Actualización parcial de software embebido en tiempo de ejecución en sistemas sin RTOS

Santiago Martínez, Matías Bakalián, Leonardo Steinfeld, Francisco Lanzari
Instituto de Ingeniería Eléctrica
Facultad de Ingeniería, UdelaR
Montevideo, Uruguay
santiagolmb@gmail.com, m.bakalian@gmail.com, leo@fing.edu.uy, flanzari@gmail.com

Resumen—**La actualización de software embebido permite alterar o modificar la funcionalidad de un sistema. En algunos casos puede llegar a ser un procedimiento engorroso si el equipo es de difícil acceso o con un alto costo operativo si se trata de gran cantidad de dispositivos. En este contexto, la opción de realizarlo de manera remota mediante comunicación inalámbrica resulta una alternativa muy atractiva. Además puede ser importante minimizar el volumen de información a transmitir para disminuir la energía de reprogramación. Siguiendo esta línea se han propuesto mecanismos para realizar una programación parcial. En el presente trabajo, a partir de la observación de que en muchos de estos sistemas el código correspondiente a la aplicación es sensiblemente menor en tamaño que las bibliotecas utilizadas, se propone un mecanismo para realizar la actualización de la aplicación manteniendo las bibiliotecas.**

Esta solución simple logra una implementación que reduce el costo de reprogramación, ocupa poca cantidad de memoria y tiene despreciable overhead de ejecución. Además sirve como primer acercamiento a alternativas de programación parcial más complejas.

Palabras Clave; reprogramación; comunicación inalámbrica ; software embebido

I. Introducción

En el contexto de los sistemas embebidos, con frecuencia es necesario y/o deseable actualizar cierta sección de código, por ejemplo para la corrección de bugs o modificación de algoritmos (por ejemplo: encriptación de datos, codificación o decodificación, etc). También puede desearse modificar solamente la aplicación manteniendo los servicios que bibliotecas existentes ya proveen. Debido a esto, sería deseable y de gran aplicabilidad poseer una herramienta para la actualización parcial de código sin necesidad de reprogramar completamente el sistema.

Se han propuesto e implementado varios mecanismos para modificar el código dentro de un sistema embebido. La técnica más adecuada a cada caso depende de las funcionalidades requeridas y también de las limitaciones en términos de recursos, ya sean de potencia de cómputo, memoria o incluso energía en casos de sistemas portátiles alimentados a pilas.

La primera opción, trivial, para actualizar el software es la reprogramación total del código, donde una nueva imagen binaria del software remplaza el programa antiguo, por ejemplo vía JTAG o BSL (Bootstrap Loader). Éste es el mecanismo más sencillo y ampliamente utilizado, normalmente no se realiza en campo y es llevado a cabo por un usuario utilizando un PC.

Cuando se cuenta con una cantidad importante de equipos instalados, el método anterior tiene un alto costo y es de difícil implementación logística. Entonces resulta atractivo realizar la reprogramación en campo, transmitiendo el nuevo código al equipo que necesita su actualización de manera inalámbrica. En diversas ocasiones, solamente es necesario realizar pequeñas modificaciones al código, como en la corrección de bugs, por lo que el código binario final sólo sufre pequeñas modificaciones. En estos casos es posible, en lugar de realizar un reemplazo total de la imagen del programa, cargar tan solo las diferencias entre el binario original y el modificado. Esta técnica está motivada por la necesidad de minimizar el costo energético de la reprogramación y tiene sentido en sistemas donde el costo asociado a la transmisión es varios órdenes de magnitud mayor al costo de procesamiento. De esta manera se minimiza el volumen de datos transferidos al sistema embebido a costas de un incremento en el procesado local para la obtención de la imagen combinada [1]. Otra alternativa es usar sistemas con máquinas virtuales, ya que por más que se programe totalmente el sistema, el código interpretado o bytecode generalmente es sensiblemente menor que el código directamente ejecutable[2].

Una alternativa a los mecanismos mencionados anteriormente es la utilización de módulos cargables en sistemas que cuentan con un kernel o sistemas operativos diseñados especialmente para contemplar esta funcionalidad. Algunos ejemplos son Contiki [3] y SOS [4], ambos de código abierto. Cuando un nuevo módulo es cargado, esto es, se incorpora a un sistema ya corriendo, contiene referencias a variables y funciones externas al propio módulo. Estas pueden ser resueltas en tiempo de compilación en el host, proceso llamado pre-enlazado, o en el propio sistema embebido en tiempo de ejecución, llamado enlazado dinámico. Los módulos dinámicamente enlazados contienen referencias sin resolver, mientras que en los módulos pre-enlazados éstas ya fueron sustituidas por direcciones físicas absolutas. En el primer caso, la utilización de tablas es una técnica comúnmente utilizada para la resolución de etiquetas, donde el proceso de enlazado es completado en el sistema embebido a través de tablas donde los

nombres simbólicos se traducen en direcciones absolutas. El sistema operativo SenSpireOS [5] contempla la funcionalidad de enlazado dinámico de módulos SELF (basados en el formato estándar ELF) mediante una tabla de direcciones, con la finalidad de realizar la reprogramación parcial de los nodos de una red de sensores inalámbricos. [6]

