDOG_CONFIG );
     load( proceso41, &p3, proceso413_stack, PROCESO_STACK_WORDS,
          WATCHDOG_CONFIG );
     load( proceso41, &p4, proceso414_stack, PROCESO_STACK_WORDS,
          WATCHDOG_CONFIG );
     load( proceso42, 0, proceso42_stack, PROCESO_STACK_WORDS,
          WATCHDOG_CONFIG );

     run();
     return 0;
}
```

## 5.6 Archivo de configuración

Este archivo muestra una configuración básica compatible con todos los ejemplos.

```
/**
     Universidad de la República - Facultad de Ingeniería
     Instituto de Computación - Instituto de Ingeniería Eléctrica

     @file     ukernel_config.h
     @version  6.13
     @date     07/11/2013
     @author         Gustavo De Martino

     @brief          Configuración del Kernel para MCS
*/
#ifndef __UKERNEL_CONFIG_H__
#define __UKERNEL_CONFIG_H__

/** Procesador a usar */
#include "MSP430F5438A_hal.h"

#define CONTROL_STACK_OVERFLOW
#define CONTROL_STACK_CONSISTENCY
#define CONTROL_KERNEL_STATUS

/** Una sola copia del kernel */
// #define KERNEL_COPY

/* Cantidad de procesos */
#define MAX_PROGRAM_COUNT 5

/** Recursos */
#define RESOURCE_1 1
#define RESOURCE_2 2
#define RESOURCE_3 3

/*  Cantidad de recursos */
#define MAX_RESOURCE_COUNT 4

/* Configuración para el watchdog en LPM: Valor adecuado para
MSP430F5438A*/
#define LMP_WATCHDOG_CONFIG 4

#endif    /* ifdef __UKERNEL_CONFIG_H__ */
```

# Referencias


[1] Real Time Engineers “FreeRTOS operating system”, Disponible en “http://www.freertos.org”, Visitado en noviembre de 2013.

[2] Micrium, Inc. “µCOS/II Kernel”, Disponible en “http://micrium.com/rtos/ucosii/”, Visitado en noviembre de 2013.

[3] Labrosse, Jean J. (2002) “MicroC/OS-II The Real Time Kernel”, Second Edition.

[4] Texas Instruments Incorporated, “Real-time operating system SYS/BIOS online documentation”, Disponible en “http://processors.wiki.ti.com/index.php/Category: SYSBIOS”, Visitado en noviembre de 2013.

[5] Texas Instruments Incorporated (2013) “TI-RTOS 1.10 User's Guide” Revisión C.

[6] Texas Instruments Incorporated (2013) “TI-RTOS 1.10 Getting Started Guide” Revisión E.

[7] Texas Instruments Incorporated (2013) “TI SYS/BIOS v6.35 Real-time Operating System User's Guide” Revisión M

[8] Texas Instruments Incorporated (2012) “IAR Embedded Workbench - IDE Project Management and Building Guide” Cuarta edición , Part number: UIDEEW-5.

[9] Texas Instruments Incorporated (2011) “IAR C/C++ Compiler Reference Guide for Texas Instruments’ MSP430 Microcontroller Family”, Novena edición, Part number: C430-9

[10] Texas Instruments Incorporated (2009) “IAR Embedded Workbench® MISRA C:2004 Reference Guide”, Segunda edición, Part number: EWMISRAC:2004-2.

[11] Texas Instruments Incorporated “ULP Advisor” integrado a IAR Embedded Workbench for MSP430 5.52, Disponible en “http://www.ti.com/tool/ulpadvisor”, Visitado en noviembre de 2013.