Universidad de la República
Facultad de Ingeniería

Proyecto de fin de carrera en
Ingeniería Eléctrica

SATELITEST

Test de inyección de fallas en satélite ANTEL-SAT

José Basualdo, Martín Vázquez, Fernando Viera

Tutor: Julio Pérez Acle

Montevideo, Uruguay
Junio de 2014

RESUMEN

En este documento se presenta el proyecto SATELITEST, que consiste en el diseño y la implementación de un mecanismo de inyección de fallas capaz de evaluar la robustez de un sistema embebido dado. Concretamente nos enfocamos en el satélite ANTEL-SAT, el cual sigue el estándar CUBESAT y fue desarrollado en virtud de un convenio entre ANTEL y la Universidad de la República.

El satélite consiste en una serie de módulos interconectados y basa su funcionamiento en el microcontrolador MSP430. Nuestro trabajo se centró en algunos subsistemas considerados críticos y en particular se tomó como objeto de estudio el kernel, considerando fallas que pudieran afectar la memoria. En resumen, el objetivo es evaluar la robustez del kernel del sistema ANTEL-SAT frente a fallas que puedan ocurrir en memoria, evaluando el impacto de las mismas en su funcionamiento.

Fue necesario en primer lugar estudiar el espacio de posibles fallas e identificar las más significativas para luego pasar al desarrollo del mecanismo de inyección. Posteriormente se introdujeron fallas en un prototipo dado del sistema bajo estudio, procediendo luego a relevar la respuesta frente a las perturbaciones a fin de evaluar la sensibilidad ante las mismas. Para ello se diseñó un método que permitiera automatizar el proceso y almacenar los resultados. Se lograron identificar algunos modos de mal funcionamiento que presenta el sistema, determinando así en qué grado lo afecta o no cada una de las fallas inyectadas.

AGRADECIMIENTOS

En primer lugar queremos agradecer a Julio Pérez por otorgarnos la posibilidad de realizar este proyecto, por la excelente disposición mostrada en todo momento y sus constantes correcciones y aportes a lo largo de este tiempo.

A Gustavo de Martino por atender reiteradamente nuestras inquietudes relativas al funcionamiento del kernel del sistema ANTEL-SAT y proporcionarnos sucesivas versiones del código.

A Andrés Touya y Matías Tassano por su tiempo para interiorizarnos acerca de la estructura del satélite, sus módulos y los periféricos del integrado MSP430 que emplea.

A Olivier Girard, desarrollador del openMSP430, por compartir su código en una plataforma pública y especialmente por responder nuestras consultas relativas al mismo.

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

Marco General

El ANTEL-SAT ha sido desarrollado en virtud de un convenio entre ANTEL y la Universidad de la República y será el primer satélite uruguayo en órbita. En su diseño e implantación han trabajado conjuntamente un equipo de la Facultad de Ingeniería (FING) constituido por unas 20 personas de forma permanente, y otro de ANTEL de 8 integrantes. Desde 2007 son aproximadamente 60 las personas han pasado por este proceso de desarrollo, en su gran mayoría estudiantes de la FING [1].

Tras una serie de tests en Uruguay, el satélite viajó a Estados Unidos para ser sometido a las últimas pruebas en la Universidad Politécnica de California. Luego será trasladado a Roma para incorporarse al UNISAT-6 de la Universidad de La Sapienza y finalmente llegará a Yasny, Rusia. Allí se integrará al cohete DNERP-1 para aguardar hasta su lanzamiento, previsto para mediados de junio [2].

La radiación y sus efectos

Un elemento importante a tener en cuenta a la hora de diseñar un satélite es el riesgo al que estará expuesto producto del medio en el que se encontrará. Entre los factores que pueden atentar contra su correcto funcionamiento se encuentran la posibilidad de colisionar con otros cuerpos, los efectos de la temperatura y la exposición a la radiación. Esta última puede conllevar disminución del tiempo de vida, descenso de performance o incluso la pérdida total de funcionalidades. Las fuentes de radiación pueden dividirse básicamente en anillos de partículas, viento solar, llamas solares y rayos cósmicos [3].

La radiación se compone básicamente de electrones, protones e iones pesados, siendo estos últimos los principales causantes de anomalías en el comportamiento de un satélite. Cuando una de estas partículas atraviesa un material va liberando energía a su paso. En particular, cuando un ión cruza la zona activa de un dispositivo electrónico deja rastros de carga a lo largo de su trayectoria, quedando la misma atrapada en el campo eléctrico presente. La corriente asociada puede inducir ciertos fenómenos conocidos en general como SEEs (single event effects) [4].

Un SEE es una alteración temporal en el nivel de un valor lógico y puede ser del tipo single event upset (SEU) o single event transient (SET). Un SEU puede ocurrir en cualquier FLIP FLOP en cualquier período de reloj y dura cierta cantidad de ciclos. El SET es algo similar pero se da de forma transitoria, incorporando la variable temporal en la caracterización de la anomalía. En este proyecto se ha descartado el estudio de SETs, centrándonos en los eventos de tipo SEU, así como también en fallas permanentes, a raíz de las cuales el valor de un bit queda fijo en 0 o en 1.

Motivación

El interés del proyecto radica mayormente en la imposibilidad de realizar mantenimiento correctivo una vez que el satélite es puesto en órbita. Por ello es de suma importancia poder determinar su respuesta frente a cierto tipo de fallas que eventualmente puedan alterar su funcionamiento. Esto es útil tanto en la etapa de diseño, para implementar mecanismos de tolerancia y robustecer el dispositivo, o incluso luego del lanzamiento, para estar prevenido frente a lo que pueda ocurrir.

Asimismo, más allá del aporte que se logre hacer al proyecto ANTEL-SAT, el trabajo también reviste interés per se al propiciar la familiarización con las técnicas usuales empleadas en este campo y motivar el desarrollo de mecanismos de inyección de fallas, registro de resultados y evaluación de los mismos.

Objetivo

El proyecto consiste en el diseño y la implementación de un mecanismo de inyección de fallas capaz de evaluar la robustez del satélite ANTEL-SAT. Este último se compone de una serie de módulos interconectados y basa su funcionamiento en el microcontrolador MSP430. El análisis se centra en algunos subsistemas considerados críticos y en particular se toma como objeto de estudio el kernel. Se consideran fallas de tipo SEU y permanente –a 0 o a 1- que puedan afectar la memoria, tanto de datos como de programa.

En resumen, el objetivo del proyecto contribuir a mejorar el diseño del kernel del sistema ANTEL-SAT, específicamente en cuanto a sus mecanismos de detección y tolerancia a fallas. Se considerarán fallas de tipo SEU o permanente que puedan ocurrir en memoria, evaluando el impacto de las mismas en el funcionamiento.

Visión global del trabajo

Para posibilitar una mejor comprensión del proyecto y visualizar globalmente el trabajo realizado, se presentarán las diferentes aristas del mismo comentando la vinculación entre ellas.

Hardware

Para obtener el hardware en el que corre el módulo µkernel a evaluar se grabó en un FPGA un core en lenguaje Verilog que emula a la familia 1 de microprocesadores MSP430, llamado openMSP430. Esto permitió obtener una implementación básica de la familia 1 de estos integrados, a la cual luego se le agregaron algunos periféricos para incorporar características de las familias 5 y 6 usadas en ANTEL-SAT, además de posibilitar la inyección de fallas.

El periférico más destacado es el módulo de inyección de fallas. El mismo toma como punto de partida una base de periférico diseñada por Olivier Girard. El diseño incluye algunos puertos estándar de entrada y salida y provee un banco de cuatro registros para darle instrucciones al módulo. Con el fin de obtener la funcionalidad relativa a la inyección de fallas, se incorporó un bloque de lógica que permite el enmascaramiento de los bits de interés.

Dada la necesidad de capturar los instantes en que ocurren algunos eventos significativos se implementó el “timer A1”. El mismo posee capacidad para interrumpir al micro, permitiendo finalizar el experimento mediante un timeout. Para su implementación se optó por “clonar” el timer original del openMSP430 pero otorgándole inmunidad ante resets del sistema.

En busca de un mejor manejo y comprensión de los estados por los que pasa el microcontrolador se incorporó un banco de registros de 8 bits, donde se lleva la cuenta de los distintos resets sufridos por el sistema. Se usó el mismo diseño de base comentado anteriormente.

Finalmente, se incorporó el módulo CRC, encargado de implementar el algoritmo de comprobación de redundancia cíclica, la cual no está prevista en la familia 1 del MSP430. Se utilizó nuevamente el template de Olivier Girard, al cual se le incorporó un bloque de cálculo de CRC obtenido a partir de un generador de código.

Síntesis de hardware y configuración de FPGA

Ante cualquier modificación que se realice al core, tanto para agregar nuevos periféricos como para ampliar la memoria del programa, es necesario sintetizar nuevamente el código, en nuestro caso mediante la herramienta Quartus II 13.0.0. Luego de compilar se obtiene la versión que se grabará en la placa, utilizando el programador lógico de Quartus.

El prototipo emulado mediante el FPGA es configurable a través de modificaciones en las definiciones de constantes en el archivo “defines.v”, donde es posible definir tamaño de memoria de programa y datos, habilitación de relojes, etc.

Software

Kernel

El modulo original µkernel utilizado está diseñado para las familias 5 y 6 de microprocesadores MSP430, siendo necesaria entonces su adaptación para funcionar en el core que emula el funcionamiento de la familia 1 de dichos integrados.

El módulo fue originalmente diseñado para correr en arquitecturas de 20 bits, por lo que se lo adaptó para funcionar en 16 bits. Fue necesario también modificar el código del µkernel para migrar a las herramientas incluidas en el proyecto del openMSP430: debuggers GDB y Minidebugger, compilador GCC.

Una vez adaptado, el módulo fue testeado a través de aplicaciones de prueba. En primera instancia se empleó una única aplicación sencilla que incrementa un contador hasta alcanzar un valor final preestablecido y luego usaron dos aplicaciones corriendo en simultáneo sobre el kernel y comunicándose entre sí.

Para debuggear y correr el programa es necesario compilarlo mediante GCC. Luego de haber grabado en la placa el core desde Quartus II se procede a cargar el programa en C, en este caso el µkernel y la aplicación, en la memoria del micro emulado. Para ello debe previamente generarse el archivo de extensión “elf”, ejecutando el comando “make” (lo cual llama al

compilador GCC) desde consola de comandos. Una vez obtenido el “elf”, el mismo es usado para abrir el código desde el minidebugger o GDB a través de un ”load”.

Gestor de campañas de inyección de fallas

Para automatizar el mecanismo de inyección de fallas y registro de resultados implicado en las campañas se escribió un script para correr desde GDB. Se trata principalmente de un loop en el cual se van realizando las sucesivas corridas para cada set de valores con los que se configurará el módulo de inyección de fallas. En este código se definen los vectores que se escribirán en los registros de control, se realiza la conexión con GDB proxy y se setean breakpoints en lugares de interés para luego poder determinar el flujo que siguió el programa.

Una corrida correspondiente a una inyección de falla individual puede terminar debido a la finalización exitosa de la aplicación que corre en el µkernel o por timeout del TimerA1. Luego de finalizar por cualquiera de estos motivos se evalúa el camino que tomó el programa en base a la cantidad de veces que el mismo pasa por ciertos puntos de interés. Lo que interesa en cada paso de la iteración es saber por ejemplo si la ejecución terminó en el tiempo esperado y con el resultado correcto, si se produjo algún reset y por cuál mecanismo. El script arroja esta información en un archivo de texto, que posteriormente es usado para describir la campaña en base al modelo FARM, un estándar para caracterizar la inyección de fallas.

En algunos casos interesa estudiar el efecto de alguna falla en particular y para ello lo que se hace es debuggear la misma individualmente desde GDB o Minidebugger. Se utiliza el archivo de extensión “.lst” generado, que permite observar en detalle el código del programa generado en assembler junto a su significado en C simultáneamente.

A modo de resumen, en la Figura 1 se muestran los diferentes elementos involucrados en el proyecto, mostrando cómo la vinculación entre los mismos deriva en el producto final.

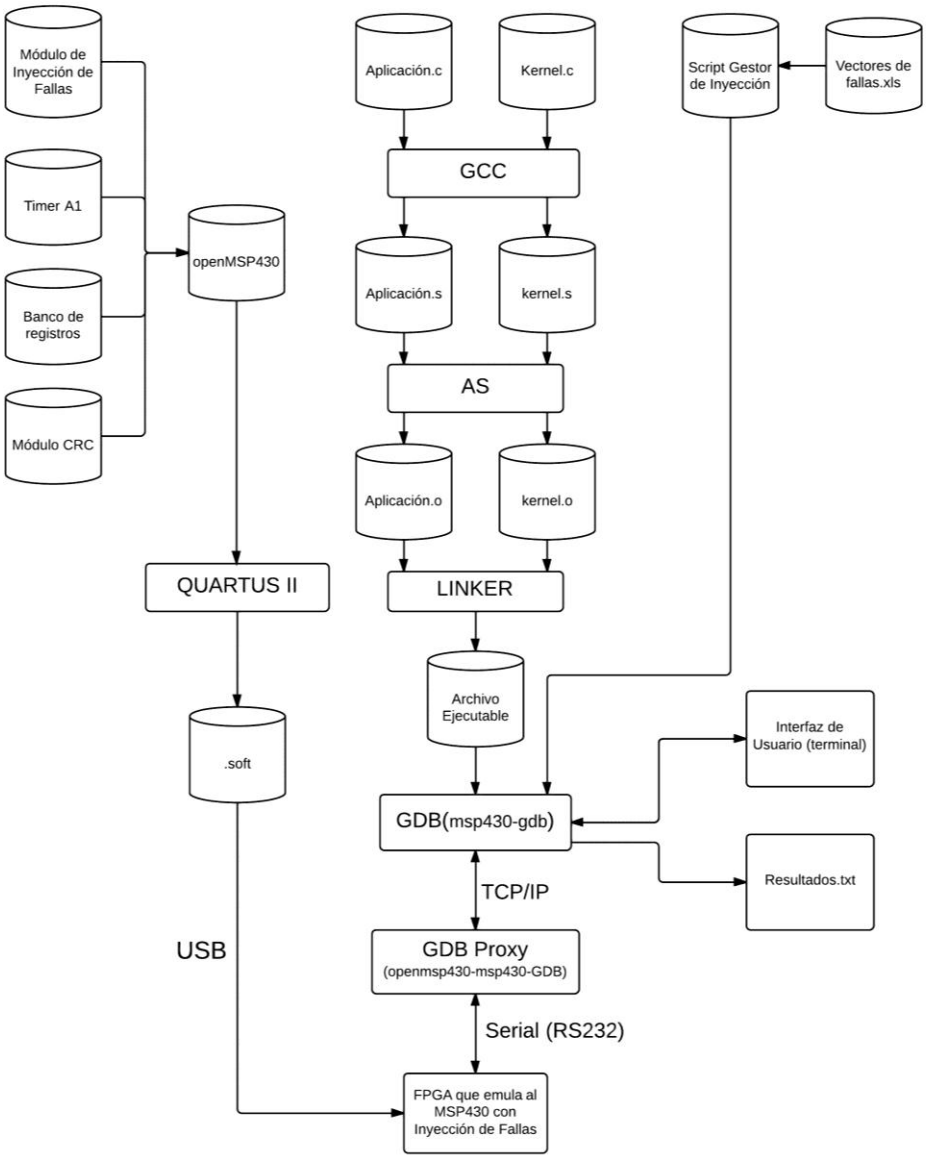


Figura 1: Vinculación entre los elementos involucrados en el proyecto

Estructura del documento

En esta sección se describe cómo están distribuidos los contenidos del documento, mostrando la estructura de los capítulos y sus apartados para introducir al lector en el proyecto y dar una idea de lo que encontrará en los textos venideros.

A lo largo de la Introducción se ha esbozado el contexto del proyecto y su motivación, presentando además los objetivos y la forma de encarar el problema planteado. Se pasa ahora a detallar el resto de los capítulos en los que se divide el documento.

2. ANTEL-SAT y MSP430: Se describen brevemente los módulos que componen el satélite ANTEL-SAT así como los periféricos del integrado MSP430. Se detallan luego cuáles de estos últimos son empleados por el primero, aclarando además en qué grado están implementados en el core openMSP430 que se tomará como base del trabajo.

3. KERNEL DEL SISTEMA ANTEL-SAT: Aquí se comenta el funcionamiento general del kernel sobre el cual corren las aplicaciones que ejecuta el satélite, explicando sus principales funciones, en especial el despachador de procesos o "scheduler". Luego se describe el proceso de adaptación del código para lograr compilar y debuggear con las herramientas de GCC y finalmente se comentan un par de aplicaciones diseñadas para probar el sistema.

4. INYECCIÓN DE FALLAS: Se introducen algunos conceptos teóricos relativos a la inyección de fallas y se comentan algunas técnicas basadas en diferentes métodos como simulación o emulación. Se presenta el modelo FARM.

5. PERIFÉRICOS INCORPORADOS: Se definen y especifican los periféricos que fueron incorporados al openMSP430 para implementar el mecanismo de inyección y registro de fallas así como cálculo de CRC.

6. CAMPAÑAS DE INYECCIÓN DE FALLAS: En esta sección se explica lo relativo a la preparación de las campañas, la automatización de las sucesivas corridas, las diferentes configuraciones del módulo de inyección de fallas y el mecanismo para registro de resultados. Para la segunda campaña se agrega el código necesario para incluir detección de overflow e inconsistencia de stack y triplicado de código.

7. RESULTADOS Y ANÁLISIS: Se despliegan los resultados de las dos campañas de inyección, analizando los casos de mayor interés, en los cuales las fallas alteran de alguna forma el flujo normal del programa. Se agrega una prueba particular para verificar el chequeo de CRC a las múltiples copias de código.

8. CONCLUSIONES: Se presenta un balance global del proyecto en base al análisis de los resultados obtenidos y a la experiencia adquirida en este período. Se plantean posibles mejoras a futuro.

ANEXOS: Se agrega información complementaria para la comprensión del trabajo. Se adjuntan descripciones de algunas herramientas utilizadas y procedimientos empleados en algunas etapas del proyecto.

2. ANTEL-SAT Y MSP430

Estructura del ANTEL-SAT

Introducción

El ANTEL-SAT es un satélite que sigue el estándar CUBESAT, es decir, de tipo miniatura y formados por uno o más cubos de 10 cm de arista, tratándose de dos en este caso. Las especificaciones del CUBESAT fueron establecidas en 1999 por las Universidades de Stanford y California, en procura de simplificar la estructura, bajar costos y reducir trámites burocráticos a la hora del lanzamiento [6].

El sistema puede dividirse en dos unidades: la aviónica y el payload. La primera está constituida por las partes necesarias para que el satélite funcione, tales como paneles solares, baterías, sistema de gestión de energía, control de actitud (orientación) y módulos de comunicación. La segunda consta básicamente de dos cámaras (una a color y otra para infrarrojos) y el sistema para el control de las mismas.

Módulos

El sistema ANTEL-SAT consta de varios módulos interconectados por un bus de comunicación I2C[1]. Su estructura se ilustra en el siguiente diagrama.

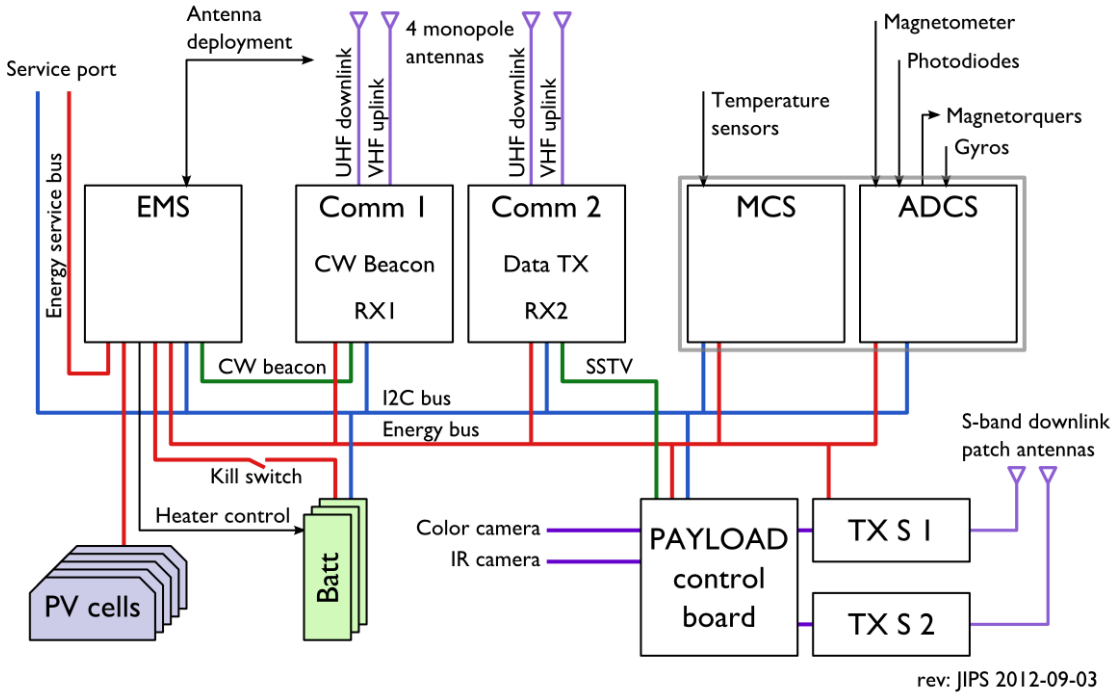


Figura 2: Estructura del ANTEL-SAT [2]

[1] I2C (Inter-Integrated Circuit) es un bus serial de comunicación.
[2] Figura extraída del Manual de usuario µKernel Versión 6.14, Gustavo de Martino, 2013.

CW beacon**:** Generación directa de baliza durante el modo seguro.

I2C bus: Bus de comunicaciones serial que utiliza 2 líneas (reloj y datos) para transmitir información.

SSTV**:** Método para transmitir y recibir imágenes a través de enlaces de radio.

UHF**:** Transmisor en la banda amateur.
VHF**:** Receptor en banda amateur.

COMM1**:** Módulo que trasmite baliza a solicitud del módulo de gestión de energía.
COMM2**:** Módulo que trasmite paquete de datos. Construye tramas AX25[3] para trasmitir los datos recibidos a través del bus I2C.

ADCS**:** Control de actitud (alinear el satélite respecto al campo magnético de la tierra). Utilizando las medidas de los sensores de luz ubicados en cada cara, se determina la posición respecto del sol, mientras que con un magnetómetro se determina el campo magnético actual. A partir de la hora UTC y conociendo la trayectoria, se determina la ubicación teórica del satélite. En base a eso y al modelo del campo magnético terrestre se determina la trayectoria esperada del campo magnético. Luego, el módulo de detección y control de actitud acciona las bobinas de torque para rotar el satélite a orientación deseada.

PAYLOAD**:** Carga científica. Aloja los sensores y sistemas de procesamiento y descarga de imágenes.

EMS**:** Módulo de gestión de energía.

PV cells**:** Paneles solares.

MCS**:** Esta pieza es el nodo central de comunicaciones y almacenamiento de datos del satélite. Está constituido por un microprocesador MSP430F5438A fabricado por Texas Instruments y algunas piezas de hardware adicional. El módulo se integra a la placa del ADCS.