La utilización de kernels o sistemas operativos con estas funcionalidades normalmente trae aparejado un incremento significativo en la complejidad del software embebido, un incremento en el total de código y un overhead de tiempo de ejecución asociado a la búsqueda en tablas [1].

El presente trabajo plantea dar una solución simple al problema de la programación parcial remota en arquitecturas de software donde no se cuenta con un kernel con funcionalidades especiales donde se realiza la actualización de la aplicación manteniendo las bibliotecas. El objetivo es minimizar el costo de reprogramación a través de una implementación con baja cantidad de memoria y despreciable overhead, y que sirva como primer acercamiento a alternativas más complejas.

II. Propuesta

En base a las consideraciones realizadas, se propone el uso de módulos pre-enlazados para permitir, en tiempo de ejecución, la modificación parcial del software del sistema. El objetivo es actualizar la aplicación manteniendo las bibliotecas utilizadas por la misma. Para ello es necesario que el sistema a actualizar reciba a través de una interfaz de comunicación el código binario de la nueva aplicación, guardar el mismo en la memoria, típicamente no-volátil (FLASH), para finalmente comenzar a ejecutar la nueva versión. Para la generación de dicho código binario es necesario contar con herramientas adecuadas.

Es conveniente aclarar que debe tenerse en consideración el contenido existente en la memoria del sistema que se encuentra ya instalado en campo. Una opción sería diseñar un esquema, de cierta complejidad, que a partir de los archivos ejecutables del sistema existente y de la nueva aplicación, identifique y extraiga la porción de ejecutable a actualizar. Esto podría realizarse mediante la edición de los archivos ELF (Executable and Linkable Format) asociados. Sin embargo se optó por una solución más sencilla mediante la utilización de opciones avanzadas del *linker*. La idea es usar una tabla de direccionamiento indirecto, al igual que en soluciones anteriormente mencionadas, para permitir hacer uso de las bibliotecas sin necesidad de disponer de ellas al momento de compilar y enlazar la nueva aplicación. Para permitir la actualización de la aplicación sin sobrescribir las bibliotecas, es necesario organizar las memorias de manera de poder guardar en áreas diferentes aplicación y bibliotecas. Esta solución, a diferencia de la edición de los archivos ELF, requiere reprogramar la aplicación en forma completa. En principio puede considerarse un costo aceptable debido a la sencillez del método en comparación con otros, más si se considera que el tamaño de la aplicación, en general, es mucho menor que el tamaño de las bibliotecas.

A. *Tabla de direccionamiento indirecto*

Durante el desarrollo de una nueva aplicación basada en bibliotecas, ya sea para corregir bugs o cambiar completamente la funcionalidad, es necesario que el *linker* resuelva las llamadas desde la aplicación a funciones y los accesos a variables de dichas bibliotecas precargadas en el sistema.

Para realizar lo anterior se utilizó una tabla de direccionamiento indirecto, con el fin de realizar la interacción entre las bibliotecas y la aplicación.

Como paso inicial, compilan las bibliotecas junto con una aplicación de inicialización y con la tabla con el fin de cargar en la misma las direcciones donde se encuentran alojadas las funciones públicas pertenecientes a las bibliotecas. De este modo, la llamada a una función de las bibliotecas desde una nueva aplicación puede realizarse mediante una llamada a la dirección de memoria almacenada en el lugar correspondiente de la tabla, permitiendo el acceso a la función en forma indirecta.

En la figura 1 se muestra el mapa de memoria en el primer paso cuando las bibliotecas y la tabla han sido cargadas en memoria. Luego se muestra el mapa de memoria resultante después de la actualización, donde se puede apreciar el flujo de invocación para una función desde la aplicación hasta la biblioteca correspondiente.

En principio se podría asumir que es posible crear el nuevo software y simplemente enviar al sistema la porción de código relativa a la aplicación. Sin embargo, nada garantiza que el *locator* ubique las funciones de las bibliotecas en las mismas direcciones de memoria que lo hizo anteriormente y en consecuencia las llamadas desde la aplicación a las funciones se realizarían a una dirección de memoria diferente a donde estas se encuentran. Mediante el uso de la tabla, es decir, la nueva aplicación realizando las llamadas a la biblioteca a través de ésta, se evitan este tipo de inconvenientes ya que la misma contiene siempre las direcciones reales donde las funciones se encuentran alojadas.