Elementos a tener en cuenta

- Ya sea por eventos de radiación o falta de energía, el módulo de control principal puede reiniciarse o directamente apagarse.
- El único medio de interconexión entre los módulos del sistema es el bus I2C. Un bloqueo del mismo deriva en la paralización total del satélite.
- La escasa potencia de transmisión del satélite, las variaciones en la frecuencia derivadas de los cambios de temperatura y la gran atenuación debido a la distancia, hacen que la comunicación sea muy delicada.
- Deben implementarse los mecanismos para garantizar la persistencia de datos.
- Si bien el satélite puede mantener una operación básica sin el módulo de control principal, la mayor parte del dispositivo queda inutilizado si deja de funcionar**.**
- El problema más crítico es la falta de confianza en los dispositivos de memoria. Se deben tomar ciertas precauciones frente al riesgo de mutación de código, datos o

[3] Protocolo de comunicación derivado del X.25 usado por radioaficionados.

estado de dispositivos. Una de ellas es el uso de un periférico para detección de errores mediante cálculo de CRC16.[4]

- Las rutinas de atención a interrupciones realizarán el menor proceso posible y no tendrán bucles.
- Con respecto al kernel del sistema, es necesario proveer mecanismos que permitan retornar a una situación segura luego de una falla de hardware o software. El sistema debe ser capaz de seguir operando y en lo posible recuperarse ante mutaciones del código y/o los datos.
- El bus I2C funciona en modo multimaestro.
- La capa de enlace AX25 implementa la capa de red.

Periféricos del MSP430

MSP430 es una familia de microcontroladores fabricados por Texas Instruments para aplicaciones de bajo costo y poco consumo de energía. En el ANTEL-SAT se usan chips de las familias MSP430x5xx y MSP430x6xx. A continuación se comentan brevemente algunos de los periféricos presentes en esta variedad de integrados [7] que resultan de utilidad para el proyecto ANTEL-SAT. Una descripción más exhaustiva se incluye en el *Anexo 1: Periféricos del MSP430*.

Interfaz USCI

USCI es la interfaz de comunicación serial universal, la cual permite manejar múltiples modos de comunicación a partir de un único módulo de hardware. El MSP430 incluye 2 módulos diferentes de USCI: USCI_A y USCI_B. Los modos soportados por cada uno son los siguientes:

USCI_A:
- Modo UART[5]
- Formas de pulso para comunicaciones IrDA[6]
- Detección automática de ratios para comunicaciones LIN[7]
- Modo SPI

USCI_B:
- Modo I2C
- Modo SPI

SPI

SPI (serial peripheral interface) es uno de los modos de comunicación serial que soporta la interfaz USCI (interfaz de comunicación serial universal). Los datos son transmitidos y recibidos por múltiples dispositivos usando un reloj compartido que provee el dispositivo maestro.

[4] La comprobación de redundancia cíclica (CRC) es un código de detección de errores usado para detectar alteraciones en datos.
[5] UART es el transmisor/receptor asíncrono universal, el cual traduce datos entre los protocolos paralelo y serial.
[6] IrDA es un estándar de transmisión y recepción de datos por rayos infrarrojos.
[7] LIN es un protocolo de red serial.

El modo SPI incluye:

- Largo de palabra de 7 u 8 bits
- Transmisión y recepción de datos LSB-first o MSB-first.
- Modo de operación 3 o 4 pines
- Modo Maestro o Esclavo
- Shift registers para transmisión y recepción
- Registros buffer de recepción y transmisión
- Operación de transmisión y recepción continua
- Selección de polaridad de reloj y control de fase
- Frecuencia de reloj programable en modo maestro
- Capacidad de interrupción independiente para recepción y transmisión
- Modo esclavo en LPM4[8]

I2C

En modo I2C, el módulo USCI provee una interfaz hacia los periféricos conectados mediante el bus serie I2C. Soporta cualquier dispositivo maestro/esclavo compatible con I2C, pudiendo cada uno de ellos operar como transmisor o receptor. La comunicación se realiza a través del pin de datos seriales (SDA) y el pin de reloj serial (SCL). Ambos son bidireccionales y deben ser conectados a una fuente de voltaje positiva mediante una resistencia de pull-up.

El modo I2C incluye:

- Modos de direccionamiento de 7 y 10 bits
- Llamada general
- START/RESTART/STOP
- Modo multi-master de transmisión/recepción
- Modo esclavo de transmisión/recepción
- Soporte de modo estándar hasta 100 kbps y rápido hasta 400 kbps
- Frecuencia UCxCLK programable en modo maestro
- Diseño para bajo consumo
- Detección de START en recepción de esclavo para despertarse de cualquiera de los modos LPMx (Low Power Mode).
- Operación de esclavo en LPM4 (Low Power Mode)

UART

En modo UART, la USCI transmite y recibe caracteres a los demás dispositivos de manera asíncrona. Las funciones de transmisión y recepción usan la misma tasa de transferencia. En modo asíncrono, el módulo USCI_Ax conecta el dispositivo a un sistema externo a través de dos pines, UCAxRXD y UCAxTXD. El modo UART se selecciona cuando el bit UCSYNC está en 0.

El modo UART incluye:

- Datos de 7 u 8 bits, con paridad par, impar o sin paridad
- Registros de transmisión y recepción independientes

[8] LPM4 (Low power mode 4) es uno de los modos de bajo consumo del MSP430, en el cual se deshabilitan el CPU y todos los relojes.

- Registros buffer de transmisión y recepción separados
- Transmisión y recepción de datos en modo LSB-first y MSB-first
- Detección de flancos de transmisión para que el receptor despierte del modo LPMx
- Baud rate (tasa de baudios) programable
- Banderas de estado para detección de errores
- Banderas de estado para detección de direcciones
- Capacidad independiente de interrupción de recepción y transmisión

TIMER A

El timer_A es un timer/contador de 16-bits con 7 registros capture/compare. Cuenta con una gran capacidad de manejo de interrupciones, pudiendo generarlas desde el contador debido a una condición de overflow o desde cada uno de los registros de capture/compare.

El timer_A incluye:

- Contador y timer asíncronos de 16-bit con 4 modos de operación: stop, up, continuous y up/down.
- Selección y configuración de reloj externo
- Hasta 7 registros capture/compare configurables
- Salidas configurables para modulación por ancho de pulso (PWM)
- Latcheo asíncrono de entrada y salida
- Registro de vector de interrupción para decodificar rápidamente todas las interrupciones del Timer_A

TIMER B

Diferencias respecto al Timer_A

- El largo del Timer_B es programable, pudiendo ser de 8, 10, 12 o 16 bits, al tiempo que en el Timer_A es fijo, de 16 bits.
- Los registros TBxCCRn[9] del Timer_B tienen doble buffer y pueden ser agrupados, mientras que el en el Timer_A no cuentan con ninguna de esas características.
- Todas las salidas del Timer_B pueden ponerse en estado de alta impedancia
- La función del bit SCCI[10] (bit 10 del registro TAxCCTLn[11]) no está implementada en el Timer_B

Periféricos del MSP430 usados por ANTEL-SAT y su presencia en el openMSP430

El ANTEL-SAT basa la implementación de sus módulos en el chip MSP430 de Texas. Para ganar libertad a la hora realizar campañas de inyección de fallas se optó por emular este integrado en un FPGA[12]. Como punto de partida se tomó el openMSP430 [5] disponible en el sitio web OpenCores, que implementa buena parte del chip original. No obstante, cabe señalar que

[9] Registro de captura y comparación del Timer_B
[10] Synchronized capture/compare input.
[11] Timer_Ax Capture/Compare Control n Register
[12] Field-programmable gate arraw (FPGA) es un circuito integrado diseñado para ser configurado por el usuario luego de manufacturado.

mientras en el satélite se usan integrados de las familias 5 y 6, el core emula el de la familia 1, el cual presenta algunas diferencias con los anteriores [8].

El análisis de la arquitectura del satélite y su funcionamiento, el estudio de la especificación de las diferentes familias del MSP430 y la lectura de la documentación del openMSP430 [9] permitieron determinar qué periféricos o protocolos faltantes en este último sería necesario desarrollar.

A continuación se describen las versiones implementadas en la familia 1 del MSP430 de los periféricos que son usados por ANTEL-SAT. Asimismo, se aclara si están incluidos o no en el openMSP430.

Timers

El timer A es un contador/temporizador de 16 bits con tres registros de captura y comparación. Soporta múltiples capturas y comparaciones, salidas PWM e intervalos de tiempo y una amplia variedad de interrupciones. El timer B es bastante similar, con algunas excepciones. Ambos son empleados por el ANTEL-SAT, en los módulos de comunicación (transmisión y recepción), energía (para el Beacon), control de actitud, conversores ADC/DAC y en el kernel (para los tics). Para algunos de estos módulos se implementa un RTC. El protocolo de comunicación también emplea contadores de los timers para interrupciones. Cabe señalar que para modulación PWM se emplea el timer B.
El timer A está completamente implementado en el openMSP430, mientras que el timer B no lo está.

CRC

La comprobación de redundancia cíclica (CRC) es de un código de detección de errores usado para percatarse de cambios accidentales en los datos. Los paquetes ingresados en estos sistemas contienen un valor de verificación adjunto que permite establecer si sufrieron alteraciones durante su transmisión o almacenamiento. En las familias 5 y 6 del MSP430 existe un periférico encargado de realizar el chequeo de CRC, del cual se valen el módulo de comunicación al enviar o recibir un bloque de datos y el kernel, que tiene la posibilidad de aplicarlo al código. Dicho periférico no está presente en la familia 1, por lo cual no se incluye en el openMSP430.

Watchdog

El watchdog es un mecanismo de seguridad que provoca un reset del sistema en caso de que el mismo permanezca demasiado tiempo ejecutando una misma tarea. Consiste en un temporizador que continuamente decrementa un contador, inicialmente con un valor alto. Cuando este contador llega a cero el sistema se reinicia, por lo que el programa principal debe contener una subrutina que deshabilite el watchdog o le cargue un nuevo valor antes de que se provoque un reset. Si el programa falla o se bloquea, al no actualizarse el contador éste se decrementará hasta cero, reiniciando el sistema.
El openMSP430 implementa en un 100% el periférico de la familia 1 pero el mismo presenta algunas diferencias respecto al de las familias 5 y 6.

Multiplicadores

El hardware multiplicador es un periférico que no forma parte del CPU del MSP430, por lo cual su actividad no interfiere con la de este último. Cuenta con registros periféricos que son cargados y leídos mediante instrucciones del CPU. El multiplicador soporta producto con y sin signo, en ambos casos con la posibilidad de acumulación. Permite multiplicaciones del tipo 16×16 bits, 16×8 bits, 8×16 bits, 8×8 bits.
El openMSP430 cuenta con un multiplicador de 16 bits, mientras que en el satélite se usa uno de 32 bits.

SPI

El Bus SPI (Serial Peripheral Interface) es un estándar de comunicación serial que incluye una línea de reloj, dato entrante, dato saliente y un pin de chip select. En el MSP430 los dispositivos comparten un reloj provisto por el maestro. En el satélite se usa SPI en los módulos de comunicación, energía y actitud.
Este protocolo no está implementado en el openMSP430.

I2C

I2C (Inter-Integrated Circuit) es un bus serial de comunicación. Utiliza una línea para datos (SDA), otra para la señal de reloj (SCL) y finalmente una para la tierra (GND). Se usa principalmente para la comunicación de control general con los demás módulos, o de algunos módulos entre sí. Este protocolo se usa principalmente para la comunicación de control general con los demás módulos, y de algunos módulos entre sí. Se utiliza en modo multimaster y por decisión de diseño sólo se permiten escrituras. Se emplea para transmisión y recepción. Asimismo, el gestor de energía maneja un bus I2C extra para la comunicación con las baterías, pero esto eventualmente se puede obviar.
No se cuenta con este protocolo en el core disponible.

ADC

Un conversor analógico-digital (ADC) es un dispositivo capaz de convertir una señal analógica en una digital que toma valores binarios. La entrada, que varía de forma continua en el tiempo, es muestreada a una velocidad fija, obteniéndose así una señal digital a la salida del conversor. En el satélite es usado por el módulo de comunicación (para audio) y para telemetría.
El openMSP430 no dispone de un ADC implementado.

DAC

Un conversor digital-analógico (DAC) es un dispositivo capaz de convertir una señal digital (de valores binarios) en una analógica (continua). En el satélite es usado por el módulo de comunicación (para audio).
El openMSP430 tampoco posee un DAC.

Otros detalles

Los dos modelos de MSP430 usados en el satélite son de arquitectura extendida, la cual no es soportada por el openMSP430, limitando así la cantidad de funciones compatibles. En ese sentido, es menester modificar los módulos que explícitamente hacen uso de la arquitectura extendida.

El módulo "µkernel" es uno de ellos y se encarga de programar y organizar la ejecución de los programas. Tras el estudio de su estructura y funcionamiento se decidió realizar la migración hacia la arquitectura básica de 16 bits, de modo de acoplar el programa al microprocesador emulado (openMSP430). En secciones venideras se detallan los pasos relativos al *Proceso de migración*.

- Relojes

En el satélite se usan los relojes SMCLK, ACLK, EXT. Todos ellos están disponibles en el openMSP430.

- LPM

El MSP430 tiene varios modos de bajo consumo (LPM). En el satélite el kernel emplea LPM para ahorrar energía cuando no se están ejecutando tareas que requieran uso del microprocesador. En particular, se emplea el LPM0 para apagar el CPU.
El openMSP430 ofrece las opciones de bajo consumo propias del MSP430.

Resumen

La siguiente tabla resume el relevamiento realizado.

Tabla 1: Periféricos usados por ANTEL-SAT y su presencia en el openMSP430

Periféricos utilizados en módulos ANTEL-SAT	Incluidos en OpenMSP430
Timer A	X
Timer B	
RTC	
CRC16	
Watchdog	X
Multiplicador 16x16	X
SPI	
I2C	
ADC	
DAC	
Relojes (SMCLK, ACLK,EXT)	X
LPM (Low Power Mode)	X
Memorias (RAM, ROM)	X

Tal cual se ha comentado, se disponía en un principio del openMSP430, el cual emula el chip MSP430 de la familia 1, mientras que el kernel corre en los chips de las familias 5 y 6. Por un lado al core le falta la implementación de algunos periféricos de la misma familia 1, además de otros propios de las familias 5 y 6. Asimismo, está diseñado para correr en la placa DE1, por lo cual se debe realizar la migración a DE0, que es el FPGA disponible.

3. KERNEL DEL SISTEMA ANTEL-SAT

Introducción

La idea original para la primera etapa del proyecto era a partir del openMSP430 obtener un diseño similar al chip en el cual corre el ANTEL-SAT. En ese sentido a fines de 2013 dispusimos de un core SPI y otro I2C (sin terminar) que fueron heredados como resultado de proyectos de Diseño Lógico 2. Luego de un tiempo de estudio y evaluación se concluyó que el esfuerzo que conllevaría incluirlos en el proyecto sería desproporcionado en comparación con su utilidad. Por lo tanto, se optó por tomar como objeto de estudio el kernel del sistema ANTEL-SAT.

En primer lugar, la tarea fue comprender su estructura y funcionamiento, y adaptarlo de manera que pudiera funcionar en el openMSP430. También hubo que realizar modificaciones para poder emplear compiladores y herramientas de debug diferentes a las originales. Luego se procedió a diseñar algunas aplicaciones que pudieran correr sobre él para finalmente inyectar fallas durante la ejecución de las mismas.

Generalidades

El kernel desarrollado para el satélite ANTEL-SAT [10] es de ejecución en tiempo real e incluye entre sus funciones la ejecución y sincronización de aplicaciones, servicios, tareas concurrentes y transferencia de datos. Particularmente se trata de un módulo que provee las funcionalidades más básicas de un sistema del tipo descripto, por lo cual se lo ha llamado µkernel. El µkernel original ocupa cerca de 50 kB, pero en nuestro proyecto se usó una adaptación simplificada que apenas supera los 10 kB. De hecho, la versión de trabajo empleada fue cambiando conforme iba avanzando el desarrollo del ANTEL-SAT.

Entre las ventajas de este sistema se encuentran su modularidad, su tamaño pequeño y la capacidad de adaptación a otros requerimientos. El Para lograr su cometido utiliza diferentes módulos, tales como manejador de procesos, manejador de tiempos, manejador de interrupciones externas y semáforos para sincronizar procesos. [11]

El sistema está diseñado con cierta robustez para poder soportar las condiciones del ámbito en el que se utilizará, lo cual implica manejo de redundancia e inclusión de funciones que permitan su recuperación ante fallas. Entre sus características se distingue el diseño orientado a un medio en el que los procesos se ejecutan en forma permanente, es decir que no finalizan.

Cada proceso tiene asociado un conjunto de datos de control incluidos en un bloque llamado Process Control Block (PCB), que contiene información detallada del mismo. Otra característica a destacar es la posibilidad de que exista más de una instancia del mismo código ejecutando diferentes procesos.

Funcionamiento general

El kernel sincroniza los procesos asignándoles el procesador por orden de llegada. Es decir, cada proceso entra en una cola de espera ordenada según el momento en que es cargado a la cola. Así los procesos de mayor prioridad (primeros en la fila) tienen preferencia sobre los de menor prioridad.

Por otro lado, cabe señalar que el sistema no le quita el procesador a una aplicación hasta que ésta lo ceda a otra que lo necesite. Para ello el programador define un tiempo de timeout asociado a cada proceso, el cual es incluido al ser cargado el mismo. Este tiempo se utiliza para configurar un "watchdog" cada vez que se entregue el procesador a una tarea. Si el proceso no devuelve el procesador antes de que el temporizador expire, el "watchdog" se encarga de resetar el sistema.

Funciones del µkernel

Para comprender el funcionamiento del sistema y poder cargar aplicaciones de prueba en nuestro prototipo fue necesario familiarizarnos con el funcionamiento del µkernel, lo cual requirió un estudio minucioso de las diferentes funciones dedicadas a inicialización de procesos, administración de los mismos y uso de aplicaciones.

Se detalla a continuación una breve descripción de las funciones del µkernel:

Funciones de inicialización

- kernel_init: controla la integridad del código e inicializa los datos del kernel (Process Control Block (PCB), Resource Control Block (RCB), contador de procesos, etc.). Su ejecución cambia el estado del kernel a "INITIALIZED".

- load: inicializa el contexto del proceso. Recibe como parámetros: aplicación a ejecutar, dirección de memoria que se entregará al programa, menor dirección de memoria RAM del área reservada como stack, tamaño del stack en palabras y tiempo de watchdog.

- run: ejecuta el despachador (scheduler). No tiene parámetros de entrada y retorna sólo en caso de error. Su ejecución cambia el estado del kernel a "RUNNING".

Funciones para uso de aplicaciones

- get_resource_status: informa acerca del estado de un recurso. Como parámetro de entrada recibe el identificador del recurso que se desea consultar.

- lock: intenta obtener el control de un recurso. Si éste se encuentra bloqueado por otro proceso, el proceso llamador se coloca en la cola de espera para el recurso en cuestión, quedando a la espera de que el recurso quede disponible o hasta que expire su tiempo de "timeout". Su ejecución con salida "NO_ERROR" cambia el estado del recurso a "LOCKED". Utiliza como parámetros de entrada el identificador del recurso y el tiempo máximo de espera expresado en tics. Un timeout de 0 implica esperar hasta que el recurso quede disponible.

- unlock: libera un recurso que estaba siendo utilizado (bloqueado) para que otros procesos puedan usarlo. Si existían procesos a la espera por el recurso, el kernel le asigna el recurso al que siga en la cola de espera. Como parámetros recibe únicamente el identificador del recurso a liberar.

- set_buffer: define el espacio de transferencia de datos asociado al recurso referenciado. El recurso debe haber sido previamente bloqueado por el proceso llamador. En caso de que haya datos sin leer en el buffer se reporta el error y se define el buffer con los nuevos parámetros. Estos últimos son: identificador del recurso al que se quiere asignar el espacio de transferencia, dirección de dicho espacio y su tamaño expresado en bytes.

- reset_buffer: restablece el espacio de transferencia de datos asociado a un recurso. El recurso debe haber sido bloqueado por el proceso llamador. Si el proceso había ejecutado la función set_buffer, reset_buffer restablecerá el mismo buffer. El parámetro que recibe como entrada es el identificador del recurso al cual se quiere asignar el espacio de transferencia.

- read: bloquea un proceso hasta que el recurso utilizado tenga datos prontos para ser leídos o se alcance el tiempo máximo de espera. La función permite leer datos mayores al tamaño de buffer mediante una señalización especial. Cabe señalar que el kernel no almacena datos, sino que el flujo se logra mediante la intervención de un emisor que inyecta datos y el receptor que los recibe. Como parámetros de entrada la función read recibe: identificador del recurso por el que se desea esperar, dirección en la que se almacenará el largo de datos recibidos (se usa el 0 para ignorar el resultado) y tiempo máximo de espera expresado en tics (se usa 0 para esperar por tiempo indefinido).

- write: bloquea el proceso hasta que el recurso a utilizar se encuentre en estado de espera de datos o se alcance el tiempo definido. Si el largo a trasmitir es mayor que el espacio de buffer definido por el proceso que espera los datos (proceso que hizo set_buffer), se copian tantos bytes como permita el buffer y se reporta el error a los dos procesos. Los parámetros que recibe la función son: identificador del recurso, espacio de transferencia (se usa 0 si no se transfieren datos), largo de datos a enviar, dirección en la que se almacenará, largo de datos entregados (0 para ignorar el resultado) y tiempo máximo de espera en tics (0 para esperar por tiempo indefinido).

- writei: análoga a la función write pero nunca bloquea al recurso. Es ideal para utilizar en rutinas de atención a interrupciones para transferencia de datos o señalización.

- tsleep: suspende al proceso durante un tiempo. El proceso pasa a estado “READY” después de que el kernel detecta que se consumieron tantos intervalos de tiempo como los indicados en el parámetro. El proceso continuará cuando no exista ningún otro en estado “READY” de mayor prioridad. Un proceso puede disminuir su nivel de prioridad mediante el uso de esta función con parámetro 0. El tiempo de espera está acotado por el tamaño de palabra del procesador, siendo de 109 minutos para una palabra de 16 bits con tiempo de tic de 100 ms. Como parámetro de entrada recibe el tiempo expresado en tics.

- sleep: suspende el proceso durante un tiempo expresado en segundos. El proceso puede disminuir su nivel de prioridad mediante el uso de esta función con parámetro 0. El tiempo máximo queda definido por el tamaño del entero largo del sistema (unsigned long). Para un entero largo de 32 bits la espera máximo es del orden de unos 136 años. Como parámetro de entrada recibe el tiempo expresado en segundos.

- stop: detiene indefinidamente la ejecución de un proceso. La función no recibe ningún parámetro.

Programador de Procesos

El “scheduler” o despachador de procesos requiere un apartado especial, en virtud de su mayor complejidad e importancia respecto a las funciones anteriores. A continuación se describe la manera en que éste administra los procesos de la cola de espera.