Otro beneficio de la utilización de la tabla es que no es necesario disponer de las bibliotecas a la hora de generar la nueva aplicación.

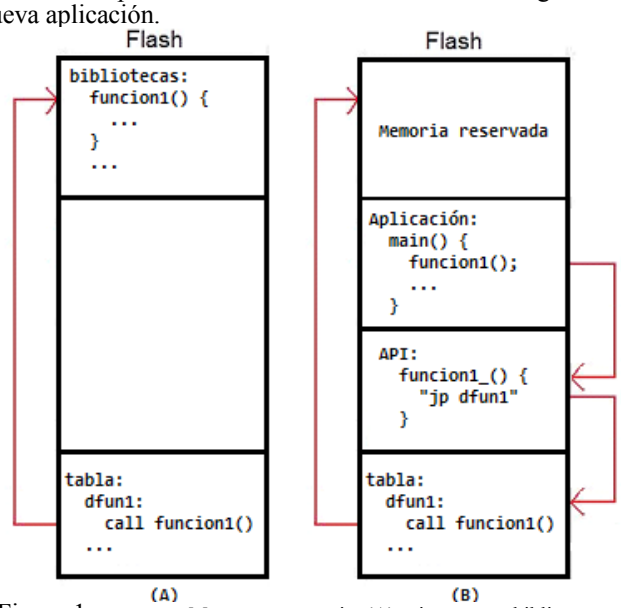


Figura 1. Mapeo en memoria: (A) primer paso: bibliotecas y tabla cargadas en memoria, (B): nueva aplicación y flujo de invocación para una función.

B. *Organización de las memorias*

Para que sea posible sustituir la aplicación y a la vez dejar inalteradas las bibliotecas existentes en la memoria del sistema, es necesario conservar código, constantes y variables de las funciones pre-cargadas. Por ello éstas últimas deben estar separadas en memoria del resto, para así tener la seguridad de que no sean sobrescritas. Para conseguirlo, una opción de organización es disponer de regiones disjuntas en las distintas memorias (FLASH y RAM) para las bibliotecas, la aplicación presente y la nueva aplicación. En este esquema se logra guardar completamente la nueva aplicación sin sobrescribir la aplicación actual antes de conmutarlas. También es posible sobrescribir la aplicación corriente con la nueva, ya que todas las rutinas necesarias para realizar la conmutación (recepción y escritura en memoria) residirían en la región destinadas a las bibliotecas. El código, las constantes y los valores de inicialización residen en FLASH mientras que las variables residen en RAM, por lo que la organización de la memoria debe contemplar ambas. En la Figura 2 se muestra un posible mapeo de memorias, en donde se distinguen los segmentos "Tabla" y "Aplicación" en FLASH y en RAM los segmentos "Aplicación", "Tabla" y "código para escritura en FLASH"

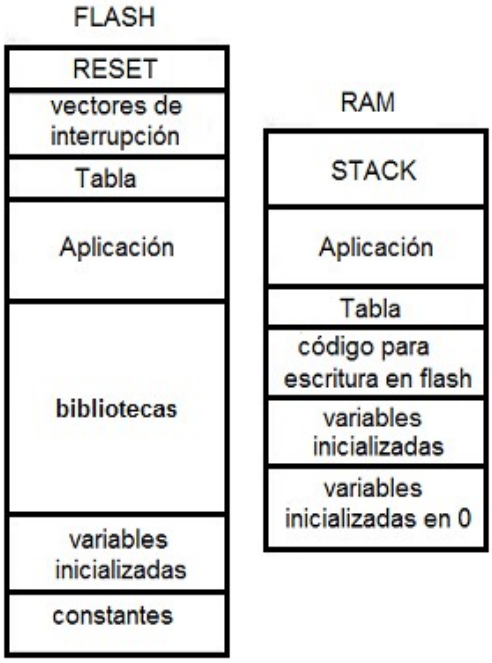


Figura 2. Mapa de memoria RAM y FLASH con los segmentos involucrados

C. *Recepción de datos y puesta en marcha de la nueva aplicación*

Tal como fuera dicho anteriormente, en el proceso de actualización, primero debe establecerse una comunicación entre el sistema embebido (mediante una interfaz de comunicación) y el sistema que posea el código de la nueva aplicación ya compilada. Este sistema puede ser, en principio, de muy diversos tipos, tanto un PC como otro sistema embebido.

Posteriormente, los datos correspondientes a la nueva aplicación son recibidos por el sistema embebido y almacenados en memoria volátil, para luego ser grabados en memoria persistente para finalmente poder ser ejecutados. El grabado en memoria persistente puede sobrescribir el código anterior, minimizando así la cantidad de memoria requerida, o puede ser alojado en un lugar diferente, pudiendo retornar a la aplicación anterior en caso de problemas tanto con el grabado como en el funcionamiento de la nueva aplicación.

Finalmente, antes de que la nueva aplicación pueda ejecutarse, es necesario llevar al sistema a un estado inicial, de modo de asegurar el buen funcionamiento de los servicios del sistema embebido y de las bibliotecas (configuración de periféricos, reinicialización de variables de las bibliotecas, inicialización de variables de la aplicación, etc.).

III. Estudio de caso

Para verificar la metodología propuesta se utilizó el kit de desarrollo ez430-RF2500 [7], compuesto por dos dispositivos que cuentan con un microcontrolador MSP430F2274 y un transmisor/receptor de radio CC2500, permitiendo establecer una comunicación punto a punto entre ellos. Para el desarrollo de las aplicaciones, se cuenta con bibliotecas provistas por el fabricante, en particular con un stack de comunicación, SimpliciTI [8], que proporciona un protocolo de comunicación simple que permite cumplir con el objetivo. Se utilizó el entorno de desarrollo IAR Embedded Workbench 6.0 [9], aunque también soporta el Code Composer Studio [10]. De las diferentes aplicaciones ejemplos provistas se eligió *"Simple peer to peer"*, en la cual se configuran dos dispositivos, uno para enviar y el otro para recibir mensajes a través de la radio. Inicialmente se estudió el código del stack para determinar las funciones públicas utilizadas por la aplicación, siendo éstas las que necesariamente deberán ser incluidas en la tabla de direccionamiento indirecto.

IV. Implementación

A continuación se describen algunas consideraciones importantes a la hora de la instrumentación de la propuesta, en especial, las relativas a la implementación particular a este caso de estudio y del conjunto de herramientas (compilador, *linker*, etc.) utilizado.

A. *Primeras consideraciones*

Al momento de guardar la nueva aplicación en memoria no volátil, puede ser necesario realizar un borrado total el segmento involucrado. Esto depende de las limitaciones que presente el microcontrolador utilizado a la hora de escribir en este tipo de memorias. Por ejemplo, en algunos microcontroladores se permite el borrado de bits pero no setearlos. En este caso, para poder guardar la nueva aplicación, es necesario inicializar el segmento guardando en todos sus bytes el valor 0xFF.

Hay casos en los cuales la rutina encargada de escribir en FLASH debe ejecutarse desde RAM. Para esto existen dos alternativas, guardar dicha rutina en RAM, o crear una copia en RAM justo antes de utilizarla. En ambos casos será necesaria una rutina auxiliar que se encargue de alojarla en RAM. La segunda opción es interesante cuando el sistema cuenta con muy poca RAM y se desea optimizar su utilización, pero genera overhead a la hora de requerirla (al actualizar). Se mencionaron dos alternativas a la hora de guardar la nueva aplicación en memoria. Para el caso en el que la nueva

aplicación sobrescribe a la anterior, es necesario que la rutina de actualización conserve el control del sistema hasta que la nueva aplicación esté completamente cargada en memoria y se haya llevado el sistema al estado inicial, para finalmente ceder el control a la nueva aplicación. El otro caso presenta más flexibilidad, en el sentido de que la actualización podría realizarse en varios pasos y la aplicación original podría seguir ejecutándose durante la misma, esto si se puede asegurar el correcto funcionamiento de la aplicación.

Una vez cargada la nueva aplicación en memoria, como se mencionó anteriormente, es necesario inicializar el sistema. La inicialización asociada a la configuración de periféricos, dispositivos conectados al microcontrolador (como por ejemplo la radio) que es realizada a través de llamadas a funciones, no presenta problemas. Sin embargo durante el start-up, antes de ejecutar la primera línea del "main", se asigna el valor inicial a las variables estáticas inicializadas (segmento data) y se resetean aquellas no inicializadas (segmento bss). Este paso puede realizarse de manera transparente para el usuario, agregando el código de start-up de la aplicación al realizar la actualización, y también el de las bibliotecas precargadas, o en forma manual, agregando explícitamente al inicio del código de aplicación el llamado a las funciones correspondientes para copiar los valores iniciales desde memoria no volátil a RAM para el caso de variables inicializadas y para el borrado, para las variables no inicializadas.