La función es ejecutada por tiempo indefinido dentro de un “for(;;)”, según la siguiente secuencia:

1. Se consulta la cantidad de tics generados por interrupción del timer A.
2. Mientras la función “get_tics” no devuelva 0 (hay tics), se recorre toda la tabla de procesos del primero “pid = 0” hasta el último “process_count”.
   Si se encuentra a un proceso que esté a la espera de escritura (write waiting), lectura (read waiting), o bloqueo (lock waiting), se resta una unidad a su variable “timer”.
   Se evalúa si el “timer” del proceso llegó a 0 y en caso afirmativo se consulta si es el primero de la cola de espera. De cumplirse esto se libera el proceso y de lo contrario se asigna su identificador al siguiente lugar en la cola. Luego el proceso queda en estado “READY”. Se retorna entonces al punto 1, en el cual se consulta nuevamente si hay tics.
3. Ahora, suponiendo que “get_tics” devuelve 0 (no hay tics), significa que ya no hay más intervalos de tiempo que esperar y el “timer” de todos los procesos ha llegado a 0. Entonces se busca el primer proceso listo (“READY”) en la cola de procesos y se ejecuta la función de chequeo de corrupción de memoria de datos “low_mem_CRC16”. Si el resultado es favorable se setea el “watchdog” y mediante la ejecución de la función resume() se salta a ejecutar el proceso correspondiente.

El siguiente diagrama de flujo ilustra la secuencia que se acaba de explicar.

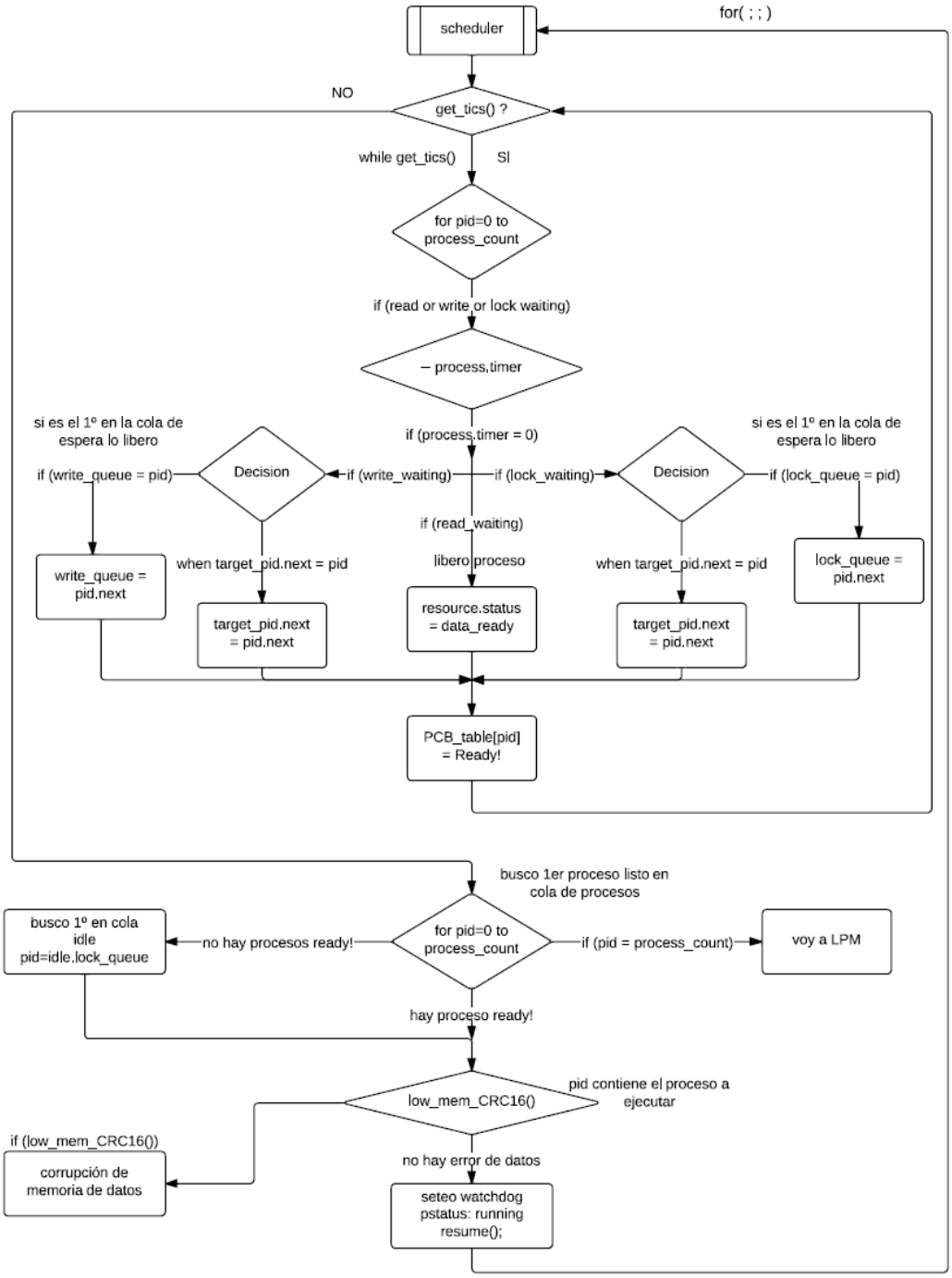


Figura 3: Diagrama de flujo de la función "scheduler"

A modo de ejemplo y repaso se incluye la figura 3, que se representa una situación usual en la cual varios procesos, alojados en la tabla PCB, esperan a ser atendidos.

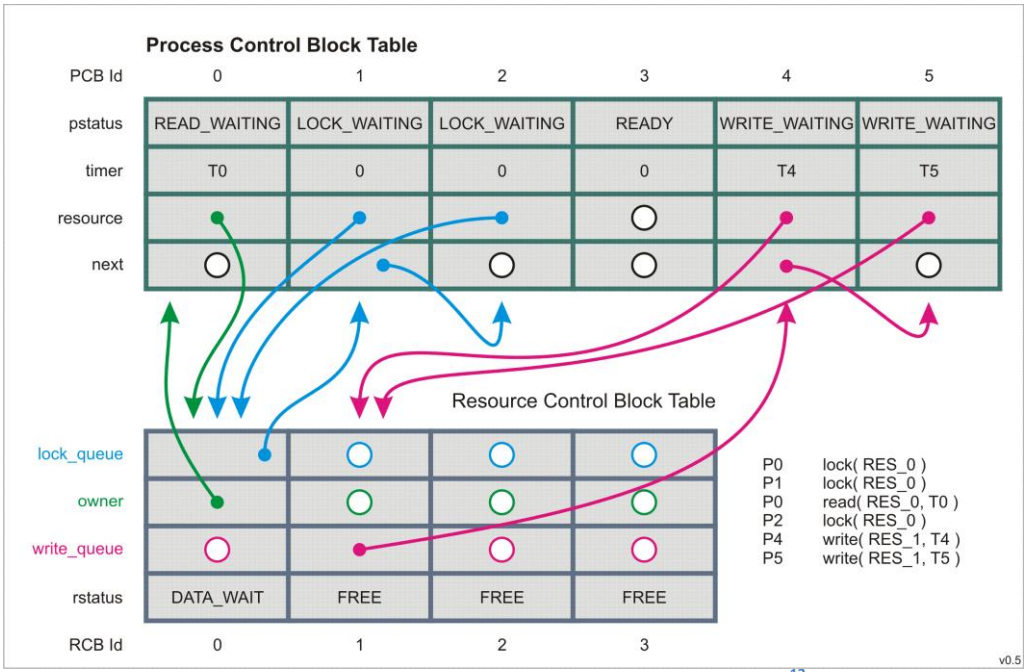


Figura 4: Secuencia de procesos y asignación de recursos [13]

La fila "*pstatus*" muestra el estado en que queda el proceso luego de ejecutarse los comandos correspondientes para cada uno de los procesos 'Pi', para i de 0 a 5.

La secuencia transcurre de la siguiente manera:

1. El proceso 0 quiere bloquear al recurso 0; "lock (RES_0)". Si el recurso no se encuentra bloqueado entonces es cedido al proceso 0 con cierto tiempo de "timeout" (watchdog), quedando ahora bloqueado. Antes de que transcurra este tiempo el proceso debe desbloquear el recurso con un "unlock", de lo contrario se da un reset del sistema.

2. Luego el proceso P1 hace un "lock(RES_0)". Al estar el recurso 0 bloqueado por P0 (si aún no lo ha liberado) P1 se coloca en la cola de espera de bloqueo "lock_queue" y queda en estado "lock_waiting". Al tener un "timeout" igual a 0 el bloqueo es por tiempo indefinido hasta que el recurso sea liberado.

3. Ahora P0 quiere hacer un "read" del recurso 0, o sea read(RES_0,To). Por lo tanto, si está disponible lo bloqueará ("lock") ya sea hasta liberarlo poniéndolo en estado "data_ready" o por un tiempo máximo de "timeout". En este caso el recurso se encuentra bloqueado (su estado no es "data_ready") por P0, y a la espera de ser bloqueado por P1. Entonces P0 pone su estado en "read_waiting" y suspende el proceso (se ejecuta "suspend"; esta función llama al "scheduler").

4. Aparece un nuevo proceso P2 que quiere bloquear el recurso 0. Como este último aún no ha sido liberado, P2 se pone en la cola de bloqueo quedando en estado "lock_waiting". Suponiendo que en la cola de espera de bloqueo sigue P1 esperando, P2 se coloca en la cola seteando la variable ".next" del proceso P1 con el valor '2' haciendo referencia al valor de su índice. Al no haber más procesos en la cola de espera de bloqueo, la variable ".next" de P2 se carga con 'NO_PID' (no siguen procesos en la cola). ¿Qué ocurre cuando

[13] Figura extraída del Manual de usuario µKernel Versión 6.14, Gustavo de Martino, 2013.

P1 bloquea el recurso 0? Le cede el primer lugar en la cola a su proceso con el índice guardado en su variable ".next".

5. Algo similar ocurre con los dos últimos comandos, ejecutados por los procesos 4 y 5. P4 se coloca en la cola con el primer lugar y luego llega P5 a colocarse en la siguiente posición. La ubicación que toman en la cola no coincide con el orden de ejecución, sino que este último es determinado por el tiempo de "timeout".

Estados de los recursos

A continuación se incluye la figura 4, donde se representan los posibles estados de un recurso dependiendo de la acción que ejecute un proceso:

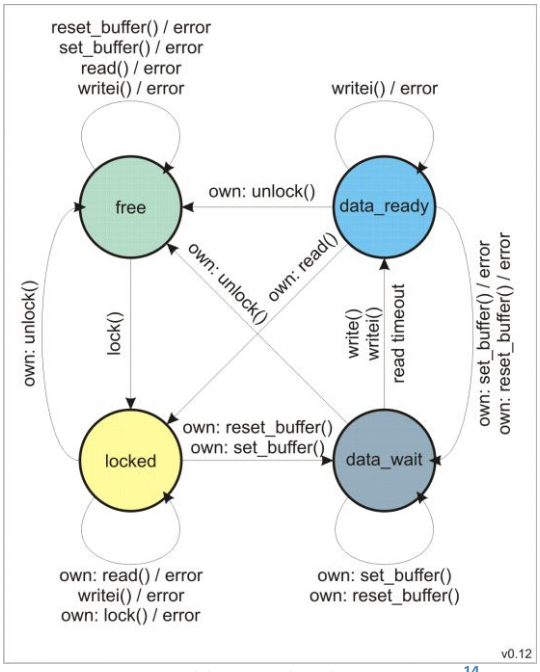


Figura 5: Posibles estados de un recurso [14]

Proceso de migración

Los módulos del ANTEL-SAT fueron implementados con la herramienta de desarrollo para sistemas embebidos de "IAR Systems". Por lo tanto estas versiones sólo compilan con el compilador IAR y será necesaria una adaptación de las mismas en caso de querer hacerlo con otras herramientas.

En nuestro caso particular, se utilizan herramientas las de "debugging" "openmsp430-gdb" y "openmsp430-minidebug", que fueron desarrolladas específicamente para el "openMSP430" en un ambiente de desarrollo GNU y sólo admiten archivos de extensión ".elf". Fue necesario entonces migrar todos los módulos del sistema para compilar con las herramientas nativas que brinda el openMSP430. Se genera de ese modo el archivo de extensión ".elf" necesario para debuggear con la herramienta "openmsp430-minidebug".

[14] Figura extraída del Manual de usuario µKernel Versión 6.14, Gustavo de Martino, 2013.

A continuación se pasan a enumerar los módulos originales que componen el kernel del sistema, seleccionando luego algunos para incluir en una primera versión capaz de correr en el openMSP430. Finalmente, se detallan las etapas que debieron cumplirse para cumplir con el proceso de migración.

Módulos originales

El sistema µkernel original se compone de los siguientes archivos:

- ukernel.c: contiene las funciones de inicialización y funciones para uso de aplicaciones.
- ukernel.h: es la interfaz del módulo "ukernel.c" y contiene las declaraciones de todas las funciones de "ukernel.c".
- ukernel_ll.h: declara de rutinas de bajo nivel para el micro MSP430.
- ukernel_hal.h: contiene las declaraciones de las funciones relativas al microprocesador MSP430. Se incluye el encabezado de la función "hardware_init" que inicializa el hardware; la función "reset" que reinicia el microprocesador; y la función low_power_mode.
- MSP430X_ukernel_ll.s43: módulo que implementa las funciones de bajo nivel para MSP430X en modo 20 bits.

El µkernel necesita de un módulo principal que lo arranque, es decir, se encargue de inicializar el hardware y las funciones del sistema, cargar algunos procesos para realizar transmisión y recepción de datos, y ejecutar la función "run", que dejará al sistema operando normalmente. El módulo de control principal "mcs.c" es un ejemplo de esto, el cual se tomó como punto de partida en nuestro trabajo.

Sub-módulos del módulo de control principal:

- mcs.c: es la raíz del módulo de control principal. Inicializa el sistema, reserva un espacio de stack para los procesos que van a ser cargados, inicializa el hardware del MSP430, carga los procesos y ejecuta la función "run".
- mcs_hal.c: es la capa de abstracción de hardware para MCS con el MSP430. Aquí se implementan las funciones de acceso al timer y de "reset", se configura el watchdog y se atienden las interrupciones generadas. Incluye las librerías "io430.h" de IAR para microprocesadores MSP430 y a la interfaz "ukernel_hal.h", que declara las funciones relativas a dicho integrado. También incluye a common.h, que contiene las declaraciones comunes a todos los módulos (definición de variables, recursos e identificación de los módulos del satélite).
- ukernel_config.h: Define constantes relacionadas a cantidad de procesos a ser cargados, cantidad de recursos, control de overflow y consistencia del área de stack.

Se incluye también la implementación de 2 aplicaciones ejemplo a ser cargadas. Éstas son "ax25_layer" y "immp_config", las cuales contienen las funciones "immp_tx", "immp_rx" y "ax25_rx", utilizadas para transmisión y recepción de datos.

Con el fin de adaptar los módulos para que compilaran en la versión 4.8.0 de GCC se llevaron a cabo una serie de pasos descriptos a continuación.

Módulos para primera versión de "µkernel"

Se realizó una selección de los módulos necesarios para el desarrollo de una primera versión de "µkernel" que corriera en el openMSP430 y permitiera ejecutar una aplicación sencilla. En ese sentido, se escogieron los siguientes: "mcs.c", "mcs_hal.c", "mcs_hal.h", "ukernel.c" "ukernel.h", "ukernel_hal.h", "ukernel_config.h", "common.h", "msp430X_ukernel_ll.s43" y ukernel_ll.h.

Modelo openMSP430 para compilar en IAR

Esta etapa fue necesaria para comenzar a adaptar todo el sistema µkernel a una plataforma de microprocesadores con las características del openMSP430. La herramienta IAR Systems contiene una extensa lista de modelos de microcontroladores MSP430 que sirven de referencia al momento de compilar. Cada uno de estos modelos se compone de los siguientes archivos: MSP430xxxx.menu, msp430xxxx.h, msp430xxxx.ddf, lnk430xxxx.xcl y msp430xxxx.sfr.

EL archivo MSP430xxxx.menu contiene la configuración por defecto de algunos parámetros de la familia. Estos son por ejemplo: nombre de la familia, dirección base de vector de interrupciones, tensión VCC, "stackdefault", "heapdefault", etc. El archivo de interfaz msp430xxxx.h contiene la definición estándar de todos los registros y bits de la familia.

Los archivos del tipo msp430xxxx.ddf poseen toda la información de memoria, donde se definen los diferentes espacios y tipos. Se incluye además la tabla de interrupciones de cada uno de sus periféricos, donde se definen prioridades y direcciones.

El archivo lnk430xxxx.xcl incluye la configuración del linker para la familia. El mismo debe ser referenciado en la pestaña "linker" del menú "options" del nodo principal, en este caso mcs.c. En la figura 5 se aprecia una captura de pantalla del menú descripto.

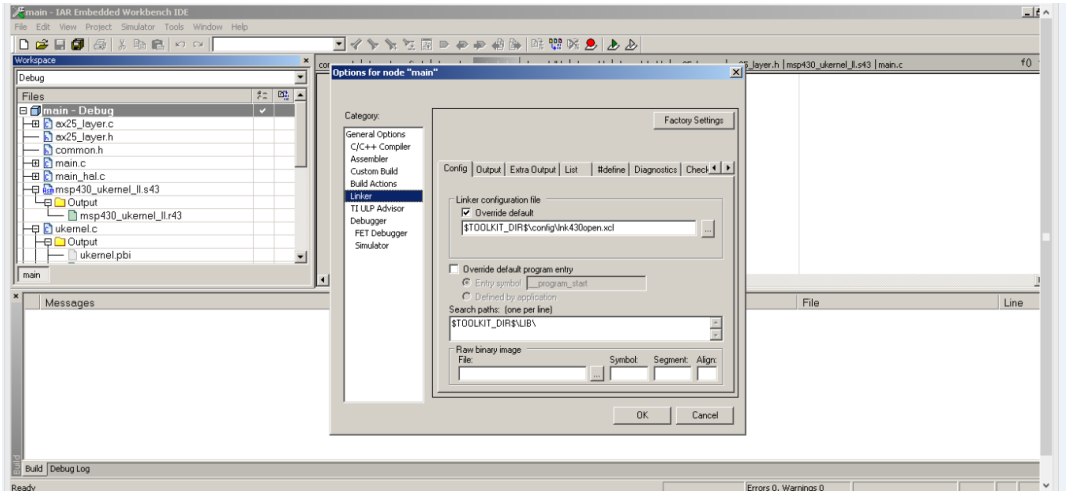


Figura 6: Configuración del linker

Por último msp430xxxx.sfr contiene las definiciones de registros y bits de todo el espacio de periféricos de entrada/salida de la familia en cuestión.

Una vez estudiado el código de cada uno de estos archivos se procedió a su adaptación para la familia "openMSP" a ser utilizada en IAR. Para ello se tomó como base una copia de cada uno

de los archivos de la familia MSP430C111 (de las más sencillas) a partir de la cual se generaron los siguientes archivos: MSP430OPEN.menu, msp430open.h, msp430open.ddf, lnk430open.xcl y msp430open.sfr. Para que los módulos compilaran fue necesario incluir el nombre del MSP430open.h en la librería io430.h de IAR. Luego de esto y de haber generados los archivos correspondientes al modelo de la familia openMSP430 se elige al dispositivo MSP430open en la pestaña "Target" ubicada en Options > General Options.

En la siguiente figura se muestra el menú de selección de la familia y como aparece el modelo diseñado a medida para el openMSP430.

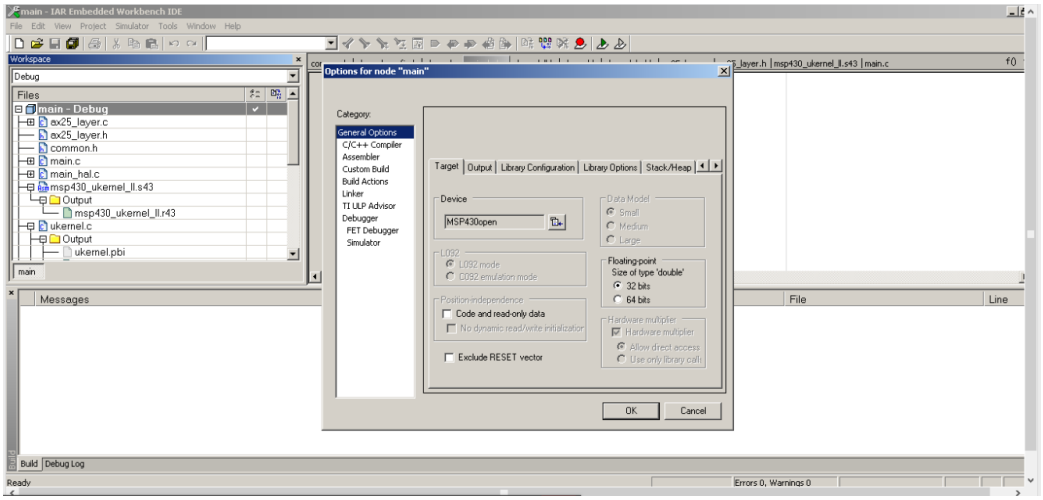


Figura 7: Menú de selección de la familia

Adaptación de módulos del µkernel y compilación en IAR

Para esta fase fue preciso realizar diferentes modificaciones a los módulos, las cuales se detallan a continuación.

- El archivo msp430X_ukernel_ll.s43 se renombró (msp430_ukernel_ll.s43), adaptando su código para ser ensamblado en plataformas de 16 bits como el openMSP430. Para ello se modificaron las siguientes instrucciones:

Tabla 2: Adaptación de instrucciones a arquitectura de 16 bits

	arquitectura	
código	**20 bits**	**16 bits**
MOV	MOVX.A	MOV.W
PUSH	PUSHX.A	PUSH.W
BIS	BIS	BIS.W
RET	RETA	RET
POP	POPX.A	POP.W
SUB	SUBX.A	SUB.W

- En el módulo ukernel.c se comentó la inclusión del archivo ukernel_ll.h ya que este no se usará más. El compilador deberá ir a buscar las funciones implementadas en assembler al archivo msp430_ukernel_ll.s43.

- En mcs_hal.c se comentan las atenciones de interrupción a periféricos no pertenecientes a la familia openMSP430, como lo son el TimerB, USCI_A, USCI_B, RTC. Se sustituyeron las funciones que configuran al Timer B por líneas de código que configuran al Timer A utilizado por el openMSP430. Se configuró dicho periférico modificando la función system_timer_config, donde se selecciona la fuente de reloj auxiliar (ACLK). Fue necesario habilitar interrupciones, configurarlo para trabajar en bajas frecuencias y por último definir la constante del timer, en este caso para trabajar con interrupciones de 100ms. Se modificó también la función low_mem_crc16 para que al ejecutarse omita el chequeo de memoria y devuelva únicamente el valor 0xFFFF, lo cual implica que no hubo corrupción de memoria. Esta simplificación le quita complejidad al sistema, lo cual fue necesario en primera instancia para obtener un prototipo básico y evitar complicaciones innecesarias. En etapas posteriores se mostrará cómo se incorporó esta funcionalidad.

- En mcs.c se implementó una primera versión del módulo de control principal en la que se comentaron las líneas que definen los espacios de stack de las aplicaciones originales (immp_tx, Immp_rx y ax25_rx) y se incorporó una nueva variable de stack para la primera aplicación básica a correr en nuestra versión de µkernel. Se agregó también la rutina de inicialización del Timer A (system_timer_config) y una línea en la función load para cargar la nueva aplicación. A continuación se muestra el código del módulo de control principal para esta versión de µkernel.

```
#include "hardware.h"
#include "ukernel_hal.h"
#include "ukernel.h"
//#include "immp.h"
//#include "ax25_layer.h"
#include "main_app1.h"

void main( void )
{

      //stack_t immp_tx_stack[0x100];
      //stack_t immp_rx_stack[0x100];
      //stack_t ax25_rx_stack[0x100];
      stack_t app1_stack[0x100];

      //Rutinas de inicialización
      hardware_init();
      system_timer_config(); //configura TimerA
      kernel_init ();
      //immp_init ();

      //Carga de aplicaciones

      //load( immp_tx, 0, immp_tx_stack, 0x100, WDTIS1+WDTIS0 );
      //load( immp_rx, 0, immp_rx_stack, 0x100, WDTIS1+WDTIS0 );
```

```
        //load( ax25_rx, 0, ax25_rx_stack, 0x100, WDTIS1+WDTIS0 );
        load( app1, 0, app1_stack, 0x100, WDTIS1+WDTIS0 );

        run();

        return 0;
}
```

Finalmente, se obtiene una versión del µkernel compilando para IAR que se ajusta las características del openMSP430 utilizado. A partir de ello se logró generar un archivo .elf pero el mismo no contenía la información de debugging necesaria para ser usado en GDB. Al no poder generarla desde IAR, fue necesario entonces migrar los módulos ajustados al modelo del micro que estamos emulando, pero esta vez para ser compilados con GCC y de ese modo poder hacer uso del debugger.

Adaptación para compilador GCC y ensamblador MSP430-AS

Para migrar de entorno de compilación se tomó como base la estructura de archivos de los programas de ejemplo que se incluyen en el proyecto del microcontrolador openMSP430. En particular se identificaron los siguientes archivos como críticos para la compilación en GCC: link.ld, makefile y omsp_system.h. Los mismos resultan imprescindibles ya que aportan la información del micro y del programa que se está compilando. En particular:

- link.ld: contiene la información del espacio de memoria de datos, texto, vectores y secciones.
- makefile: define el formato de las salidas de los diferentes módulos, así como la jerarquía de los mismos y el compilador que se desea usar (en este caso msp430-gcc).
- omsp_system.h: es el análogo al msp430open.h de IAR que es incluido en el io430.h, pero para el caso del compilador GCC. El archivo contiene todas las definiciones de bits de los diferentes registros y periféricos del openMSP430.

Por lo tanto, antes de compilar es necesario incluir estos 3 archivos dentro de la carpeta contenedora junto a los demás módulos del µkernel y modificar el archivo makefile. Para adaptar este último a las necesidades de nuestra versión de µkernel y aplicaciones que correrán en él, se tomó como base la versión del archivo incluido en el ejemplo memledtest del proyecto openMSP430. A continuación se incluyen fragmentos de código que ilustran lo realizado.

```
# makfile configuration
NAME            = mcs
OBJECTS         = mcs.o   mcs_hal.o    ukernel.o    a.out    main_app1.o

CFLAGS            = -O2 -Wall -g -mcpu=430 -mivcnt=16 -mmpy=16      # Uniarch
flags

# use custom linker script
LDFLAGS         = -Tlink.ld

#switch the compiler (for the internal make rules)
CC              = msp430-gcc
```

```
LD                = msp430-gcc
```

El código anterior muestra el tipo de salida del resultado de la compilación de cada módulo y la versión del compilador que se quiere utilizar.

Se puede distinguir también que se define una salida de nombre a.out, la cual se corresponde con el archivo msp430_ukernel_ll.s43 y es la salida del ensamblador msp430-as que es llamado por el GCC.

```
#project dependencies
main.o:      mcs.c  ukernel_hal.h      ukernel.h    hardware.h  main_app1.h
ax25_layer.o:      ukernel.h
main_hal.o:  hardware.h   ukernel_hal.h      common.h  7seg.h  mcs_hal.c
ukernel.o:   ukernel_config.h   ukernel.h    ukernel_hal.h
      msp430_ukernel_ll.s43
a.out: msp430_ukernel_ll.s43
main_app1.o: main_app1.c hardware.h ukernel.h
```

Luego de concluir con la definición de los módulos, las salidas y el compilador a utilizar, resta realizar la compilación utilizando el programa make.exe incluido en las herramientas de Tool Command Language (TCL). Para ello todos los módulos se almacenan dentro de una carpeta a la cual se apunta desde ventana de comandos. Mediante el comando make se inicia la compilación.

Si hay errores, los mismos se muestran en la ventana de comandos, indicando línea y módulo donde se encuentran. De esta manera, utilizando como guía los errores de compilación e investigando las instrucciones equivalentes en GCC, se fueron depurando los módulos del sistema hasta conseguir que compilaran satisfactoriamente.

A continuación se listan las modificaciones que fueron necesarias para compilar los módulos del µkernel en GCC:

- En el módulo mcs_hal.c se agrega el archivo hardware.h, que además incluye la librería omsp_system.h del openMSP para GCC.

- Los encabezados a las rutinas de atención a las interrupciones cambian:
  Compilando con IAR:
  - #pragma vector = WDT_VECTOR
    _interrupt void WDTISR (void) {}
  - #pragma vector = TIMER_Ax_VECTOR
    _interrupt void TIMERx_AxISR (void) {}

  Compilando con GCC:
  - wakeup interrupt (WDT_VECTOR) INT_Watchdog (void)
  - wakeup interrupt (TIMERAx_VECTOR) INT_TIMERx (void)

- Se quita la función __no_operation();

- Hay algunas diferencias en el modo de armar las máscaras para definir valores en los registros, por ejemplo:
  - para IAR es TACTL = TASSEL_1 + MC_1 + ID_0

- para GCC es TACTL = TASSEL_1 | MC_1 | ID_0

- En ukernel.c se eliminaron los encabezados "__no_init" en la definición de variables.
  Las funciones para deshabilitar interrupciones cambian:
  - para IAR: __disable_interrupt();
  - para GCC: __dint();

  Lo mismo ocurre para las funciones de habilitación:
  - para IAR: __enable_interrupt();
  - para GCC: __eint();

  Observación 1: Para la segunda campaña de inyección de fallas se incluyeron los encabezados "__no_init" con un "define", como se muestra a continuación:

```
//#include "release.h"
//#pragma message("µKernenl v" KERNEL_VERSION_NUMBER "."
KERNEL_REVISION_NUMBER " by GDM "__DATE__ " " __TIME__ )

#include "ukernel_config.h"
#include "ukernel.h"
#include "ukernel_hal.h"
#include "tools.h"
#include "intrinsics.h"

#define __no_init    __attribute__ ((section (".noinit")))
```

  Observación 2: La versión del kernel utilizada en la segunda campaña incluye la función "__istate_t interrupt_state = __get_interrupt_state()", a partir de la cual se define la variable "interrupt_state", del tipo "__istate", que guarda el estado del vector de interrupciones en un momento dado. En el instante en que se quiera devolver el estado guardado se llama a la función "__set_interrupt_state(interrupt_state)", pasándole la variable "__istate" como parámetro de entrada. Fue necesario incluir la librería "intrinsics.h" para lograr que GCC reconociera dichas funciones.

  También fue preciso implementar código de GCC en inline assembler para suplantar la función __get_SP_register(), que es nativa de IAR y devuelve el valor del stack pointer. Se implementó agregando la siguiente línea al código:

```
asm ("MOV r1, %0": "=r" (kernel_sp) : : "r1")
```

  Una explicación mediante ejemplos del uso de esta opción se adjunta en el *Anexo 7: GCC inline assembler*.

  Hasta aquí se detallaron todas las modificaciones que fueron necesarias para lograr compilar los módulos del µkernel con msp430-gcc. No obstante, aún resta ensamblar el archivo msp430_ukernel_ll.s43, el cual contiene código de funciones escritas en bajo nivel.

  Para ensamblar el archivo de extensión .s43 se utilizó el msp430-as, el cual se incluye entre las herramientas recomendadas por el autor del proyecto openMSP430. Haciendo uso del manual "Using as" [12] y analizando los errores desplegados al querer ensamblar el archivo, se encontró pertinente realizar las siguientes modificaciones:

- Se reemplaza la directiva NAME msp430_ukernel_ll de IAR por el método de etiquetado de AS a través del uso de ":", quedando de la forma "msp430_ukernel_ll de IAR:".
- Los comentarios mediante "//" que utiliza IAR fueron sustituidos por el símbolo punto y coma ";".
- La directiva "RSEG" de IAR utilizada para alojar secciones de código en memoria fue reemplazada por ".section NOMBRE, "w", @progbits" para la sección de memoria de datos, y ".section .text" para memoria de programa.
- En la declaración de variables se reemplaza:
  - EXTERN var1 por .extern var1
  - PUBWEAK var2 por .weak var2
  - PUBLIC var3 por .global var3
- El ensamblador AS para msp430 no admite los nombres SP y SR utilizados en el código del archivo assembler, de modo que fue necesario sustituir dichas designaciones por su equivalente en registros. De acuerdo al manual del openMSP430, al SP le corresponde el registro r1 y al SR el r2. Esta información se puede ver en la tabla de registros presente en la interfaz gráfica de la herramienta openmsp430_minidebug.
- Se comentó la directiva REQUIRE utilizada en IAR, dado que no se encontró sustituto para la misma en AS.
- Hace falta ahora identificar el uso que GCC le da a los registros al momento de pasar parámetros de entrada a una función determinada y cuál es la diferencia respecto a IAR. Para descifrar esta incógnita se diseñó un módulo que invoca a las diferentes funciones implementadas en bajo nivel, identificando mediante la herramienta de debugger los registros implicados. En el código simplemente se llaman las funciones en cuestión y se incluyen los módulos necesarios, tal como se muestra a continuación:

```
#include "ukernel_ll.h"

void main() {

unsigned short     stack_pointer;

void(*program)(void*);
program = 0xff;

void* parameters;
parameters = 0x55  ;

typedef unsigned short stack_t;
stack_t* stack;

void (*scheduler) (void);
scheduler = 0xee;

unsigned short kernel_sp = 0x1111 ;

stack_pointer = init_program_stack( (unsigned short) &stack, parameters,
program);
```

```
resume(stack_pointer);

task_change(scheduler, kernel_sp);

stack_pointer = get_suspend_sp();

}
```

Debuggeando el código anterior y registrando los valores que tomaban los registros paso a paso, se observó que GCC usa los registros a partir del r15 en orden descendiente. Es decir que si la función es func(var1,var2,var3), GCC usa respectivamente los registros r15, r14 y r13 para pasar los valores de cada variable. Resta únicamente sustituir dichos registros por los utilizados en el archivo original.

Aplicaciones de prueba

Al concluir exitosamente las etapas anteriores queda lista la versión del sistema µkernel para ser compilado y debuggeado con las herramientas desarrolladas para el openMSP430. Se adjunta al final del documento un instructivo con los pasos a seguir para compilar y debuggear el sistema "µkernel + aplicación" (*Anexo 6: Uso del minidebugger*).

Para verificar el correcto funcionamiento del µkernel, se programaron algunas aplicaciones de prueba para correr en él. A continuación se presentan dos escenarios generados para comprobar la operación del µkernel.

Aplicación sencilla

Esta primera tarea básica de prueba contiene un contador que incrementa la cuenta en 1 cada vez que la aplicación es ejecutada. Al finalizar cada corrida, le pide al kernel ser despertada luego de 2 tics del timer A0 (200ms) y así incrementar el contador nuevamente (cont = cont + 1). A medida que la cuenta aumenta se va almacenando el valor de 2^cont en una variable llamada num, es decir que num=2^cont. Cuando el contador alcanza el valor 3 la aplicación comunica al kernel que no se ejecutará más, llamando a la función stop() y finalizando para siempre. Se pueden consultar más detalles en el *Anexo 5: Carga y ejecución de aplicación*.

Dos aplicaciones en simultáneo

En este caso se realizó una prueba con dos aplicaciones corriendo simultáneamente en µkernel y comunicándose entre sí. Si bien este modo no se utilizó para las campañas de inyección de fallas fue útil a la hora de familiarizarnos con el uso del kernel y para chequear su correcto funcionamiento.

Al haber 2 aplicaciones coexistiendo en el µkernel es necesario que la función (que en este caso recibe datos de "mcs_gray_f") defina un buffer para lograr un hilo de comunicación, y que además llame a la función lock(), quedando a la espera de un recurso libre. De esta manera la aplicación "mcs_gray_f" coloca los datos llamando a la función "write()", pasando como parámetros adicionales el buffer creado y el recurso utilizado por ambas aplicaciones, entre otros.

A continuación se incluye el código del módulo principal del kernel “mcs.c”, donde se puede observar la manera en que se incluyen ambas aplicaciones: “mcs_gray_f” presente en el módulo "mcs_gray.c" y “f” correspondiente al propio “mcs.c”. Nótese que en este caso se incluyen las funciones “lock()” y “set_buffer()” en la aplicación “f”.

```
#include "hardware.h"
#include "ukernel_hal.h"
#include "ukernel.h"
#include "mcs_gray.h"

int proceso1_input_buffer;

void f (void *p){

	for(;;){

		//bloquea indefinidamente a la espera de un recurso
		lock(resource_1,0);

		//define un espacio de transferencia del tamaño de un entero
		set_buffer(resource_1, &proceso1_input_buffer, 2);

		read(resource_1,0,0);

		//prendo led con número leído
		P3OUT = proceso1_input_buffer;

		unlock(resource_1);
		sleep(6);

	};

}

int main( void )
{

	stack_t f_stack[0x0050];
	stack_t gray_stack[0x0050];

	hardware_init();
	system_timer_config ();

	kernel_init ();

	load( f, 0, f_stack, 0x50, WDTIS0 );
	load( mcs_gray_f, 0, gray_stack, 0x50, WDTIS0 );
```

```
	run();

	return 0;

}
```

Se incluye ahora el código utilizado por la aplicación “mcs_gray_f”, que pone los datos en el buffer:

```
#include "gray.h"
#include "hardware.h"
#include "ukernel.h"
#include "mcs_gray.h"

volatile unsigned int cont;

void mcs_gray_f( void* parameters )
{
	for(;;){
		cont = cont +1;
		int dato = cont;
		write(resource_1, &dato, 2,0,0);
		sleep(7);
	};
}
```

Ambas aplicaciones se incluyen en la carpeta “Prueba_kernel+2app” incluida entre los archivos entregados.

4. INYECCIÓN DE FALLAS

Introducción

Las nuevas tecnologías, el descenso en las tensiones de alimentación y al aumento en la frecuencia y complejidad de los circuitos han llevado a la aparición cada vez más frecuente de single event effects (SEEs). [13] Se trata de la alteración temporal en el nivel de un valor lógico. Pueden ser del tipo single event upsets (SEUs) o single event transients (SETs). Los primeros son cambios de estado causados por carga libre de ionización cerca de algún nodo importante de un elemento lógico. Pueden ocurrir en cualquier FLIP FLOP, en cualquier período de reloj y durar cierta cantidad de ciclos. Los SETs son algo similar pero ocurren de forma transitoria, incorporando como variable el lapso durante el cual se da la anomalía. Pueden aparecer en cualquier compuerta, en cualquier instante y durar un pulso aleatorio, lo cual implica que el espacio de posibles SETs sea infinito, además de que se hace necesario considerar retardos.

Para nuestro proyecto en principio descartaremos los SETs y por temas de practicidad nos centraremos en los SEUs. Cabe señalar que si un SET es latcheado en un FLIP FLOP puede transformarse en un SEU, pero en general la mayoría de los SETs no tienen efecto alguno en el funcionamiento del circuito. Por otro lado, es importante tener en cuenta que un SEU no daña la celda, por lo cual su efecto puede durar hasta que la misma sea escrita nuevamente. Otro tipo de fallas que consideraremos son las permanentes, a causa de las cuales el valor de un bit queda fijo en 0 o en 1.

Todas estas fallas pueden derivar en la aparición de malfuncionamientos, es decir, modos de operación en los que el servicio prestado por el sistema se aparte del correcto. [14] Es de interés conocer entonces el tipo de fallas que pueden ocurrir y evaluar el comportamiento frente a las mismas. Durante la fase de desarrollo esto será de utilidad para mejorar los mecanismos de tolerancia a fallas, mientas que en la etapa de producción servirá para verificar se hayan alcanzado ciertos niveles de confiabilidad. Una manera de mitigar los efectos de esta clase de fenómenos es realizar campañas de inyección de fallas, evaluando el impacto de las mismas en el funcionamiento del sistema y corrigiendo el diseño a fin de robustecerlo. Se llama inyección de fallas al proceso de insertar fallas deliberadamente en un circuito bajo prueba o circuit under test (CUT). Es decir, analizar el comportamiento de un modelo del sistema en presencia de fallas provocadas artificialmente. La ventaja de hacerlo intencionalmente es que se reduce considerablemente la duración de los experimentos.

Hay diferentes métodos de inyección de fallas. En principio el criterio más general para diferenciarlos es el objeto de análisis, que puede ser el circuito manufacturado o el diseño. Para el primer caso, la fuente de perturbación puede ser física (por radiación o a nivel de pines por ejemplo) o lógica (JTAG, on-chip debugger o por reconfiguración). En el segundo caso se usa un modelo de las posibles fallas sumado a herramientas de diseño y prototipo, realizando simulaciones o emulando el sistema en un FPGA.

Inyección basada en Simulación

Para emplear esta técnica es necesario disponer de un modelo del sistema bajo prueba que permita simular el comportamiento del mismo en presencia de fallas. Se utilizan en general modelos en VHDL o Verilog. Las principales ventajas de esta variante son el bajo costo y la posibilidad de observar y/o alterar los valores lógicos con suma facilidad en cualquier momento. Como contrapartida, el tiempo de simulación es muy prolongado para cualquier

circuito de tamaño razonable y el desarrollo de los modelos a menudo requiere un esfuerzo considerable. Por otra parte, la inyección de fallas se realiza sobre la ejecución simulada de un modelo del sistema y no sobre un ejemplar del sistema final, por lo que no se podrán activar fallas que se introduzcan o manifiesten en las etapas posteriores del desarrollo.

Inyección basada en Emulación

Si bien la inyección basada en simulación es muy flexible a menudo resulta demasiado lenta. En cambio, al emular el circuito en un FPGA el proceso se hace mucho más rápido dado que el sistema puede correr casi a su velocidad real y tiene la ventaja además de soportar la interacción con sensores u otros dispositivos físicos del medio.

Es por esto que si el modelo que se dispone es sintetizable, es recomendable la inyección de fallas por esta vía, es decir, sobre un prototipo hardware. Pero para ello es necesario no sólo agregar mecanismos para observar y perturbar las señales internas sino también para controlar el instante en que se inyecta la falla. Por lo tanto, las modificaciones que deben introducirse al circuito son significativas y a menudo específicas para cada sistema.

Dentro de la inyección de fallas basada en emulación, se puede optar por la reconfiguración o la instrumentación. En el primer caso lo que se hace es correr el sistema hasta el momento de la falla y allí modificar el conexionado del circuito para luego continuar con la ejecución. En el segundo escenario se altera el circuito, agregando lógica que permita la inyección de fallas, en general enmascarando los bits de interés. El siguiente esquema ilustra a grandes rasgos esta idea:

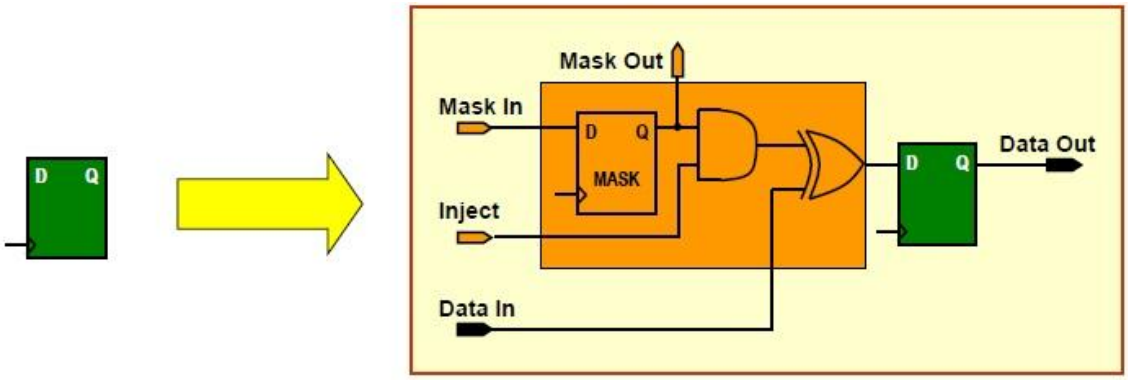


Figura 8: Esquema de la lógica para inyección basada en instrumentación [15]

Analizando el circuito es posible ver que cuando a la salida de la compuerta AND hay un 1 el xOR se encargará de invertir el valor de Data In, inyectando así una falla. Para que ello ocurra deben valer 1 simultáneamente las señales Inject y Mask In. El mecanismo está pensado para SEUs en registros pero puede aplicarse a buses de datos o direcciones, y con pequeñas modificaciones permite generar fallas de tipo permanente a 0 o a 1. Algo de este estilo es lo que se aplicará en nuestro proyecto.

[15] Figura extraída de “Fast Fault Injection Techniques using FPGAs”, Luis Entrena, 2013.

Modelo FARM

En procura de sistematizar la inyección de fallas resulta útil la utilización de un estándar en base al cual especificar las variables involucradas en este proceso. En ese sentido, el modelo FARM es una forma de caracterizar la inyección de fallas, definida a comienzos de 1990, en el cual se consideran dos conjuntos de entradas: la lista de fallas a inyectar “F” (faults) y la forma de activarlas “A” (activation), así como dos conjuntos de salidas: las lecturas que han de registrarse “R” (readouts) y las correspondientes medidas “M” (measures). [14]

Cabe señalar que *F* es un subconjunto dentro del espacio de todas las posibles fallas que se hayan considerado para el modelo del sistema bajo estudio. *A*, por su parte, constituye un set de parámetros de entrada del experimento, que pueden definirse en excitación y estado inicial. La acción conjunta *F* y *A* derivará en una respuesta del sistema definida en base a la trayectoria de su estado y salidas, la cual conforma el conjunto *R*. En base a este último, en *M* se resume en qué medida las fallas inyectadas derivaron o no en malfuncionamientos del sistema.

5. PERIFÉRICOS INCORPORADOS

Introducción

Sobre la base del openMSP430 donde corre el sistema "µkernel + app" se trabajó para incorporar algunos periféricos adicionales. Esto tiene por un lado el cometido de lograr un funcionamiento más similar al del kernel original del ANTEL-SAT, y por otro agregar las funcionalidades que permitan la inyección de fallas y registro de resultados.