B. Tabla de direccionamiento indirecto

Una vez reconocidas las funciones del stack utilizadas por la aplicación, se generó la estructura de la tabla en un lugar adecuado de memoria y se almacenaron los punteros a las funciones. También fue necesario definir los tipos punteros a dichas funciones.

En el archivo de encabezado, *tabla.h,* se definen los tipos a utilizar, los punteros y la estructura de la tabla. En el archivo *tabla.c* se define la tabla, donde se especifica explícitamente su ubicación en memoria y la función de inicialización. En las Figuras 3 y 4 se muestra una sección de cada archivo, donde se puede apreciar el código relativo a dos funciones de las bibliotecas: *BSP_Init()* y *SMLP_Init(linkID_t)*.

```
typedef uint8_t (*funptr)(linkID_t);
typedef void (*ptr_BSP_Init)(void);
typedef smplStatus_t (*ptr_SMPL_Init)(funptr);
typedef struct dirfunctions{
   ptr_BSP_Init_ d1;
   ptr_SMPL_Init_ d2;
   // siguen más miembros
}dirfunctions;
```

Figura 3. Archivos encabezado tabla.h.

```
__no_init dirfunctions dfunctions @ 0xFFC0;

void table_init(void){
   dfunctions.d1 = BSP_Init;
   dfunctions.d2 = SMPL_Init;
   // siguen más miembros
}
```

Figura 4. Archivo tabla.c

De esta manera, se crea una tabla en la dirección 0xFFC0 y al ejecutarse la función *table_init()* quedan guardadas en la misma las direcciones donde se encuentran las funciones.

Para usar los servicios de las bibliotecas, la nueva aplicación es compilada utilizando un archivo de encabezado donde se declaran todas las funciones públicas de las bibliotecas y un archivo donde se definen dichas funciones de manera de acceder al código respectivo a través de la tabla, tal como se muestra en las Figuras 5 y 6.

```
void BSP_Init(void);
smplStatus_t SMPL_Init(funptr);
// siguen más prototipos
```

Figura 5. Archivos de encabezado con todas las funciones publicas de las bibliotecas.

```
__no_init ptr_BSP_Init* pBSP_Init @ "SEG_MAIN_RAM";

__no_init ptr_SMPL_Init* pSMPL_Init @ "SEG_MAIN_RAM";

void BSP_Init(void) @ "SEG_MAIN_FLASH" {
   pBSP_Init = (ptr_BSP_Init*) 0xFFC0;
   (*pBSP_Init)();
}

smplStatus_t SMPL_Init_(funptr f)@ "SEG_MAIN_FLASH" {
   pSMPL_Init = (ptr_SMPL_Init*) 0xFFC2;
   return (*pSMPL_Init)(f);
}
```

Figura 6. Archivo con las definiciones de las funciones.

Se puede observar el "cast" al tipo puntero correspondiente y la utilización de segmentos de memoria creados específicamente para la aplicación, con el fin de independizar en memoria el código perteneciente a la aplicación del perteneciente al stack de comunicación.

Para lograr ubicar los diferentes segmentos según la organización de memoria elegida se utilizan opciones específicas del *linker* [11]. Ello es posible modificando un archivo de configuración que el *linker* utiliza. En IAR, este puede encontrarse dentro de las propiedades del proyecto. El fragmento modificado del mismo se muestra en la Figura 7, donde se puede apreciar los rangos de direcciones asignadas a cada área.

```
// FLASH

-Z(CONST)DATA16_C,DATA16_ID,DIFUNCT, CHECKSUM=8000-80B5
-Z(CODE)CSTART,ISR_CODE=8086-80B5
-P(CODE)CODE=80B6-9BF5
-P(CODE)SEG_MAIN_FLASH=9BF6-BFFF

// RAM

-Z(DATA)DATA16_I,DATA16_Z,DATA16_N, DATA16_HEAP +
 _DATA16_HEAP_SIZE=0200-047F
-Z(DATA)CODE_I
-Z(DATA)CSTACK+_STACK_SIZE#
-Z(DATA)SEG_MAIN_RAM=04F0-05FF
```

Figura 7. Definición de segmentos.

Una vez creados los segmentos deseados desaparecen los problemas asociados a la sobrescritura de datos y se tiene campo fértil para intentar recibir una nueva aplicación. La recepción de la nueva aplicación se realiza a través de la UART del microcontrolador.