En primera instancia se diseñó un módulo que permite inyectar fallas en memoria. Por otro lado, la necesidad de capturar los instantes en que ocurren ciertos eventos de interés derivó en la implementación de un timer adicional. Asimismo, para almacenar información relativa al flujo del programa se creó un banco de registros donde se guarda la cantidad de veces que la ejecución toma ciertos caminos en particular. Finalmente, para la última etapa se incluyó un módulo que realiza comprobación de redundancia cíclica (CRC).

A continuación se presentan los diferentes periféricos incorporados al openMSP430.

Módulo para inyección de fallas

Introducción

Para realizar la inyección de fallas se diseñó un periférico adicional que se conecta al openMSP430. El mismo permite inyectar fallas permanentes del tipo "stuck at 0" o "stuck at 1" ya sea en cualquier bit de una palabra de memoria o en los buses de datos o direcciones. Para el caso de palabras de memoria también existe la opción de insertar SEUs. Asimismo, el periférico fue instanciado en dos oportunidades, para actuar en la memoria de datos y en la de programa.

El módulo se implementó tomando como punto de partida una base de periférico diseñada por Olivier Girard, [15] que maneja algunos puertos estándar de entrada y salida y provee un banco de cuatro registros para darle instrucciones al módulo. A dicho bloque se le incorporaron funcionalidades para permitir el comportamiento deseado, generando máscaras para conectar a los buses de datos o direcciones de memoria. Para lograr que dichas máscaras actúen de la forma deseada se agrega cierta lógica de interconexión.

Para los casos de fallas permanentes, el bloque actúa de forma puramente combinatoria, en tanto que para el caso de SEUs fue necesario diseñar dos máquinas de estados. Una de ellas implementa el mecanismo para que el efecto del módulo se inhiba una vez realizada una escritura posterior en la celda de memoria afectada. La otra es simplemente un contador que permite elegir el instante en el que se inyectará la falla.

Interconexión

El diseño tiene la virtud de que, ya sea para datos o para direcciones, se genera una única máscara que vale para todos los tipos de fallas que se inyectarán. La misma tiene en 1 los bits que se harán fallar y en 0 los restantes. Luego se realiza una operación lógica entre la señal y la máscara para obtener el resultado deseado según el tipo de falla. Finalmente, un multiplexor elige la salida correspondiente según cómo hayan sido seteados los registros del módulo. El conexionado se ilustra en el siguiente esquemático.

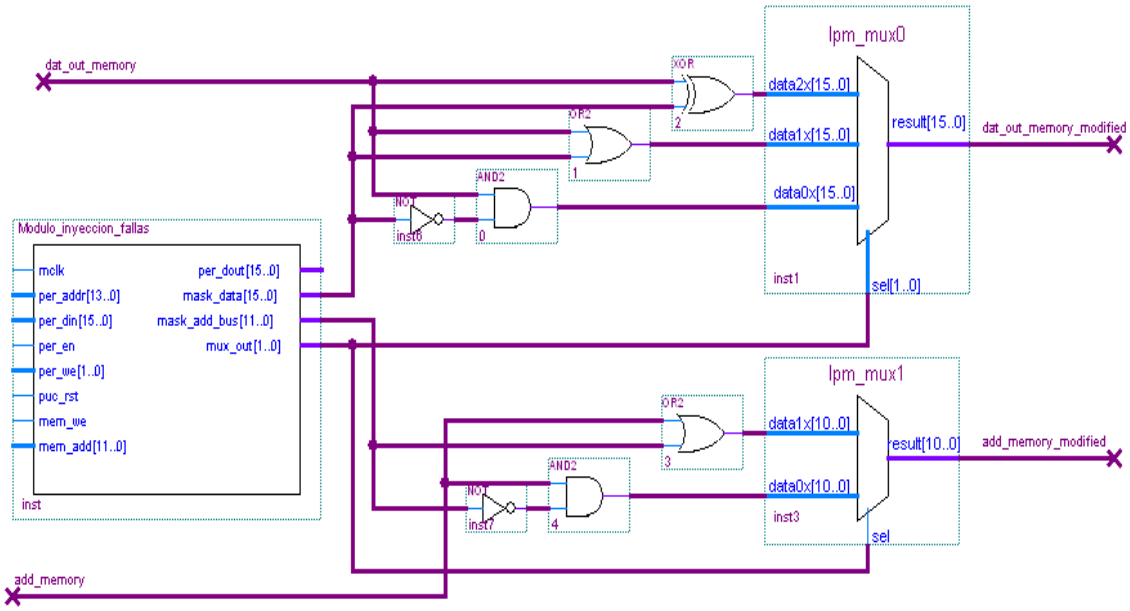


Figura 9: Esquemático de interconexión del módulo para inyección de fallas

Tabla de verdad

La siguiente tabla de verdad ilustra cómo opera el módulo según el tipo de falla a inyectar para un bit dado.

Tabla 3: Tabla de verdad de los diferentes tipos de fallas

	Entrada	**M**áscara	**S**alida
SEUs (S = E xOR M)	1	1	0
	0	1	1
	1	0	1
	0	0	0
Permanente a 1 (S = E OR M)	1	1	1
	0	1	1
	1	0	1
	0	0	0
Permanente a 0 (S = E AND $\overline{M}$)	1	1	0
	0	1	0
	1	0	1
	0	0	0

Puertos

El periférico tiene una serie de entradas y salidas genéricas heredadas del diseño base elaborado por Olivier Girard, y algunas adicionales propias de la funcionalidad específica que se le agregó para la inyección de fallas. La siguiente tabla resume los puertos del módulo.

Tabla 4: Puertos de E/S del módulo de fallas

	Entradas	**Salidas**
Heredadas del periférico base	mclk	per_dout
	per_adress	
	per_din	
	per_en	
	per_we	
	puc_rst	
Específicas	mem_we	mask_data [15:0]
	mem_add[10:0]	mask_address_bus[10:0]
		mux_out[1:0]

Registros

El módulo cuenta con cuatro registros de control que deben setearse para configurar el módulo. Dependiendo del modo de operación, puede no ser necesario inicializar todos. A continuación se detalla el cometido de cada registro.

Tabla 5: Registros del Módulo para inyección de fallas

REGISTRO	**DIRECCIÓN**	**ESTADO INICIAL**
CTRL1_D	0x0190	Reset manual
CTRL2_D	0x0192	Reset manual
CTRL3_D	0x0194	Reset manual
CTRL4_D	0x0196	Reset manual
CTRL1_P	0x01F0	Reset manual
CTRL2_P	0x01F2	Reset manual
CTRL3_P	0x01F4	Reset manual
CTRL4_P	0x01F6	Reset manual

Tabla 6: Registro CTRL1 del módulo de fallas – CONTROL

7	6	5	4	3	2	1	0
		LUGAR_FALLA			PERM	MODO_FALLA	

BIT	CAMPO	TIPO	DESCRIPCIÓN
15	ON_OFF	RW	Se activa inyección de fallas si está en 1
11-8	BIT_FALLA	RW	Bit de la palabra (o bus) que se hará fallar.
5-4	LUGAR_FALLA	RW	Define dónde se inyectará la falla: 00b = Bus de datos (de salida) de la memoria 01b = Bus de direcciones de la memoria 10b = Dirección (palabra) específica de la memoria
2	PERM	RW	Para fallas permanentes, define si es a 0 o a 1: 0b = Falla permanente a 0 1b = Falla permanente a 1
1-0	MODO_FALLA	RW	Modo de operación de la inyección de fallas: 00b = Permanente en un bit 01b = Permanente con máscara personalizada 10b = SEU en un bit 11b = SEU con máscara personalizada

Tabla 7: Registro CTRL2 del módulo de fallas - DIR_FALLA

15	14	13	12	11	10	9	8
					DIR_FALLA [10..8]		

7	6	5	4	3	2	1	0
DIR_FALLA [7..0]							

BIT	CAMPO	TIPO	DESCRIPCIÓN
10-0	DIR_FALLA	RW	Dirección de memoria en la cual se inyecta la falla

Tabla 8: Registro CTRL3 del módulo de fallas - MÁSCARA

15	14	13	12	11	10	9	8
MASCARA [15..8]							

7	6	5	4	3	2	1	0
MASCARA [7..0]							

BIT	CAMPO	TIPO	DESCRIPCIÓN
15-0	MASCARA	RW	Máscara personalizada para inyectar falla

Tabla 9: Registro CTRL4 del módulo de fallas - TIEMPO (para el caso de SEUs)

15	14	13	12	11	10	9	8
TIEMPO [15..8]							

7	6	5	4	3	2	1	0
TIEMPO [7..0]							

BIT	CAMPO	TIPO	DESCRIPCIÓN
15-0	TIEMPO	RW	Instante (en períodos de reloj) en el que se inyecta la falla.

Diagrama de estados

Cuando las fallas son de tipo permanente, las mismas estarán activas durante toda ejecución, por lo cual el bloque actúa de forma puramente combinatoria. Sin embargo, para el caso de SEUs hay que tener en cuenta dos elementos adicionales. Por un lado, la falla debe inyectarse en un momento dado, siendo dicho instante una variable más a tener en cuenta y que debe poder elegirse. Esto se logra incluyendo un contador e indicando el tiempo mediante el registro correspondiente (CTRL4). Por otro lado, como los SEUs no son destructivos ni permanentes, una posterior escritura a la dirección donde ocurrió la falla debe dejarla sin efecto, léase, el módulo debe dejar de actuar. Para ello se implementó una máquina de estados cuyo funcionamiento se ilustra en el siguiente diagrama.

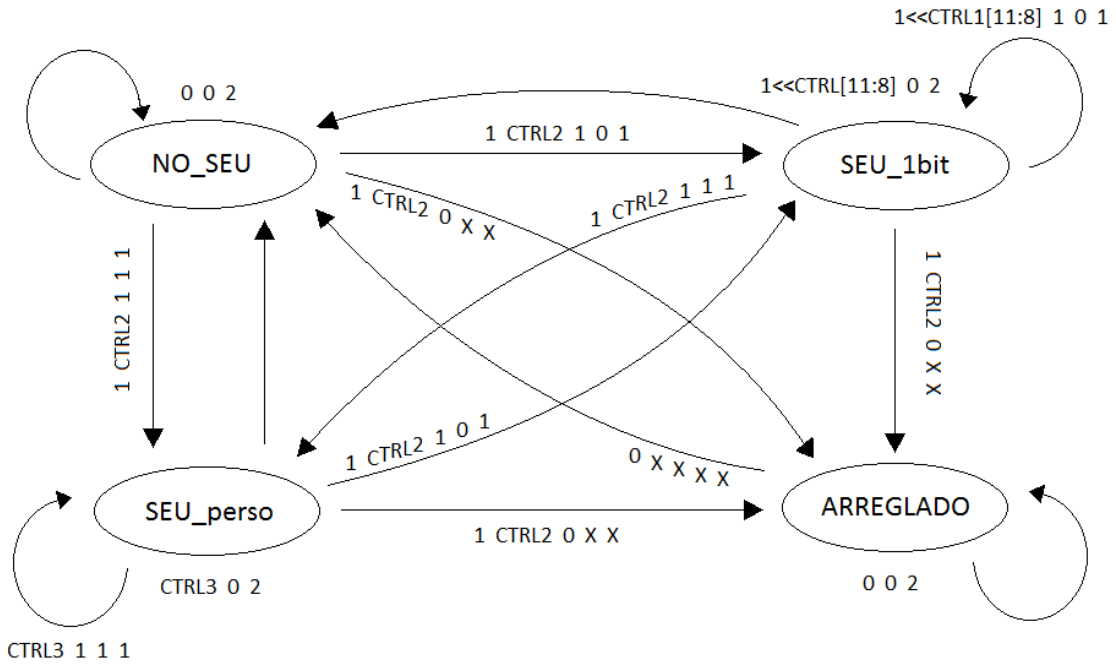


Figura 10: Diagrama de estados para fallas de tipo SEU

Entradas:
CRTL1[15] mem_add mem_we CRTL1[0] CRTL1[1]

Cabe señalar que mem_we es activa por nivel bajo, es decir, un 1 hace referencia a una escritura, mientras que un 0 implica una lectura.

Salidas:
mask_data mask_address_bus mux_out

La salida mask_address_bus está siempre en 0 (0x0000 más precisamente) porque el blanco de los SEUs son direcciones de memoria específicas y no buses. Asimismo, mux_out se mantiene en 2 porque es el valor correspondiente a SEUs (0 y 1 es son para permanentes a 0 y a 1 respectivamente). Por otro lado, en los estados NO_SEU y ARREGLADO mask_data vale 0 (0x0000) de modo que no se inyectan fallas.

Para el caso de SEU_1bt, la máscara se genera colocando un 1 en la posición indicada por CTRL[11:8], mientras que en SEU_perso se da como salida el valor del registro CTRL3. Es fácil ver que en realidad SEU_1bit es un caso particular de SEU_perso, si se escribe la máscara adecuada en CTRL3. El usuario puede optar por cualquiera de las dos modalidades, aunque la primera permite ahorrar una escritura al microprocesador.

Se deben setear antes que nada los registros CTR2, CTRL3 y CTRL4. Posteriormente, al escribir en el registro de control un valor tal que CTRL[1]=1, el contador se iniciará. El sistema permanecerá en reposo (estado NO_SEU) hasta que la cuenta alcance el valor seteado en CTRL4. Recién en ese momento la falla quedará lista para activarse en caso de ocurrir una lectura a la dirección fijada en CRTL2.

Vale aclarar que en cada estado, la transición para la cual no se indican las entradas corresponde al caso en que no se da ninguna de las otras condiciones de salto, es decir, el escenario del “else”. Sólo se considerarán las fallas de a una por vez, por lo cual luego de pasar por el estado ARREGLADO el sistema sólo puede retornar al reposo, es decir, a NO_SEU.

Finalmente, cabe señalar que máquina de estados no opera por flancos de un sentido en particular sino que responde a cualquier cambio en la señal de reloj, es decir que actúa "medio período de reloj tarde” pero a tiempo como para provocar el cambio en los datos mediante la máscara correspondiente.

Simulaciones

Fallas permanentes

A continuación se incluye una simulación para inyección de fallas permanentes a 0, tanto en el bus de datos como en palabras específicas de memoria. El caso de permanentes a 1 es análogo, ya que las máscaras generadas son las mismas, siendo la lógica de interconexión la que se encarga de lograr el efecto buscado.

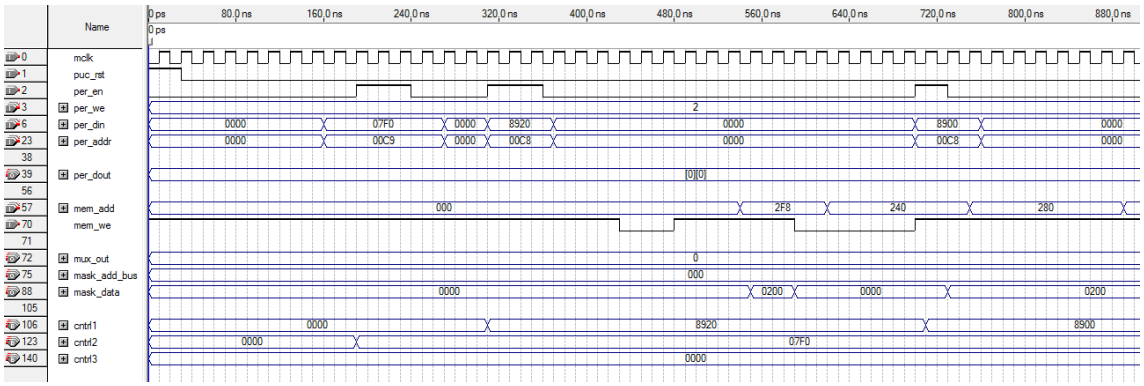


Figura 11: Simulación de una falla permanente a 0 en memoria de datos

Se comienza configurando los registros de control con los siguientes valores:

- CRTL2 = 7F0 (Dirección de la falla)
- CTRL1 = 8920 (Falla permanente en el bit 9 de la palabra de memoria elegida)

No corresponde en este caso setear el registro CTRL3, dado que la máscara ya queda determinada en el CTRL1, ni el CTRL4, de uso exclusivo para SEUs.

Luego vemos que mask_dat se activa sólo para el caso de lecturas a la dirección de la falla. Cabe aclarar que debido a los offsets y la forma de manejar el direccionamiento del openMSP430, para acceder a la 7F0h se debe leer la 2F8h = (7F0h – 200h) / 2. Esto fue constatado analizando el código y empleando el analizador lógico. Posteriormente se cambia el valor del CTRL1, pasando a ser 8900 (Falla permanente en el bit 9 del bus de datos), comprobando que la máscara se genera para lecturas en cualquier dirección, es decir, afecta directamente al bus y no a una palabra de memoria en particular.

SEUs

A continuación se muestra un diagrama de tiempos para el caso de fallas de tipo SEU en un bit.

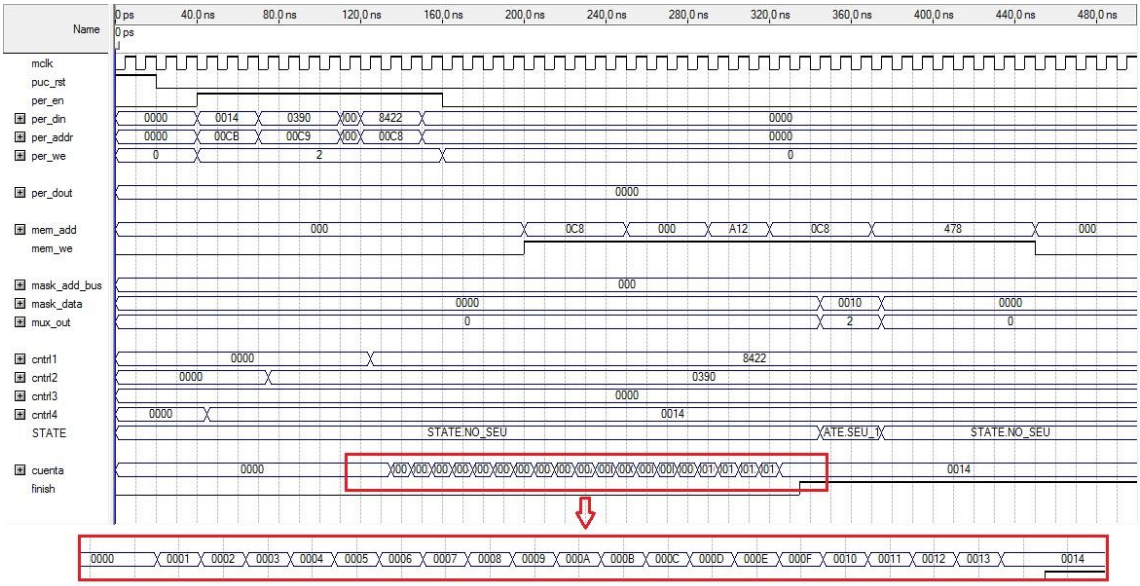


Figura 12: Simulación de un SEU en memoria de datos (parte 1 de 2)

En primer lugar se escriben los registros de control con los siguientes valores:

- CTRL4 = 14 (Instante de la falla en períodos de reloj, 14h = 20d)
- CRTL2 = 390 (Dirección de la falla)
- CTRL1 = 8422 (SEU en el bit 4 de una palabra de memoria)

No corresponde en este caso setear el registro CTRL3, dado que la máscara ya queda determinada en el CTRL1.

Inicialmente el módulo se encuentra en el estado NO_SEU. Una vez que se realiza la escritura al CTRL1, se activa el contador y al llegar a 14h, el sistema queda preparado para pasar a SEU_1bit cuando se realice una lectura a la dirección establecida por CTRL2. En ese momento se dan las salidas mask_data = 0010h (equivalente a 0000000000001000b, o sea bit 4) y mux_out = 2h (falla en palabra de memoria).

Se recuerda que para acceder a la dirección 390h se debe leer la C8h = (390h – 200h) / 2.

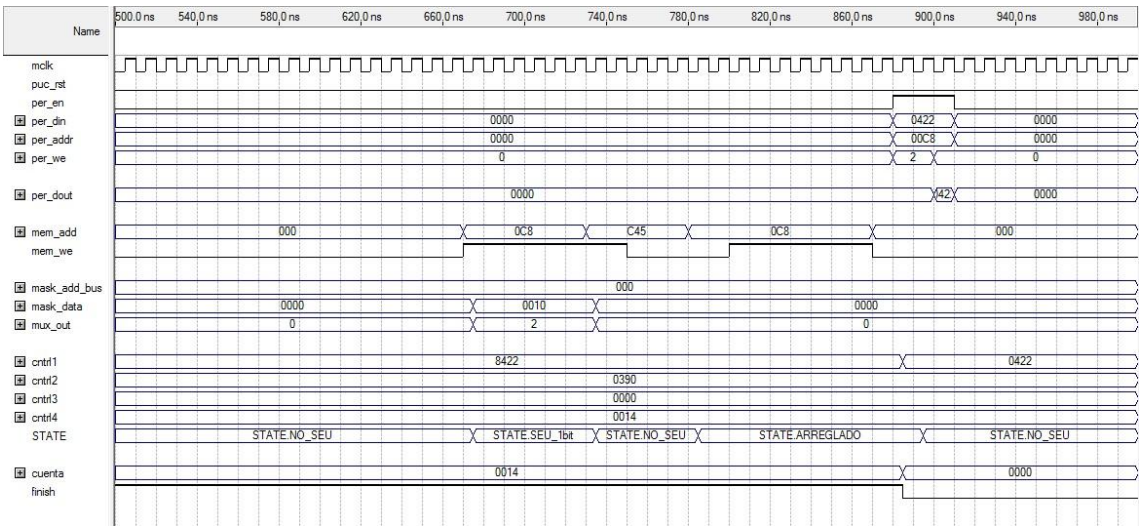


Figura 13: Simulación de un SEU en memoria de datos (parte 2 de 2)

Si ocurre una escritura en la dirección de la falla se pasa al estado ARREGLADO, en el cual se inhibe la acción del módulo, permaneciendo allí hasta que una nueva configuración del CTRL1 lo lleve al estado NO_SEU.

Módulo CRC

Introducción

Los mecanismos de redundancia que permiten realizar chequeo de consistencia de stack o validar una copia de código cuando se realiza triplicado del mismo en memoria se basan en la comprobación de redundancia cíclica (CRC). Se trata de un código de detección de errores de uso frecuente en redes digitales y en dispositivos de almacenamiento para detectar alteraciones en los datos. El cálculo consiste en una división en la cual se descarta el cociente, tomando el resto como resultado. [16] Su tamaño queda determinado entonces por la longitud del divisor, existiendo algunos polinomios generadores típicos que derivan en respectivos estándares de CRC. Entre ellos se encuentran por ejemplo el CRC-12, CRC-16 o CRC-CCITT, siendo este último el implementado por el módulo. El polinomio generador es en este caso: $f(x) = x^{16} + x^{12} + x^5 + 1$.

Implementación

El módulo CRC es un periférico encargado de implementar el algoritmo de comprobación de redundancia cíclica, el cual si bien existe en las familias 5 y 6 de MSP430 no fue implementado en la familia 1 y mucho menos en el openMSP430.

Para la implementación del módulo CRC se utilizó como en los otros casos el template de Olivier Girard, [15] ya que todos los periféricos, más allá de la familia específica del microcontrolador, utilizan un banco de registros para su configuración. A esa base se le incorporó un bloque de cálculo de CRC obtenido a partir de un generador de código [17]. A partir de cierta entrada y estado de inicialización se obtiene el código CRC asociado, de acuerdo al siguiente diagrama:

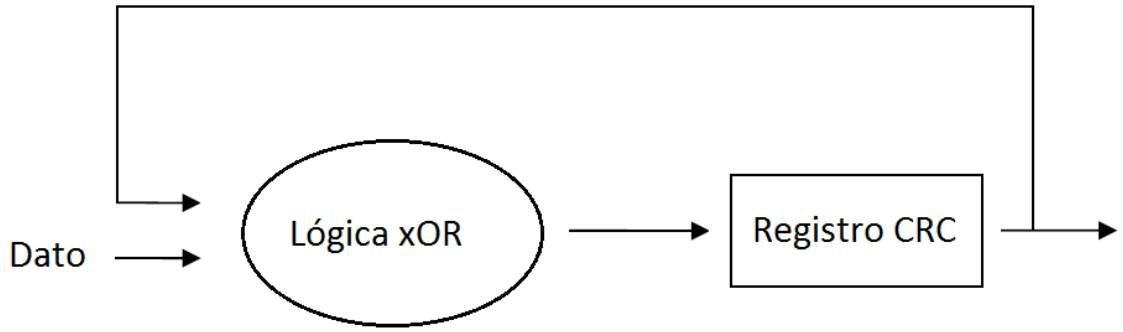


Figura 14: Esquema de funcionamiento del algoritmo "Ultimate CRC"

Las diferencias del bloque CRC generado con el programa y el deseado son básicamente las siguientes:

- El estado de inicialización es fijo e inmodificable.
- El ancho de los datos de entrada es fijo.
- No implementa la posibilidad de configuración y manejo mediante banco de registros.

Dada estas diferencias se debió adaptar la lógica combinatoria implementada en el bloque generado para que funcione en base a un banco de registros. El módulo es visto desde afuera tal como se presenta en su versión original de hardware implementada en las familias 5 y 6 del MSP430. Asimismo, se generó con el programa el código que implementa un módulo CRC de 16 bits y otro de 8 bits, pudiendo configurar la opción deseada mediante el banco de registros. A continuación se pasan las líneas ingresadas en la consola de comandos para obtener los mencionados códigos:

```
>crc-gen verilog 16 16 1021>CRC16bits_1021.txt
>crc-gen verilog 8 16 1021>CRC8bits_1021.txt
```

Registros

El módulo cuenta un registro para el dato de entrada, otro para setear el valor de inicialización y donde luego se podrá leer el resultado y finalmente una entrada adicional para cadenas de 8 bits. A continuación se detalla cada uno de ellos.

Tabla 10: Registros del Módulo CRC

REGISTRO	DIRECCIÓN	ESTADO INICIAL
CRCDI	0x0140	Reset con PUC
CRCINIRES	0x0142	Reset con PUC
CRCDI_L	0x0144	Reset con PUC

Tabla 11: Registro CRCDI del módulo CRC

15	14	13	12	11	10	9	8
CRCDI [15..8]							

7	6	5	4	3	2	1	0
CRCDI [7..0]							

BIT	CAMPO	TIPO	DESCRIPCIÓN
15-0	CRCDI	RW	Dato de entrada. Este valor es usado para calcular el código CRC, que es guardado en el registro CRCINIRES.

Tabla 12: Registro CRCINIRES del módulo CRC

15	14	13	12	11	10	9	8
CRCINIRES [15..8]							

7	6	5	4	3	2	1	0
CRCINIRES [7..0]							

BIT	CAMPO	TIPO	DESCRIPCIÓN
15-0	CRCINIRES	RW	Inicialización y resultado. Este registro guarda el resultado actual del cálculo de CRC. Permite además setear el valor de inicialización.

Tabla 13: Registro CRCDI_L del módulo CRC

7	6	5	4	3	2	1	0
CRCDI_L [7..0]							

BIT	CAMPO	TIPO	DESCRIPCIÓN
7-0	CRCDI_L	RW	Dato de entrada. Recibe un dato de 8 bits y calcula el código CRC, el cual es guardado en el registro CRCINIRES.

Simulaciones

A continuación se pasa a mostrar la simulación completa del módulo que realiza el cálculo de CRC en la modalidad de 16 bits. Se destaca que el módulo atiende al registro CRCINIRES en la dirección 0x0142 y al CRCDI_L en la dirección 0x0144, pero en el bus de direcciones se observa 0x00A1 (0x0142/2) y 0x00A2 (0x0144/2) respectivamente.

En primera instancia se procede a la inicialización del registro CRCINIRES con el valor 0xFFFF y luego se pasan sucesivamente 8 valores al registro CRCDI_L (0xFF, 0xED, 0xAA, 0x3D, 0x32, 0x11, 0x F1 y 0xAC). Por último se procede a la lectura del valor del registro CRCINIRES, en el cual se despliega el resultado del cálculo de CRC (para esta serie de valores a partir de la semilla dada). El valor obtenido es 0xA96A, como indica con la lectura realizada al final.

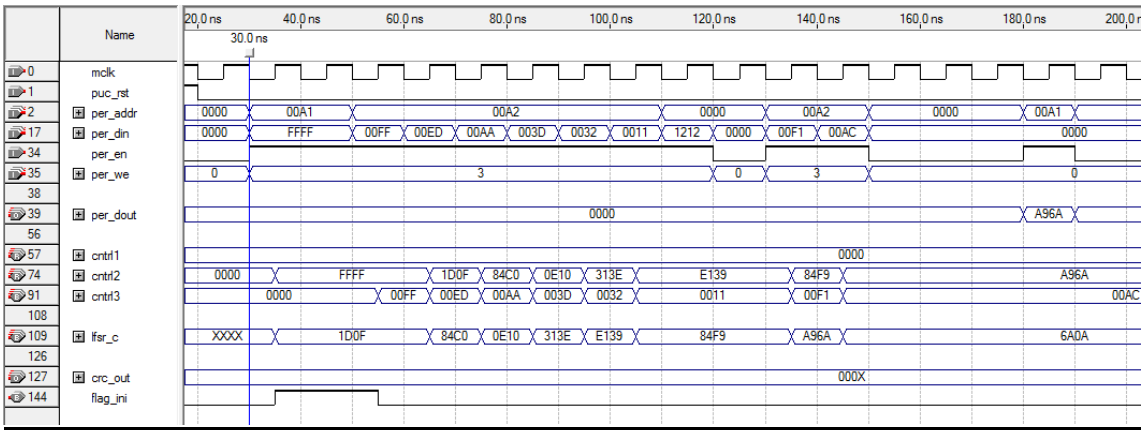


Figura 15: Simulación del módulo CRC (parte 1 de 3)

Luego se repite lo anterior, con la variante de escribir un valor cualquiera (se eligió el 0x0001) en el registro CRCDI_INRES antes de setearlo nuevamente con el valor 0xFFFF.

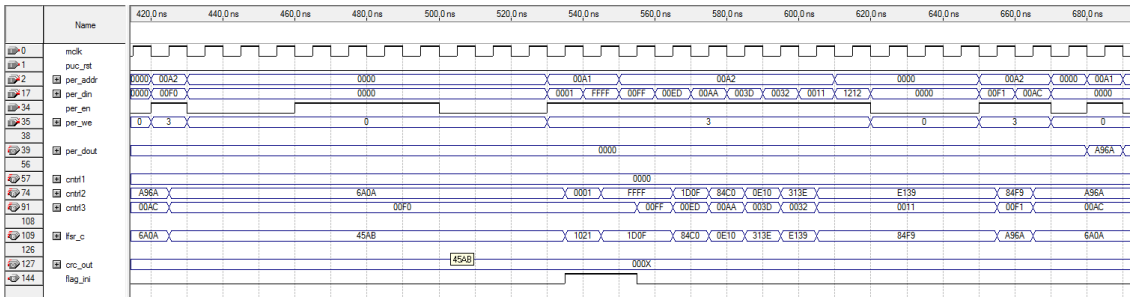


Figura 16: Simulación del módulo CRC (parte 2 de 3)

Se observa que esto no altera el cálculo, obteniendo el mismo resultado que en el caso anterior (0xA96A). Se comprueba que cadenas idénticas de entrada generan igual código CRC a partir de la misma semilla. Por último, se muestra que inicializando el registro CRCINIRES con un valor diferente al que se venía trabajando, el resultado varía más allá que la cadena de entrada siga siendo la misma.

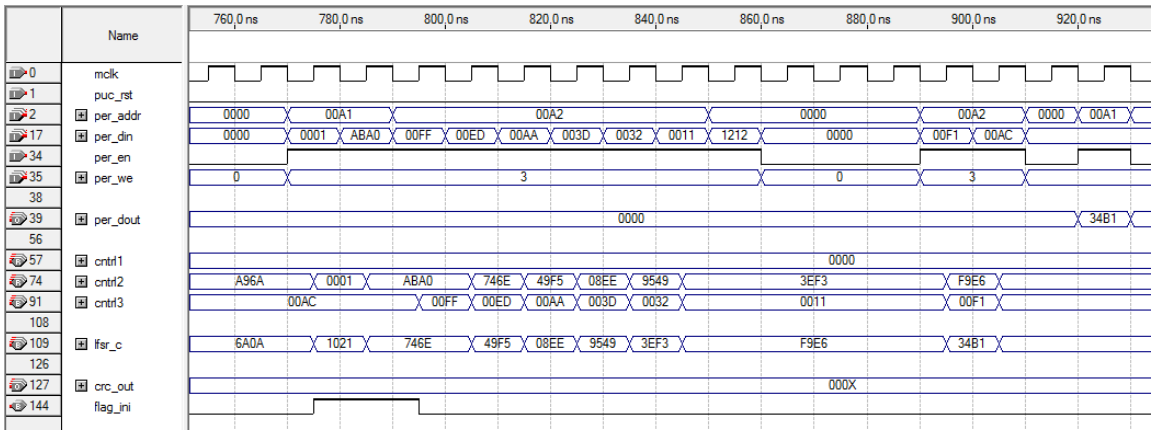


Figura 17: Simulación del módulo CRC (parte 3 de 3)

Se aprecia que al sustituir el valor de la semilla por el 0xABA0 el resultado obtenido es 0x34B1, el cual difiere de 0xA96A, como era de esperar. Finalmente, cabe señalar que si bien las simulaciones se realizaron para entradas de 16 bits, las pruebas son análogas para el caso de 8 bits.

Si bien las pruebas realizadas no verifican que el valor calculado sea correcto de acuerdo al algoritmo CRC-CCITT, sí confirman que el módulo es capaz de detectar variaciones de un bit en cadenas de largo variable, lo cual es suficiente para el uso que se le dará en nuestro trabajo.

Prueba del módulo interconectado

Para incorporar el periférico al resto del diseño, se sigue la guía disponible en el *Anexo 8: Procedimiento para agregar un periférico*. Para verificar el correcto funcionamiento del módulo (ya conectado al openMSP430) se programó un script de prueba. En el mismo se inicializa el registro CRCINIRES con un valor arbitrario x_1 y luego mediante un loop se van cargando sucesivamente las entradas de un vector de largo 100, que llamaremos v_1. Se registra el resultado obtenido CRC_1 y luego se repite el procedimiento para otra pareja (x_2,v_2) arrojando el código CRC_2. Finalmente, se realiza el cálculo una tercera vez, en este caso con los valores x_3=x_1 y v_3=v_1, verificando que CRC_3 = CRC_1. Se comprueba entonces que una secuencia dada de entradas genera el mismo CRC si se parte de la misma semilla.

Timer A1

Introducción

Debido a la necesidad de contar con un método que permita la captura de los instantes en que ocurren eventos de interés se tomó la decisión de implementar un timer que posea la capacidad de interrumpir al micro, pero que a su vez sea inmune a los resets del mismo. En una primera instancia se había estudiado la posibilidad de hacer uso del propio timer utilizado por el microcontrolador (el openMSP430) ya que ofrece la posibilidad de contar períodos de una fuente dada y desplegar el resultado en un registro. Esta opción fue rápidamente descartada dado que tras un PUC (reset del sistema) el timer se reinicia, perdiendo entonces el valor de la cuenta.

En las campañas de inyección de fallas será frecuente el hecho de que el microcontrolador se reinicie al estar sometido a situaciones anormales durante su funcionamiento. Dichos eventos ocurrirán en instantes que en principio no son predecibles, por lo cual resulta inviable guardar el valor de la cuenta en un instante previo a un reset. Si bien se podrían modificar ciertas rutinas de interrupción en procura de dicho objetivo, en todo momento se manejó como premisa ser lo menos invasivos posible hacia el código del ukernel.

El "timer A1" a incluir, además de ser inmune a los resets del sistema debe otorgar la posibilidad de finalizar el experimento mediante un time_out, es decir, generar un reset al haber expirado un tiempo arbitrario previamente seteado, dejando constancia de lo ocurrido. Durante una campaña puede ocurrir que una falla genere un punto en el software que no permita el avance del programa, provocando que se reinicie. Esto se puede dar tantas veces como dispuesto esté el usuario a esperar. Para evitar esto se utiliza el timer A1, el cual una vez expirado el tiempo considerado prudente para la ejecución normal del programa, provoca un reset para eliminar la falla y seguir adelante con otra.

Implementación

Dada las necesidades para solucionar el problema de administración de tiempos así como el control del mismo y teniendo en cuenta los medios disponibles se optó por "clonar" el timer original del openMSP430 pero otorgándole inmunidad ante resets del sistema, dejando la entrada puc_rst conectada a tierra. Para esto se creó una nueva instancia del módulo disponible, pero al momento de incorporarlo al resto de la arquitectura hardware se lo aisló de la señal PUC, la cual es transmitida a todos los periféricos del sistema cuando ocurre un reset. A continuación se despliega el código usado para agregar la instancia del nuevo timer.

```
omsp_timerA #(.TACTL('h2), .TAR('h4), .TACCTL0('h6), .TACCR0('h8),
.TACCTL1('h10), .TACCR1('h12), .TACCTL2('h14), .TACCR2('h16), .TAIV('h18) )
timerA_1 (

// OUTPUTs
   .irq_ta0      (irq_ta1_0),       // Timer A interrupt: TACCR0
   .irq_ta1      (irq_ta1_1),       // Timer A interrupt: TAIV, TACCR1,
TACCR2
   .per_dout     (per_dout_tA1),   // Peripheral data output
   .ta_out0      (ta_out0_a1),       // Timer A output 0
   .ta_out0_en   (ta_out0_en_a1),    // Timer A output 0 enable
   .ta_out1      (ta_out1_a1),       // Timer A output 1
   .ta_out1_en   (ta_out1_en_a1),    // Timer A output 1 enable
   .ta_out2      (ta_out2_a1),       // Timer A output 2
   .ta_out2_en   (ta_out2_en_a1),    // Timer A output 2 enable

// INPUTs
   .aclk_en      (aclk_en),       // ACLK enable (from CPU)
   .dbg_freeze   (dbg_freeze),    // Freeze Timer A counter
   .inclk        (),         // INCLK external timer clock (SLOW)
   .irq_ta0_acc  (irq_acc[7]),    // Interrupt request TACCR0 accepted
   .mclk         (mclk),          // Main system clock
   .per_addr     (per_addr),      // Peripheral address
```

```
    .per_din       (per_din),        // Peripheral data input
    .per_en        (per_en),         // Peripheral enable (high active)
    .per_we        (per_we),         // Peripheral write enable (high active)
    .puc_rst       (1'b0),        // Main system reset
puc_rst
    .smclk_en      (smclk_en),       // SMCLK enable (from CPU)
    .ta_cci0a      (ta_cci0a1),       // Timer A capture 0 input A
    .ta_cci0b      (ta_cci0b_a1),       // Timer A capture 0 input B
    .ta_cci1a      (ta_cci1a1),       // Timer A capture 1 input A
    .ta_cci1b      (1'b0),           // Timer A capture 1 input B
    .ta_cci2a      (ta_cci2a1),       // Timer A capture 2 input A
    .ta_cci2b      (1'b0),           // Timer A capture 2 input B
    .taclk         ()           // TACLK external timer clock (SLOW)
);
```

Para realizar la interconexión del nuevo timer sin alterar el funcionamiento del original se optó por usar la misma configuración de puertos del módulo existente, el cual utiliza algunos bits de entrada/salida de dos puertos diferentes. Lo que se hizo entonces para el periférico agregado fue hacer uso de los mismos bits pero de otros dos puertos que se encontraban libres.

Por lo tanto, al mantener la misma arquitectura hardware, la configuración del nuevo timer será prácticamente igual a la del original, a excepción de algunos detalles propios del uso que se le dará.

Registros

Tabla 14: Registros del Timer A1

REGISTRO	DIRECCIÓN	ESTADO INICIAL
TACTL	0x0102	Reset manual
TAR	0x0104	Reset manual
TACCTL0	0x0106	Reset manual
TACCR0	0x0108	Reset manual
TACCTL1	0x0110	Reset manual
TACCR1	0x0112	Reset manual
TACCTL2	0x0114	Reset manual
TACCR2	0x0116	Reset manual
TAIV	0x0118	Reset manual

Banco de registros

Motivación

Cada vez que se produce un reset en el openMSP430, el microprocesador transmite una señal de reset a todos sus periféricos, llevando los registros de todos los periféricos a un estado preestablecido. Con el fin de lograr un mejor manejo y comprensión de los estados por los que pasa el microcontrolador es necesario implementar algún mecanismo que permita el conteo

de los distintos resets sufridos por el sistema. Para ello se incorpora un banco de registros de 8 bits.

Como se mencionó, los registros se usan para llevar la cuenta de la cantidad de veces que el sistema se reinicia. Luego se los consulta para entregar un informe de cantidad de eventos registrados. Asimismo, son utilizados también en nuestro caso como offset para un puntero en memoria que almacena los instantes de tiempo. Es decir, se define un puntero en memoria y cada vez que se produce un reset se guarda el valor de tiempo en el lugar *(PDeTiempo + cantidadDeReset).

Utilización

Dado que el contenido de los registros del banco no se modifica con resets del microcontrolador, el mismo debe ser inicializado externamente antes de ser utilizado, por ejemplo desde GDB. A su vez, esta propiedad le otorga la capacidad de almacenar información que persista ante reinicios del sistema, y es allí donde se halla la esencia del módulo. Si bien la utilización del mismo se da en tiempo de ejecución del programa, podrá ser consultado en cualquier instante de tiempo en que se haga un break. Es así que el módulo cumple una función de gran interés en las campañas de fallas, ya que permite automatizar y agilizar la lectura de los distintos arreglos dedicados al almacenamiento de tiempos. Al momento de detenerse durante una campaña, el banco de registros permite saber qué arreglos y elementos asociados a reset se deben leer.

Implementación

La arquitectura del banco corresponde exactamente con la de los templates de Olivier Girard. Se utiliza el modelo de banco de registros de 8 bits dado que no es esperable contar más de 255 resets antes de que expire el timeout del timer A1. El procedimiento para incorporar un módulo se detalla en el *Anexo 8: Procedimiento para agregar un periférico*, con la particularidad de que la señal de reset es puesta a tierra para darle inmunidad con respecto a dicho evento, tal cual se comentó.

Registros

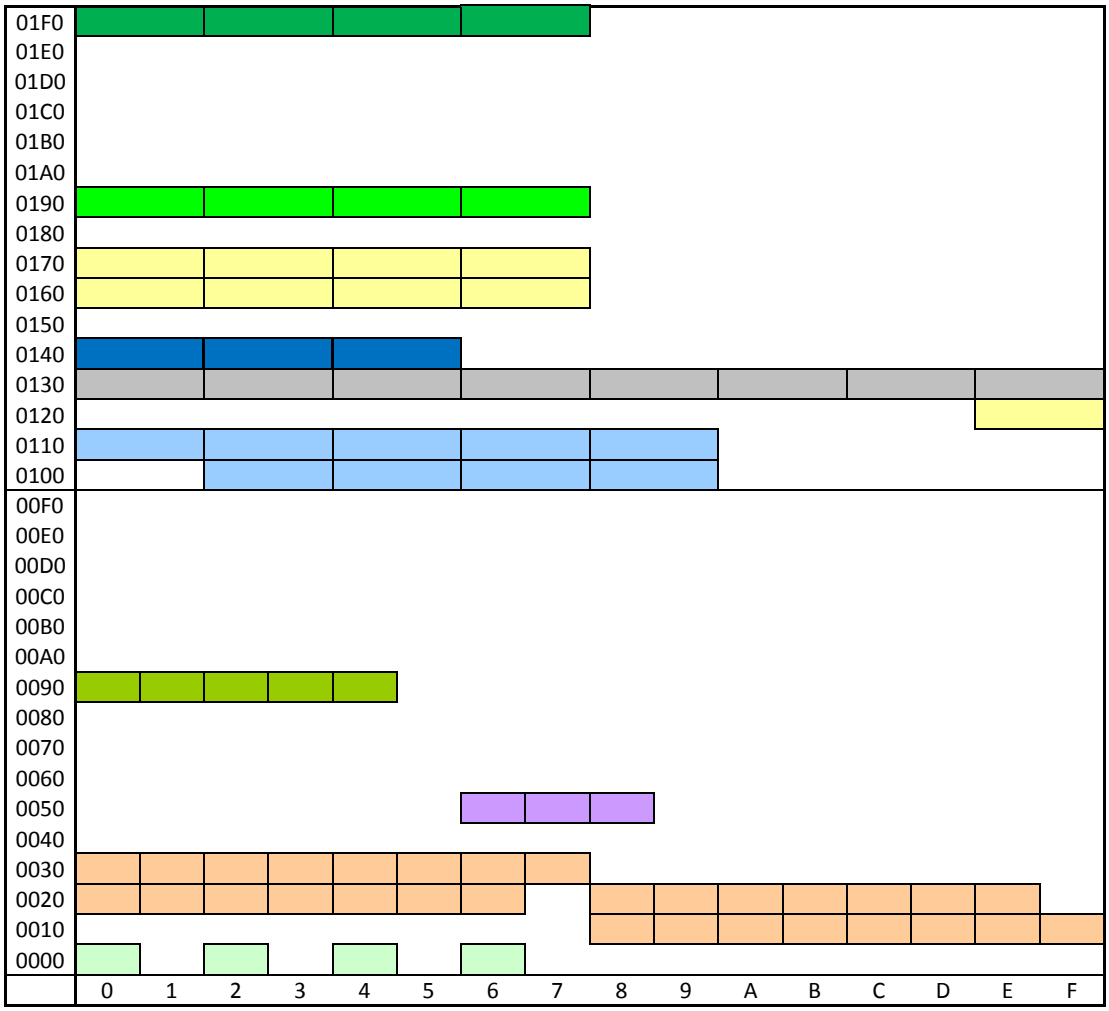
Tabla 15: Registros del Banco de registros

REGISTRO	DIRECCIÓN	ESTADO INICIAL
CNTRL1	0x0090	Reset manual
CNTRL2	0x0091	Reset manual
CNTRL3	0x0092	Reset manual
CNTRL4	0x0093	Reset manual
CNTRL5	0x0094	Reset manual

Mapeo en memoria

A continuación se presenta un mapa de memoria en el que se incluyen los módulos originales y los periféricos agregados. Cabe señalar que se dispone del rango entre las direcciones 0x0000 y 0x0200.