C. Escritura en FLASH de la nueva aplicación

Para escribir la nueva aplicación en memoria FLASH, debido al microcontrolador utilizado, fue necesario utilizar una

rutina que resida en RAM. Esta rutina de escritura, FlashWrite(), debe tener en cuenta consideraciones respecto a la cantidad de datos a escribir para no dañar la memoria. FlashWrite() es cargada en FLASH junto con las bibliotecas, por lo tanto es necesaria la existencia de otra rutina, CopyRoutine(), encargada de crear una copia de la misma en RAM para su ejecución. Las rutinas mencionadas anteriormente, específicas para el microcontrolador MSP430, forman parte de los códigos ejemplo suministrados por Texas Instruments [12]. Debido a la escasa memoria RAM que presenta el microcontrolador utilizado (1 kB), el hecho de tener que copiar FlashWrite() a RAM hace que el espacio disponible para recibir la nueva aplicación se reduzca a unos 150 bytes aproximadamente. Por lo tanto, en caso de que la misma supere dicho límite, deberá ser enviada de a fragmentos. En este caso, como fue comentado, puede resultar necesario pasarle el control a otra rutina, que espere hasta que se termine dicha transferencia para evitar problemas de datos corruptos.

V. Resultados

El sistema se fue probando y verificando con cada paso de la implementación. La metodología fue básicamente la misma para todos los casos, "debbugear" la aplicación, prestando atención al flujo del programa y los cambios producidos en la memoria y registros del microcontrolador.

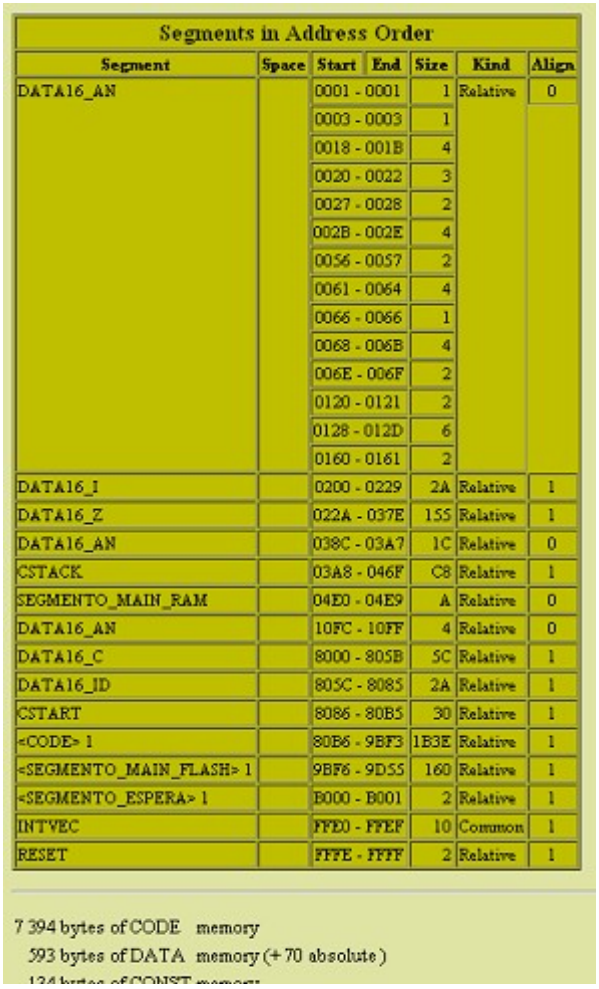
La primer prueba consistió en verificar en memoria que la tabla se generaba correctamente, es decir, que se encontrara en la ubicación correcta y se guardaran en ella las direcciones donde efectivamente se encontraban las funciones en cuestión. Para esto se tuvieron que superar varios inconvenientes, como la ubicación en memoria y la inicialización de estructuras "a posteriori", con el uso de directivas como "@" y "__no_init" respectivamente para este compilador.

Luego de esto se probó la técnica de direccionamiento indirecto propuesta. Para esto se cargó en memoria el código perteneciente al stack de comunicación y se generó la tabla, se configuraron opciones de la plataforma de desarrollo para que se mantenga la información en la memoria y se cargó la aplicación utilizando el direccionamiento indirecto. Es decir, se compiló únicamente la aplicación con los archivos de direccionamiento indirecto y se cargó en memoria. Ejecutando paso a paso la aplicación se verificó que las llamadas a las funciones efectivamente pasaban por la tabla, llamando correctamente a las funciones que debía, como era esperado.

La rutina de recepción de datos se probó enviando datos hacia la UART del sistema, utilizando para la comunicación el programa Hércules (freeware) [13]. Se verificó la llamada a la rutina de atención a la interrupción de la UART y la correcta recepción de los datos enviados, almacenados en RAM.