Tabla 16: Mapa de memoria de periféricos



Periféricos originales del openMSP430:

Hardware Multiplicador | Timer A | GPIOs | BclkM | SP Reg

Periféricos agregados:

Módulo_fallas memoria prog | Módulo_fallas memoria datos | Banco Reg | Módulo CRC | Timer A1

6. CAMPAÑAS DE INYECCIÓN DE FALLAS

Mecanismo para inyección y registro de fallas

En una primera instancia se probó exclusivamente el funcionamiento del módulo de inyección. Esto se hizo insertando fallas individualmente desde GDB, ingresando las instrucciones de forma manual mediante línea de comandos. Una vez seteados los registros de control, se realizaron sucesivas escrituras y lecturas para verificar que actuara correctamente, chequeando por ejemplo que:

- Las fallas se inyectan (sólo) en el bus o en la dirección de memoria indicada y únicamente en el o los bits especificados.
- El efecto de los SEUs desaparece luego de realizar una escritura en la dirección de memoria afectada previamente por la falla.
- Al deshabilitar el módulo de fallas el mismo efectivamente no actúa.

Luego de testear el módulo individuamente, se procedió a utilizarlo para inyectar fallas en memoria durante la ejecución de una aplicación sobre el µkernel, en este caso se trata de la *Aplicación sencilla* presentada anteriormente. Se generó entonces un script para automatizar este proceso y registrar los resultados. En este código se definen los vectores que se escribirán en los registros de control del módulo en las sucesivas corridas, se realiza la conexión con el GDB proxy, se especifica el archivo de trabajo y se setean breakpoints en lugares de interés para poder luego determinar el flujo que siguió el programa.

Posteriormente, se implementa un loop en el cual se van realizando las sucesivas corridas para cada tanda de valores a escribir en los registros de control. Lo que interesa en cada paso de la iteración es saber por ejemplo si la ejecución terminó en el tiempo esperado y con el resultado correcto, si se produjo algún reset y por cuál mecanismo (PUC, watchdog). Toda esta información es registrada en un archivo de texto que permite analizar los resultados a posteriori.

Primera campaña de inyección de fallas

Implementación de código en C

Para registrar el flujo del programa µkernel + App al inyectar una falla, se resolvió implementar un mecanismo en base a con registros que se encargan de guardar la cantidad de veces que el programa toma un camino en particular.

Se identificaron cinco caminos críticos que permiten evaluar el efecto de una falla en el programa:

1- FIN_APP: bandera que indica la finalización exitosa de la aplicación que corre en el ukernel.
2- WDT_RESET: se encarga de registrar la cantidad de resets ocurridos por vencimiento de la cuenta de watchdog. Los mismos se dan cuando la aplicación demora más del tiempo máximo estipulado para que la misma devuelva el uso de un recurso.
3- SOFT_RESET: lleva la cuenta del número de resets por software, es decir la cantidad de veces que se llama a la función “reset()” prevista para reiniciar el sistema ante inconsistencia u overflow de stack.
4- MEM_RESET: registra la cantidad de resets generados por otro mecanismo que no sea la cuenta de watchdog o la función “reset()”.

5- TOUT_RESET: bandera prevista para registrar que venció la cuenta del timer A1.

Las banderas descriptas anteriormente se implementaron como registros de 8 bits, los cuales fueron agregados a la lógica del openmsp430. A continuación se muestra la sección de código incorporado al módulo "omsp_system.h" con los nombres de referencia de los registros agregados y sus respectivas direcciones dentro del mapa de periféricos.

```
// REGISTROS DE EVENTOS DE FALLAS
#define  WDT_RESET       (*(volatile unsigned char  *) 0x0090)
#define  SOFT_RESET      (*(volatile unsigned char  *) 0x0091)
#define  TOUT_RESET      (*(volatile unsigned char  *) 0x0092)
#define  FIN_APP          (*(volatile unsigned char  *) 0x0093)
#define  MEM_RESET       (*(volatile unsigned char  *) 0x0094)
```

Se resolvió además registrar los instantes en que el programa pasa por cada uno de los caminos a modo de tener una idea del orden en el que se dan los eventos. Los mismos se almacenan en el área de memoria de datos entre las direcciones 0x400 y 0x600.

Los punteros utilizados se definieron en el módulo "common.h" del ukernel. Se incluye un fragmento de código que indica la forma en la que fueron definidos:

```
//punteros de arreglos para timeouts -SATELITEST
unsigned int volatile* wdt_ta1 ;
unsigned int volatile* app1_ta1 ;
unsigned int volatile* soft_ta1 ;
unsigned int volatile* mem_ta1 ;
```

Se fijó 4,5 segundos como límite para cada corrida, es decir que dicho valor es el que se elige como tiempo de timeout para el timer A1. Éste último se configura por única vez al inicio del programa y arranca a contar de manera sincronizada sin detenerse ni resetearse hasta finalizar la cuenta. Para acceder a los registros del timer A1 y hacer posible su configuración se agregó la siguiente sección de código en el módulo "omsp_system.h", donde se detalla el nombre de cada registro y dirección correspondiente.

```
// TIMER A1
#define  TA1CTL      (*(volatile unsigned int  *) 0x0102)
#define  TA1R        (*(volatile unsigned int  *) 0x0104)
#define  TA1CCTL0     (*(volatile unsigned int  *) 0x0106)
#define  TA1CCR0      (*(volatile unsigned int  *) 0x0108)
#define  TA1CCTL1     (*(volatile unsigned int  *) 0x0110)
#define  TA1CCR1      (*(volatile unsigned int  *) 0x0112)
#define  TA1CCTL2     (*(volatile unsigned int  *) 0x0114)
#define  TA1CCR2      (*(volatile unsigned int  *) 0x0116)
#define  TA1IV        (*(volatile unsigned int  *) 0x0118)
```

El mismo timer se utilizó para registrar los instantes en que se dan los eventos, o sea, cada vez que el programa pasa por los caminos críticos descriptos anteriormente. Además de la implementación e inclusión lógica del timer A1 y de cada registro utilizado para llevar la cuenta del flujo del programa, fue necesario agregar el código en C que hiciera posible registrar los tiempos de los eventos, incrementar las banderas e inicializar el timer A1.

A continuación se detalla el código para algunos de los módulos de ukernel.

Reset por watchdog

Para detectar los resets por watchdog se agregó código al inicio del main() dentro del módulo "mcs.c", donde se consulta el valor del primer bit del registro "IFG1", el cual indica si el PUC_RESET generado fue por causa de la cuenta de watchdog. Si se cumple IFG1=0x01 entonces se registra un reset por watchdog, incrementando la variable WDT_RESET y guardando el instante de tiempo correspondiente en el lugar de memoria indicado por el puntero definido para registrar los instantes de WDT_RESET.

El algoritmo utilizado para registrar los tiempos en memoria a través del código en C implementado es el siguiente:

I. Se incrementa el puntero correspondiente: "wdt_ta1 = wdt_ta1 + WDT_RESET", lo cual permite actualizar el puntero en memoria sin perder su valor entre los eventuales resets del programa. WDT_RESET es un registro que lleva la cuenta de la cantidad de eventos entre resets de programa.
II. Luego se guarda la cuenta del timer A1 en la dirección de memoria señalada por el puntero correspondiente.
III. Se incrementa el registro WDT_RESET, que lleva la cuenta de eventos.
IV. Para finalizar, en el caso de reset por watchdog se debe resetear la bandera "IFG1=0x00". Esto evitará que se le atribuya erróneamente al watchdog la responsabilidad de un eventual próximo reset que pudiera ocurrir por otra causa.

Reset por causa desconocida

Para detectar este tipo de reset se implementó una entrada condicional que evalúa el valor del registro IFG1 también al inicio del "main()", de esta manera si no entra por reset de watchdog entrará por un reset de causa desconocida. El procedimiento para detectar este tipo de reset y registrar los tiempos correspondientes es similar al descripto anteriormente, sin necesidad de resetar el registro IFG1.

Reset por software

Este tipo de reset se da por la llamada a la función de reset() implementada en el ukernel. Esta función resetea al sistema escribiendo una clave errónea en el registro WDTCTL del watchdog del openmsp430. A diferencia de los eventos de reset detallados anteriormente, éste se registra una vez que se entra a la función reset() implementada en el módulo "mcs_hal.c". Cabe aclarar que este reset se contabilizará además como uno por watchdog. El algoritmo implementado para guardar los tiempos y llevar la cuenta de estos eventos es similar al ya descripto. Cabe mencionar que para todos los casos el número de eventos ocurridos esta topeado a un máximo de 50.

Finalmente, con el agregado que se mencionó anteriormente y la implementación del script GDB, queda el sistema en condiciones de realizar una primera campaña de inyección de fallas. En la misma no se incluyó detección de overflow ni inconsistencia de stack y tampoco se utilizó el triplicado de código previsto para agregar redundancia. Otra observación a destacar es que en esta primera etapa no se ataca el área de stack utilizada por el programa.

Segunda campaña de inyección de fallas

Para la segunda campaña de inyección de fallas fue necesaria la ampliación de las memorias de datos, y así poder desplazar el stack hacia arriba. Esto permitió alejarlo de la zona que se preserva para registrar los instantes de tiempo en que el programa atraviesa cada uno de los puntos críticos. Dichos registros ubicaron ahora entre las direcciones 0x500h y 0x700h.

Se incluye para esta etapa detección de overflow e inconsistencia de stack y se emplea asimismo triplicado de código. El objetivo es poner a prueba estos mecanismos de tolerancia a fallas mediante una nueva campaña de inyección. Fue necesario también ampliar la memoria de programa. Para agregar al código las propiedades mencionadas se incluyeron las siguientes definiciones de constantes en el módulo "ukernel_config.h":

- #define CONTROL_STACK_OVERFLOW
  "Control de desbordamiento de stack:
  Si se define CONTROL_STACK_OVERFLOW, el kernel coloca 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."

- #define CONTROL_STACK_CONSISTENCY
  "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 entrega la CPU y antes de entregarle el control, el kernel calcula el CRC de los datos y si fueron modificados genera un reset."

- #define KERNEL_COPY
  Triplicado de código. Además se debe definir el largo de cada función:

```
#define KERNEL_CODE_LENGTH_LOAD1 126
#define KERNEL_CODE_LENGTH_RUN1 18
#define KERNEL_CODE_LENGTH_SCHEDULER1 562
#define KERNEL_CODE_LENGTH_GET_RESOURCE_STATUS1 36
#define KERNEL_CODE_LENGTH_LOCK1 266
#define KERNEL_CODE_LENGTH_READ1 208
#define KERNEL_CODE_LENGTH_SET_BUFFER1 236
#define KERNEL_CODE_LENGTH_RESET_BUFFER1 148
#define KERNEL_CODE_LENGTH_SLEEP1 98
#define KERNEL_CODE_LENGTH_STOP1 18
#define KERNEL_CODE_LENGTH_TSLEEP1 30
#define KERNEL_CODE_LENGTH_UNLOCK1 124
#define KERNEL_CODE_LENGTH_WRITE1 324
#define KERNEL_CODE_LENGTH_WRITEI1 204
#define KERNEL_CODE_LENGTH_LOAD2 126
#define KERNEL_CODE_LENGTH_RUN2 18
#define KERNEL_CODE_LENGTH_SCHEDULER2 562
#define KERNEL_CODE_LENGTH_GET_RESOURCE_STATUS2 36
#define KERNEL_CODE_LENGTH_LOCK2 266
```

```
#define KERNEL_CODE_LENGTH_READ2 208
#define KERNEL_CODE_LENGTH_SET_BUFFER2 236
#define KERNEL_CODE_LENGTH_RESET_BUFFER2 148
#define KERNEL_CODE_LENGTH_SLEEP2 98
#define KERNEL_CODE_LENGTH_STOP2 18
#define KERNEL_CODE_LENGTH_TSLEEP2 30
#define KERNEL_CODE_LENGTH_UNLOCK2 124
#define KERNEL_CODE_LENGTH_WRITE2 324
#define KERNEL_CODE_LENGTH_WRITEI2 204
```

Luego de esto hace falta incluir el periférico que realiza chequeo de CRC. El mismo se implementó de modo que fuera coherente con el código C que ya se disponía para su configuración. En ese sentido, son dos las funciones encargadas de este asunto: “data_crc16” y “code_crc16”. La primera calcula el CRC de los datos en el stack antes de entregar el control a una aplicación y si fueron modificados genera un reset, mientras que la segunda calcula el CRC del espacio de memoria ocupado por la copia de cada función del módulo “ukernel.c”.

Al verificar el funcionamiento de control de las copias de código se constató que los resultados de los CRC cambiaban cada vez que se compilaba el programa. Esto se debe a optimizaciones efectuadas en cada compilación. Para solucionar este inconveniente se evaluó la posibilidad de deshabilitar las opciones de optimización del GCC pero ello derivaba en un código que superaba nuestras capacidades de almacenamiento (memorias). Otra opción manejada fue prescindir de ciertas funciones que presentaban errores a la hora de comparar los CRC calculados. Se consideró luego la posibilidad de agregar un nuevo algoritmo al µkernel que permitiera calcular el CRC de cada copia en tiempo de ejecución una única vez al comienzo del programa, utilizando después estos resultados.

Se enfrentó el inconveniente de que el periférico para cálculo de CRC funcionaba únicamente en la modalidad de 16 bits, mientras que algunas funciones requerían cálculo de 8 bits. En una primera instancia, la solución fue alterar la función code_crc16 de modo que para entradas de 8 bits el cálculo se realizara por software. Para casos de 16 bits se seguía derivando la tarea al hardware. A continuación se muestra el código de code_crc16 [18]:

```
#define POLY 0x1021
unsigned short code_crc16( char* addr, size_t size)
{
	unsigned char i;
	unsigned int data;
	unsigned int crc = 0xffff;
	if (size == 0)
		return (~crc);
	do
	{
		for (i=0, data=(unsigned int)0xff & *addr++; i < 8;
		i++, data >>= 1)
		{
		if ((crc & 0x0001) ^ (data & 0x0001))
			crc = (crc >> 1) ^ POLY;
		else
```

```
                crc >>= 1;
            }
        } while (--size);
        crc = ~crc;
        data = crc;
        crc = (crc << 8) | (data >> 8 & 0xff);
        return (crc);
}
```

Posteriormente, se modificó el periférico para que fuera capaz de distinguir entre entradas de 8 o 16 bits y empleara en cada caso un algoritmo diferente. (El periférico encargado del cálculo se detalla en la sección *Módulo CRC.)* De esa manera fue posible retornar al uso de la función code_crc16 original, que en todos los casos confía el cálculo de CRC al periférico. La misma se despliega a continuación.

```
/* Calcula el CRC de un bloque de código ubicado en diecciones bajas */
unsigned short code_crc16( char* addr, size_t size)
{
        CRCINIRES = 0xFFFF;
        while (size-- >0)
        {
                CRCDI_L = *addr;
                addr++;
        }
        return CRCINIRES;
}
```

Un cambio que es significativo e imprescindible para que la función “code_crc16()” funcione y calcule correctamente el CRC de las copias es cambiar el tipo del parámetro de entrada de la función original “cast_ptr()”. La misma convierte el puntero de la función en un puntero a char para ser usado como parámetro de entrada en la función que calcula el CRC.

Entonces la sección de código original:

```
        /** cast_ptr()
                Convierte un puntero a función en puntero a char
        */
        extern char* cast_ptr(unsigned long func_ptr);
```

cambia por:

```
        /** cast_ptr()
                Convierte un puntero a función en puntero a char
        */
        extern char* cast_ptr(unsigned int func_ptr);
```

Se debe tener en cuenta que es necesario cambiar además el parámetro de entrada en la llamada de la función “cast_ptr()”. Por ejemplo, para el caso de las funciones load_copy1 y load_copy2 quedaría de la siguiente manera:

```
if ((arreglo_crc1[2] == 0x0000) & (arreglo_crc2[2] == 0x0000)) {

      for(z=1;z<=14;z++){

            switch (z) {

                  case 1: arreglo_crc1[0]= code_crc16(
cast_ptr((int)load_copy1), KERNEL_CODE_LENGTH_LOAD1 );

                        arreglo_crc2[0]= code_crc16(
cast_ptr((int)load_copy2), KERNEL_CODE_LENGTH_LOAD2 );

                        break;
```

Unas líneas más abajo se encuentran las comparaciones de CRC, donde también se hace uso de la función cast_ptr() y por tanto también deberá ser modificado el código, por ejemplo para la función load():

```
/* load */
      if (code_crc16( cast_ptr((int)load_copy1),
            KERNEL_CODE_LENGTH_LOAD1 ) ==
            arreglo_crc1[0] )
            load = load_copy1;
      else if (code_crc16(cast_ptr((int)load_copy2),
            KERNEL_CODE_LENGTH_LOAD2 ) ==
            arreglo_crc2[0] )
            load = load_copy2;
      else
            load = load_copy3;
```

Vectores de entrada

Si bien el módulo diseñado permite inyectar fallas en buses de datos o direcciones, en la campaña sólo se apuntó a palabras específicas de memoria. Esto se debe a que en caso de alterar algún bit de un bus, se corrompe cualquier lectura o escritura, lo cual impide incluso almacenar correctamente los resultados.

Se inyectaron en la primera campaña un total de 150 fallas a cada memoria (datos y programa), divididas en 50 permanentes a 1, 50 permanentes a 0 y 50 de tipo SEU. Para ello se generaron aleatoriamente en Microsoft Excel vectores de entrada para cada corrida, encargados de setear los registros del módulo de fallas.

Recordando la estructura de los registros de control del módulo de inyección de fallas, se recuerdan criterios generales para la configuración del registro de control CTRL1 (CONTROL). El mismo tomará valores de la forma 0xXYZW, donde:

- X indica si el módulo está activo o no. Para que lo esté debe tener un 1 en el bit más alto, o sea, alcanza con que sea mayor o igual a 8.
- Y se usa para definir el bit de la falla (para casos de fallas en un bit).

- Z permite elegir entre fallas en bus de datos, bus de direcciones o palabras de memoria. Como se comentó, a fin de poder preservar cierto rango de direcciones por fuera del blanco de ataque de las fallas, se trabajará únicamente con la última opción, es decir, Z = 2.
- W determina si inyectaremos SEUs o fallas permanentes (y en este último caso si son a 0 o a 1).

A continuación se detalla el procedimiento empleado para generar los vectores que setean los registros en cada corrida de la campaña de inyección de fallas.

Fallas permanentes

CTRL1 (CONTROL): Se generaron 50 valores de la forma 0x8Y20 y 50 de la forma 0x8Y24, donde "Y" indica el bit afectado y es generado aleatoriamente entre 0 y F, mediante la siguiente fórmula: "DEC.A.HEX(REDONDEAR(ALEATORIO()*15;0))". Se aplicaron 50 variantes de seteo del CTRL1 tanto a memoria de datos como de programa.

CTRL2 (DIR_FALLA):

- Datos: Se generaron direcciones en el rango donde se mapea la memoria de datos (incluyendo el stack), es decir, entre la 0x0200 y la 0x09FF, exceptuando el intervalo entre la 0x0500 y la 0x06FF, reservado para almacenar banderas de control y resultados de la campaña. Cabe señalar que se trata únicamente de direcciones pares, ya que las mismas son de 16 bits, mientras que las memorias tienen un byte de ancho de palabra. Para apuntar a dos rangos disjuntos se anidaron dos funciones que generan valores aleatorios, una para elegir entre los dos rangos posibles y la otra para escoger el valor dentro del intervalo seleccionado anteriormente: "SI(ALEATORIO()<0,5;DEC.A.HEX(ALEATORIO.ENTRE(256;639)*2);DEC.A.HEX(ALEATORIO.ENTRE(896;1279)*2))"

- Programa: Para este caso es más sencillo porque las direcciones son todas contiguas, entre la 0xCFFF y 0xFFFF.
  Se empleó la fórmula "DEC.A.HEX(ALEATORIO.ENTRE(49152/2;65534/2)*2)".

CTRL3 (MÁSCARA): No se usa en este caso ya que únicamente se inyectaron fallas en un bit de la palabra, el cual es elegido mediante el CTRL1.

CTRL4 (TIEMPO): No corresponde para fallas permanentes.

SEUs

CTRL1 (CONTROL): El registro de control vale siempre 0x8023, ya que el bit de la falla se elige en este caso mediante una máscara (en el CTRL3).

CTRL2 (DIR_FALLA): Se usó el mismo procedimiento que para el caso de fallas permanentes, tanto para memoria de datos como de programa.

CTRL3 (MÁSCARA): Como se hará fallar únicamente un bit por corrida, la máscara debe tener un dígito en 1 y los demás en 0. Por ende, los valores posibles serán todos potencias de 2 (del 0 al 15), lo cual se implementó de la siguiente manera:

“DEC.A.HEX(2^(ALEATORIO.ENTRE(0;15)))”.

CTRL4 (TIEMPO): Se generaron aleatoriamente valores de 16 bits, es decir, entre 0x0000 y 0xFFFF, a través de la fórmula “DEC.A.HEX(ALEATORIO()*65535)”. El reloj que se usa es el mclk, cuya frecuencia es de 21 MHz, por lo cual la falla siempre se inyectará en el transcurso de los primeros 3 ms. Posteriormente se pudo notar que dicho valor resulta bastante bajo considerando que la aplicación dura en el caso normal alrededor de 400 ms. Por lo tanto, como mejora a futuro se puede considerar el agregado de un PLL al módulo de inyección de fallas para dividir la frecuencia de reloj.

7. RESULTADOS Y ANÁLISIS

En esta sección se presentan los resultados obtenidos y el análisis de los mismos. Como se comentó anteriormente, las pruebas se dividieron en dos campañas de inyección. Una primera introduciendo fallas al sistema µkernel básico y una segunda en la que se incluye detección de overflow e inconsistencia de stack, además de triplicado de código.

Se detallan los casos en que las fallas alteraron el curso normal de ejecución, es decir aquellos en las que la aplicación no concluye correctamente por alguna razón, finaliza habiendo ocurrido resets en el sistema o simplemente se retarda su ejecución. Asimismo, para cada una de estas situaciones se estudian las posibles causas de malfunctionamiento.

Primera campaña de inyección

Los resultados de la campaña de fallas quedan registrados en texto plano según lo programado en el script GDB. Los mismos se encuentran en el archivo "resultados.txt" incluido en la documentación entregada. Se constató en la práctica que para los escenarios de éxito, que son la mayoría, la aplicación finaliza luego de 0x666 cuentas (1638 en decimal) del timer A1. Dado que dicho timer funciona en base al aclk, configurado a una frecuencia de 32,768 kHz y empleando un divisor de 8, esto equivale a 8*(1/32,768 kHz)*1638 = 400 ms.

Fallas permanentes en memoria de datos

Se inyectaron 100 fallas permanentes en memoria de datos (50 a 1 y 50 a 0). No se incluyó el área de stack. Los casos dignos de mención se presentan a continuación.