Respecto a la escritura en FLASH, se verificó el correcto funcionamiento de las funciones CopyRoutine() y FlashWrite() mencionadas anteriormente. Para el caso de CopyRoutine(), se observó la memoria ocupada por FlashWrite() y se comparó byte a byte con lo que se había copiado en RAM, verificando que no hubieran errores. Para FlashWrite(), se optó por mover a FLASH un arreglo conocido hacia una zona no inicializada de memoria y se constató su correcto funcionamiento.

Para la prueba del sistema completo se compiló y cargó el stack junto con una primera aplicación y la tabla en dos dispositivos, un receptor y un transmisor. La información correspondiente a la salida del *linker* se presenta en la figura 8. Luego se compiló una nueva aplicación junto con los archivos de acceso indirecto y se cargó en el transmisor, como se observa en la figura 9. Se constató el correcto funcionamiento del sistema con la nueva aplicación según los cambios de funcionalidad introducidos.

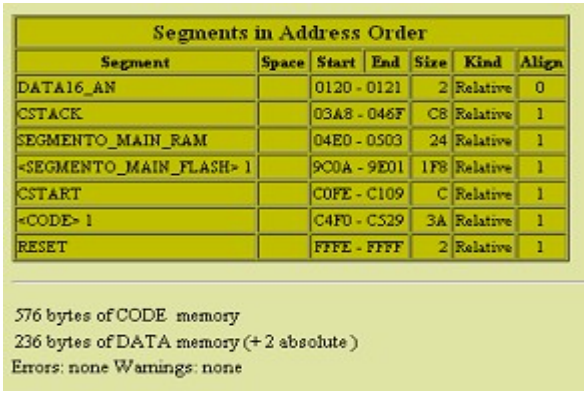


Segments in Address Order

Segment	Space	Start	End	Size	Kind	Align
DATA16_AN		0001	0001	1	Relative	0
		0003	0003	1		
		0018	001B	4		
		0020	0022	3		
		0027	0028	2		
		002B	002E	4		
		0056	0057	2		
		0061	0064	4		
		0066	0066	1		
		0068	006B	4		
		006E	006F	2		
		0120	0121	2		
		0128	012D	6		
		0160	0161	2		
DATA16_I		0200	0229	2A	Relative	1
DATA16_Z		022A	037E	155	Relative	1
DATA16_AN		038C	03A7	1C	Relative	0
CSTACK		03A8	046F	C8	Relative	1
SEGMENTO_MAIN_RAM		04E0	04E9	A	Relative	0
DATA16_AN		10FC	10FF	4	Relative	0
DATA16_C		8000	805B	5C	Relative	1
DATA16_ID		805C	8085	2A	Relative	1
CSTART		8086	80B5	30	Relative	1
<CODE> 1		80B6	9BF3	1B3E	Relative	1
<SEGMENTO_MAIN_FLASH> 1		9BF6	9D55	160	Relative	1
<SEGMENTO_ESPERA> 1		B000	B001	2	Relative	1
INTVEC		FFE0	FFEF	10	Common	1
RESET		FFFE	FFFF	2	Relative	1

7 394 bytes of CODE memory
593 bytes of DATA memory (+ 70 absolute)

Figura 8. Disposición de segmentos en memoria y tamaño de código, caso directo.



Segments in Address Order

Segment	Space	Start	End	Size	Kind	Align
DATA16_AN		0120	0121	2	Relative	0
CSTACK		03A8	046F	C8	Relative	1
SEGMENTO_MAIN_RAM		04E0	0503	24	Relative	1
<SEGMENTO_MAIN_FLASH> 1		9C0A	9E01	1F8	Relative	1
CSTART		C0FE	C109	C	Relative	1
<CODE> 1		C4F0	C529	3A	Relative	1
RESET		FFFE	FFFF	2	Relative	1

576 bytes of CODE memory
236 bytes of DATA memory (+ 2 absolute)
Errors: none Warnings: none

Figura 9. Disposición de segmentos en memoria y tamaño de código, caso indirecto.

Se observó que el código de la aplicación corresponde al 8% (576 bytes de código más 236 bytes de datos) del total de la aplicación incluyendo las bibliotecas (como se observa en el resumen de memoria de la Figura 9). En tiempo de ejecución, cuando se realiza una llamada a una función, se posee un overhead de ejecución de 2 instrucciones "jump" respecto al caso original. A continuación, en la Tabla 1, se presentan para diferentes aplicaciones ejemplo incluidas en el stack SimplicitTI: el tamaño total de la aplicación incluyendo las bibliotecas, la cantidad de bytes a transmitir correspondiente a la aplicación y finalmente el porcentaje que representa la aplicación respecto al total. Estas valores fueron obtenidos a partir del análisis del mapa de memoria al crear las distintas aplicaciones y sumando el espacio de memoria requerido por la tabla de direccionamiento indirecto. Puede observarse que la aplicación más el overhead asociado a la técnica representa entre el 3% y el 7% del total del software embebido de aplicaciones últimas analizadas, siendo la aplicación "Simple peer to peer", implementada y probada, la que resulta en el mayor peso relativo, 8%. Esto significaría un ahorro energético en costo de reprogramación de más de 90% en todos los casos en comparación con la reprogramación total.

Aplicación	Cantidad de bytes totales	Cantidad de bytes a retransmitir	Porcentaje de retransmisión
Polling whit Acces Point (Sender)	7548	544	7.2%
Polling whit Acces Point (Receiver)	7756	544	7.0%
Polling whit Acces Point (Acces Point)	6794	260	3.8%
Cascading end Devices	6924	399	5.8%
Acces Point as Data Hub(End Device)	8767	409	4.7%
Acces Point as Data Hub(Acces Point)	9885	705	6.1%
Acces Point as Data Hub(Chanel Sniffer)	6897	384	5.6%
Acces Point as Data Hub(Range Extender)	6854	216	3.2%

Tabla 1. Tamaño relativo de diferentes aplicaciones incluidas en SimpliciTI.

VI. Conclusiones

Se propuso una solución simple al problema de la reprogramación parcial que permite la actualización de la aplicación de manera remota utilizando comunicación inalámbrica. Se logró una implementación que ocupa poca cantidad de memoria y tiene despreciable overhead de ejecución.

Se diseñó e implementó un sistema basado en la metodología propuesta, pudiendo verificar la confiabilidad de la misma.

Se han contrastado los resultados obtenidos contra los del sistema sin direccionamiento indirecto obteniéndose para este caso particular un costo en memoria del 8% del total de espacio ocupado por el código original. Es decir que para realizar la actualización sólo es necesario transmitir el 8% del código que se debería transmitir si se cargaran nuevamente las bibliotecas junto con la aplicación.

Se ha estimado para otras implementaciones un costo de retransmisión que oscila entre el 3% y 7% del código total embebido.

Referencias

[1] Dunkels, A., Finne, N., Eriksson, J., and Voigt, T. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In Proceedings of the 4th international Conference on Embedded Networked Sensor Systems (Boulder, Colorado, USA, October 31 - November 03, 2006). SenSys '06. ACM, New York, NY, 15-28.

[2] Nanthanavoot P. and Chongstitvatana, P., "Code-Size Reduction for Embedded Systems using Bytecode Translation Unit", Conf. of Electrical/Electronics, Computer, Telecommunications, and Information Technology (ECTI), Thailand, 13-14 May 2004.

[3] A. Dunkels, B. Grönvall, and T. Voigt. Contiki—a Lightweight and Flexible Operating System for Tiny Networked Sensors. In EmNets, Tampa, Florida, USA, November 2004.

[4] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In ACM MobiSys, Seattle, Washington, USA, June 2005.

[5] W. Dong, C Chen, X Liu, J Bu, Y Liu, K Zheng. "SenSpire OS: A Predictable, Flexible, and Efficient OS for Wireless Sensor Networks.", 2007.

[6] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Liu, "Dynamic Linking and Loading in Networked Embedded Systems." in *Proceedings of IEEE MASS*, 2009

[7] Texas Instruments, MSP430 Wireless Development Tool, EZ430-RF2500, disponible: http://focus.ti.com/docs/toolsw/folders/print/ez430-rf2500.html; consulta: octubre de 2010.

[8] Texas Instruments; SimpliciTI Compliant Protocol Stack, disponible http://focus.ti.com/docs/toolsw/folders/print/simpliciti.html; consulta: octubre de 2010.

[9] IAR Systems, IAR Embedded Workbench 6.0; disponible: http://www.iar.com/; consulta: octubre de 2010.

[10] Texas Instruments, Code Composer Studio (CCStudio) Integrated Development Environment (IDE) v4.x; disponible: http://focus.ti.com/docs/toolsw/folders/print/ccstudio.html; consulta: octubre de 2010.

[11] IAR Systems, IAR Linker and Library Reference Guide. disponible: http://ftp.iar.se/WWWfiles/guides/xlink.pdf; consulta: octubre de 2010.

[12] Texas Instruments, MSP430 16-bit Microcontroller Code Examples and Function Library; disponible: http://www.ti.com/lit/zip/slac123; consulta: octubre de 2010.

[13] HW Group, Hercules Utilities; disponible: http://www.hw-group.com/products/hercules/index_es.html; consulta: ocutubre de 2010.