Falla nº 70:

```
$70 = 0x24c
0x120:0x6924
0x2ca:0x5a28
#0  INT_TIMER1_OVERFLOW () at mcs_hal.c:115
#1  0x0000000b in ?? ()
#2  0x0000f200 in low_power_mode () at mcs_hal.c:333
#3  0x0000f4fc in scheduler_copy1 () at ukernel.c:345
#4  0x0000f5ae in run_copy1 () at ukernel.c:407
#5  run_copy1 () at ukernel.c:390
#6  0x0000f0ba in main () at mcs.c:93
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion no finalizada ***
```

Traza del programa

Mirando los detalles desplegados por el script GDB se puede concluir que el programa no sufrió ningún tipo de resets pero sin embargo la aplicación nunca finalizó. Lo que ha ocurrido aquí es una inyección de falla permanente a la dirección de datos 0x024C. Mirando el archivo "mcs.lst" generado por el compilador se puede observar el código generado en lenguaje assembler, el cual se utilizará para identificar qué fue lo que causó el error del programa. Para

este caso en particular, buscando en qué lugares se usa la dirección de datos 0x024C" se encontró la siguiente línea de código:

```
/* Busco el primer proceso listo */
      pid = 0;
      while    (pid    <    process_count    &&    PCB_Table[pid].pstatus    !=
PROCESS_STATUS_READY)
    f340:    1b 42 4c 02  mov &0x024c, r11
```

Lo anterior nos da una pista de que unos de los posibles motivos por los que la aplicación no haya finalizado -y de seguro nunca haya sido ejecutada- es que la dirección de datos en la que se inyectó la falla es la que se usa para evaluar el estado de un proceso, es decir la aplicación. En este caso el µkernel nunca pudo saber si la aplicación estaba READY (lista para ser ejecutada).

Falla nº 71:

```
$71 = 0x21c
0x120:0x6924
0x2ca:0x5a28
#0  INT_TIMER1_OVERFLOW () at mcs_hal.c:115
#1  0x0000000b in ?? ()
#2  0x0000f200 in low_power_mode () at mcs_hal.c:333
#3  0x0000f4fc in scheduler_copy1 () at ukernel.c:345
#4  0x0000f5ae in run_copy1 () at ukernel.c:407
#5  run_copy1 () at ukernel.c:390
#6  0x0000f0ba in main () at mcs.c:93
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion no finalizada ***
```

Nuevamente la aplicación no pudo finalizar. A continuación se estudiarán las posibles causas. En este caso la falla permanente se inyectó en la dirección 0x021C. Buscando el uso que el código hace de esta dirección se encontró entre otros el siguiente:

```
/* Busco el primer proceso listo */
      pid = 0;
      while    (pid    <    process_count    &&    PCB_Table[pid].pstatus    !=
PROCESS_STATUS_READY)
    f35c:    0b 93        tst    r11
    f35e:    19 24        jz     $+52          ;abs 0xf392
    f360:    1f 42 1c 02  mov    &0x021c, r15
    f364:    1f 93        cmp    #1,    r15    ;r3 As==01
    f366:    d8 24        jz     $+434         ;abs 0xf518
    f368:    1d 43        mov    #1,    r13    ;r3 As==01
    f36a:    0f 3c        jmp    $+32          ;abs 0xf38a
    f36c:    0e 4d        mov    r13,   r14
```

```
f36e:    0e 5e        rla    r14
f370:    0f 4e        mov    r14,   r15
f372:    0f 5f        rla    r15
f374:    0f 5f        rla    r15
f376:    0f 5f        rla    r15
f378:    0f 5e        add    r14,   r15
f37a:    3f 50 1c 02  add    #540,  r15    ;#0x021c
f37e:    2f 4f        mov    @r15,  r15
f380:    0e 4d        mov    r13,   r14
f382:    1e 53        inc    r14
f384:    1f 93        cmp    #1,    r15    ;r3 As==01
f386:    8f 24        jz     $+288         ;abs 0xf4a6
f388:    0d 4e        mov    r14,   r13
f38a:    0d 9b        cmp    r11,   r13
f38c:    ef 23        jnz    $-32          ;abs 0xf36c
f38e:    82 4b 12 02  mov    r11,   &0x0212
         {
               pid++;
         };
```

El código anterior muestra que efectivamente la dirección “0x21c”, en la cual se inyecta la falla, se utiliza para recorrer y buscar en la tabla de procesos el primero listo. Por consiguiente, una de las posibles razones por las que la aplicación no finaliza es que nunca es ejecutada, ya que la falla corrompe el código utilizado para recorrer la tabla de procesos.

Fallas permanentes en memoria de programa

Tras finalizar la recorrida por la memoria de datos se procedió a atacar la de programa inyectando otras 100 fallas. En este caso las que tuvieron incidencia en la ejecución fueron las siguientes:

<u>Falla nº 161:</u>

```
$161 = 0xf336 <scheduler_copy1+6>
0x120: 0x6925
0x2ca: 0x5a28
#0 f (p=0x480) at mcs.c:35
#1 0x00000000 in ?? ()
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion finalizada en el instante 800 ***
```

Aquí se observa que la aplicación demoró más de lo usual en finalizar, registrando un tiempo de 8*(1/32768Hz)*2048 = 500 ms. El retardo con respecto a la duración usual (400 ms) es el equivalente a un tic (100 ms) del timer A0. Al verificar qué parte del código afecta la falla a la dirección de programa “0xf336” se observó lo siguiente:

```
            /* Busco el primer proceso listo */
            pid = 0;
            while (pid < process_count && PCB_Table[pid].pstatus !=
PROCESS_STATUS_READY)
  f334: 1b 42 4c 02      mov     &0x024c,r11
```

La falla impone un “1” en el bit 14 de la dirección “0xf336”, como está señalado en el código. Por lo tanto, se guardará en el registro r11 el valor almacenado en la dirección 0x224c. En caso de que sea 0 es probable que el programa siga su curso normal y la aplicación logre finalizar, tal como se aprecia en esta ocasión, más allá de un leve retardo en su ejecución.

Falla nº 192:

```
$192 = 0xf69e <reset_buffer_copy1+28>
0x120:0x6925
0x2ca:0x5a28
#0  f (p=0x480) at mcs.c:35
#1  0x00000000 in ?? ()
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion finalizada en el instante 801 ***
```

Aquí se observa que la aplicación demoró más de lo usual en finalizar, registrando nuevamente un tiempo de 8*(1/32768Hz)*2049 = 0,50 segundos. Al verificar qué parte del código afecta la falla a la dirección de programa “0xf69e”, se observó lo siguiente:

```
struct RCB* pRCB = &RCB_Table[resource];
    f69c:    0f 5f        rla    r15
    f69e:    0e 4f        mov    r15,   r14
    f6a0:    0e 5e        rla    r14
    f6a2:    0e 5e        rla    r14
    f6a4:    0e 5e        rla    r14
    f6a6:    0e 8f        sub    r15,   r14
    f6a8:    3e 50 4e 02  add    #590,  r14    ;#0x024e
```

El código anterior corresponde a la función del módulo µkernel reset_buffer_copy1(). No hay aplicación que haga uso de un buffer por lo cual esta falla no afecta el flujo que toma el programa.

SEUs en memoria de datos

En la primera campaña se inyectaron 50 SEUs en memoria de datos, siendo el siguiente el único caso digno de mención.

Falla nº 2:

```
$2 = 0x240
0x120: 0x6925
0x2ca: 0x5a28
#0  f (p=0x480) at mcs.c:35
#1  0x00000000 in ?? ()
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion finalizada en el instante 800 ***
```

Tal como ocurrió en situaciones ya estudiadas, se observa que la aplicación demoró más de lo usual en concluir, registrando un tiempo de 0,50 segundos. En busca de una posible causa se repite el procedimiento seguido hasta el momento: revisar el archivo "mcs.lst" para ver qué variable es afectada por la falla.

Para este caso se constató que la dirección de datos 0x0240 es asignada a la variable "kernel_sp", usada para almacenar el contenido del stack pointer del kernel. Un SEU en esta dirección no afecta la ejecución del programa si se inyecta antes de ser inicializada la variable. Esto se debe a que el efecto de un SEU desaparece con una posterior escritura en la dirección de la falla. Sin embargo, podría tratarse de una falla crítica en el caso de que se active luego de que la variable fuera cargada, ya que tendría un valor incorrecto al ser usada por el kernel.

SEUs en memoria de programa

Nuevamente se introdujeron 50 SEUs pero en memoria de programa. Las fallas que alteraron el flujo esperable del programa se presentan a continuación:

Falla nº 35:

```
$35 = 0xf7ea <lock_copy1+78>
0x120: 0x6925
0x2ca: 0x5a28
#0  f (p=0x480) at mcs.c:35
#1  0x00000000 in ?? ()
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion finalizada en el instante 800 ***
```

Revisando la información aportada por el módulo "mcs.lst" se verifica que el área de código donde se inyectó la falla corresponde a la función lock_copy1() del kernel. Es de esperar que la aplicación finalice correctamente –como efectivamente ocurre- dado que el flujo del programa

no requiere una llamada a la función lock_copy1(). Es correcto entonces que no ocurran resets pero se desconoce el motivo del retardo, que nuevamente es de 0,10 segundos por sobre el valor usual.

Resumen

A modo de resumen de esta primera etapa se presenta una caracterización de la primera campaña de inyección de fallas en base al *Modelo FARM* descripto anteriormente.

Faults: Se inyectaron 50 fallas permanentes a 1, 50 permanentes a 0 y 50 SEUs, tanto en memoria de datos como de programa, totalizando 300 casos distintos. En el caso de memoria de datos, no se ataca el stack.

Activation: El sistema al cual se le inyectan fallas es la *Aplicación sencilla* corriendo sobre la primera versión básica del µkernel.

Readouts: Se verificó para cada una de las 300 fallas si la aplicación finaliza correctamente, almacenando además en caso afirmativo el tiempo le toma. Además se registraron la cantidad de resets generados por software, por watchdog o por algún motivo diferente.

Measures: Las medidas efectuadas se resumen en la Tabla 17.

Tabla 17: Medidas tomadas en la primera campaña

	Total	Sin efecto	No finaliza	Retardo	Time out	Reset por watchdog
Perm datos	100	98	2	0	2	0
Perm prog	100	98	0	2	0	0
SEUs datos	50	49	0	1	0	0
SEUs progr	50	49	0	1	0	0

Segunda campaña de inyección

Para esta segunda etapa de inyección de fallas se mantuvo la misma aplicación que corre para la campaña anterior. Sin embargo, el criterio de rigurosidad cambia al incluir como blanco de ataque también el área del stack reservado para las funciones del µkernel y aplicación. Dicho espacio comienza ahora en la dirección 0x0A00 y se reservan 0x0100 lugares para la aplicación, descendiendo hasta la 0x0900. El área de stack para las funciones del µkernel queda entonces por debajo de la 0x0900. Asimismo, como se mencionó anteriormente, se ampliaron los tamaños de memoria de datos y programa.

Cabe destacar también que en esta instancia se incluirá chequeo de CRC de las copias de las funciones del ukernel. Por ende, se ha decidido activar las fallas una vez que se haya realizado el cálculo de los mismos en la función “kernelinit()”. Hechas estas aclaraciones, se procede a desplegar los resultados de la campaña.

Fallas permanentes en memoria de datos

Observando lo desplegado en el archivo “resultados.txt” se listan a continuación los resultados más significativos.

1)

```
*** Direccion atacada***
$33 = 0x214
*** Palabra de control***
$34 = 0x8520
*** Puntero a funciones***
$35 = 0xc27e
$36 = 0xc2d8
$37 = 0xc4c2
$38 = 0xc5f6
$39 = 0xc61a
$40 = 0xc674
$41 = 0xc932
$42 = 0xcc86
$43 = 0xcf6a
$44 = 0xd162
$45 = 0xd2ee
$46 = 0xd57e
$47 = 0xd784
$48 = 0xdc78
#0  kernel_init () at ukernel.c:2467
#1  0x0000c0b4 in main () at mcs.c:97
*** ocurrieron 1 reset por watchdog ***
instantes registrados
14cd
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion no finalizada ***
```

Al debuggear la falla individualmente se percibe la generación de un PUC reset mientras se recorre la tabla de procesos dentro de la fungión “Scheduler_copy1()”. La falla ataca el 5º byte de la tabla de procesos lo cual genera que el micro se reinicie en un momento dado. Esto ocurre luego de que la aplicación es ejecutada dos veces por el ukernel.

2)

```
*** Direccion atacada***
$497 = 0x7fc
*** Palabra de control***
$498 = 0x8e20
*** Puntero a funciones***
$499 = 0xc27e
$500 = 0xc2d8
$501 = 0xc4c2
$502 = 0xc5f6
$503 = 0xc61a
$504 = 0xc674
$505 = 0xc932
$506 = 0xcc86
$507 = 0xcf6a
$508 = 0xd162
$509 = 0xd2ee
$510 = 0xd57e
$511 = 0xd784
$512 = 0xdc78
#0  kernel_init () at ukernel.c:2467
#1  0x0000c0b4 in main () at mcs.c:97
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 1 reset por causa desconocida ***
instantes registrados
0
*** aplicacion no finalizada ***
```

Observando el flujo del programa al inyectar individualmente esta falla se observa que se genera un PUC reset por motivo desconocido antes de correr la aplicación.

3)

```
*** Direccion atacada***
$833 = 0x9fa
*** Palabra de control***
$834 = 0x8e24
*** Puntero a funciones***
$835 = 0xc27e
$836 = 0xc2d8
$837 = 0xc4c2
$838 = 0xc5f6
$839 = 0xc61a
$840 = 0xc674
$841 = 0xc932
```

```
$842 = 0xcc86
$843 = 0xcf6a
$844 = 0xd162
$845 = 0xd2ee
$846 = 0xd57e
$847 = 0xd784
$848 = 0xdc78
#0  INT_TIMER1_OVERFLOW () at mcs_hal.c:548
#1  0x0000000b in ?? ()
#2  0x0000c18e in low_power_mode () at mcs_hal.c:284
#3  0x0000de9a in scheduler_copy1 () at ukernel.c:2072
#4  0x0000c0ce in main () at mcs.c:106
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion no finalizada ***
```

Al correr el programa inyectando la falla individualmente y debuggeando el mismo paso a paso se verifica que efectivamente la aplicación corre una única vez y luego no vuelve a ejecutarse. El scheduler encuentra otro proceso listo antes de que expire la cuenta del Timer A1, dando por finalizada la sesión de falla.

4)

```
*** Direccion atacada***
$1265 = 0x27e
*** Palabra de control***
$1266 = 0x8324
*** Puntero a funciones***
$1267 = 0xc27e
$1268 = 0xc2d8
$1269 = 0xc4c2
$1270 = 0xc5f6
$1271 = 0xc61a
$1272 = 0xc674
$1273 = 0xc932
$1274 = 0xcc86
$1275 = 0xcf6a
$1276 = 0xd162
$1277 = 0xd2ee
$1278 = 0xd57e
$1279 = 0xd784
$1280 = 0xdc78
#0  INT_TIMER1_OVERFLOW () at mcs_hal.c:548
#1  0x0000000b in ?? ()
#2  0x0000c18e in low_power_mode () at mcs_hal.c:284
#3  0x0000de9a in scheduler_copy1 () at ukernel.c:2072
#4  0x0000c2e6 in run_copy1 () at ukernel.c:346
```

```
#5  0x0000c0ce in main () at mcs.c:106
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion no finalizada ***
```

La dirección atacada en este caso es la utilizada para guardar el comienzo de la tabla de procesos, es decir que la variable afectada es PCB_table y el efecto que causa la falla sobre el programa es evitar que la funcion Scheduler() encuentre un proceso listo para ser ejecutado, evitando de esta manera la ejecución de la aplicación hasta que expira el timerA1 lo cual finaliza la ejecución de ukernel.

Fallas permanentes en memoria de programa

1) La falla que se detalla a continuación no es considerada crítica para el sistema que incluye triplicado de código. Si bien la misma ataca una dirección de programa en la cual se ubica la función “Scheduler_copy1()”, esto no genera inconvenientes. Se incluye el resultado a continuación:

```
*** Direccion atacada***
$2209 = 0xdd74 <scheduler_copy1+252>
*** Palabra de control***
$2210 = 0x8520
*** Puntero a funciones***
$2211 = 0xc27e
$2212 = 0xc2d8
$2213 = 0xc4c2
$2214 = 0xc5f6
$2215 = 0xc61a
$2216 = 0xc674
$2217 = 0xc932
$2218 = 0xcc86
$2219 = 0xcf6a
$2220 = 0xd162
$2221 = 0xd2ee
$2222 = 0xd57e
$2223 = 0xd784
$2224 = 0xda46
#0  f (p=0x650) at mcs.c:46
#1  0x0000aaaa in ?? ()
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion finalizada en el instante 666 ***
```

Se puede observar que el puntero de la última función está ubicado en la dirección 0xda46, lo cual significa que se ha descartado la función “scheduler_copy1()”, al no superar la prueba de CRC. Por lo tanto, el programa utilizó en este caso la copia 2 de la función “Scheduler”. Esto

demuestra el correcto funcionamiento del sistema de chequeo de CRC y triplicado de código del “µkernel”. Un análisis más exhaustivo del funcionamiento de este mecanismo se incluye en la

Prueba del control de consistencia para las múltiples copias de código.

SEUs en memoria de datos

Se encontró un único caso con comportamiento anormal:

```
*** Direccion a atacar***
$18 = 0x780
*** Palabra de control***
*** 0x8023 ***
*** Cantidad de ciclos a esperar para activar la falla***
$19 = 0x65c6
*** Mascara a utilizar ***
$20 = 0x4
*** Puntero a funciones***
$21 = 0xc27e
$22 = 0xc2d8
$23 = 0xc4c2
$24 = 0xc5f6
$25 = 0xc61a
$26 = 0xc674
$27 = 0xc932
$28 = 0xcc86
$29 = 0xcf6a
$30 = 0xd162
$31 = 0xd2ee
$32 = 0xd57e
$33 = 0xd784
$34 = 0xdc78
#0  kernel_init () at ukernel.c:2467
#1  0x0000c0b4 in main () at mcs.c:97
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 1 reset por causa desconocida ***
instantes registrados
0
*** aplicacion no finalizada ***
```

Al correr la falla independiente se observó una inconsistencia de GDB, a causa de la cual el programa cae dos veces seguidas en el breakpoint seteado para configurar los módulos de fallas, una vez calculados los CRC del programa. El programa queda entonces estancado en el breakpoint, por lo cual es necesario pasar a la siguiente inyección de falla de forma manual desde GDB, mediante “CTRL + C”. Al no haber concluido la corrida anterior se despliega el mensaje de “aplicación no finalizada”.

Una posible explicación a esta situación es que hay momentos en los que el kernel deshabilita las interrupciones, lo cual impediría que el timer A1 finalice la corrida por timeout. Una

modificación que podría subsanar este inconveniente sería cofigurar las interrupciones del timer A1 como no enmascarables.

SEUs en memoria de programa

A continuación se presentan los casos de malfuncionamiento:

1)

```
*** Direccion a atacar***
$647 = 0xe060 <kernel_init+438>
*** Palabra de control***
*** 0x8023 ***
*** Cantidad de ciclos a esperar para activar la falla***
$648 = 0xf7fd
*** Mascara a utilizar ***
$649 = 0x40
*** Puntero a funciones***
$650 = 0xcc86
$651 = 0xcc86
$652 = 0xcc86
$653 = 0xcc86
$654 = 0xcc86
$655 = 0xcc86
$656 = 0xcc86
$657 = 0xcc86
$658 = 0xcc86
$659 = 0xcc86
$660 = 0xcc86
$661 = 0xcc86
$662 = 0xcc86
$663 = 0xcc86
#0  kernel_init () at ukernel.c:2467
#1  0x0000c0b4 in main () at mcs.c:97
*** ocurrieron 1 reset por watchdog ***
instantes registrados
665
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion no finalizada ***
```

Analizando la información que dispone el archivo “mcs.lst” se destaca la siguiente sección de código de interés:

```
/* write */
      if (code_crc16( cast_ptr((int)write_copy1),
    e05a:    3f 40 86 cc  mov    #-13178,r15  ;#0xcc86
    e05e:    b0 12 b2 e6  call   #0xe6b2
```

```
e062:    3e 40 44 01  mov    #324,  r14    ;#0x0144
e066:    b0 12 1e c2  call   #0xc21e
e06a:    1f 92 e2 02  cmp    &0x02e2,r15
e06e:    02 24        jz     $+6           ;abs 0xe074
e070:    30 40 04 e4  br     #0xe404
         KERNEL_CODE_LENGTH_WRITE1 ) ==
         arreglo_crc1[12] )
         write = write_copy1;
e074:    b2 40 86 cc  mov    #-13178,&0x02ee     ;#0xcc86
e078:    ee 02
  else
         write = write_copy3;
```

Observando que la falla cae en el código de la función “call”, tal cual se destaca en gris, se deduce la causa del reset generado. Considerando que la función Kernel_init() no tiene copias asociadas, el programa se hace vulnerable en este punto, en el cual se configuran las copias a usar. El malfuncionamiento del programa se debe a un salto a una dirección incorrecta debido a que la falla ha alterado estos bits.

2)

```
*** Direccion a atacar***
$715 = 0xc188 <low_power_mode>
*** Palabra de control***
*** 0x8023 ***
*** Cantidad de ciclos a esperar para activar la falla***
$716 = 0x7a8b
*** Mascara a utilizar ***
$717 = 0x2
*** Puntero a funciones***
$718 = 0xc27e
$719 = 0xc2d8
$720 = 0xc4c2
$721 = 0xc5f6
$722 = 0xc61a
$723 = 0xc674
$724 = 0xc932
$725 = 0xcc86
$726 = 0xcf6a
$727 = 0xd162
$728 = 0xd2ee
$729 = 0xd57e
$730 = 0xd784
$731 = 0xdc78
#0  low_power_mode () at mcs_hal.c:284
#1  0x0000de9a in scheduler_copy1 () at ukernel.c:2072
#2  0x0000c0ce in main () at mcs.c:106
*** ocurrieron 0 reset por watchdog ***
```

```
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion no finalizada ***
```

Observando atentamente el código que arroja el archivo “mcs.lst” se distingue lo siguiente:

```
0000c188 <low_power_mode>:
 *    LOW POWER MODE
 */
void low_power_mode(void)
{
      //__low_power_mode_0();
      eint();
    c188:    32 d2        eint
      LPM0;
    c18a:    32 d0 10 00  bis    #16,   r2     ;#0x0010
}
    c18e:    30 41        ret
```

La falla altera el código de programa donde se habilitan las interrupciones antes de entrar en el modo de bajo consumo. Al verse afectada esta sección, es muy probable que el programa haya quedado estancado en el modo de bajo consumo sin salir del mismo, ya que las interrupciones nunca lo estarían despertando.

3)

```
*** Direccion a atacar***
$698 = 0xe0d4 <kernel_init+554>
*** Palabra de control***
*** 0x8023 ***
*** Cantidad de ciclos a esperar para activar la falla***
$699 = 0x5272
*** Mascara a utilizar ***
$700 = 0x40
*** Puntero a funciones***
$701 = 0xc27e
$702 = 0xc2d8
$703 = 0xc4c2
$704 = 0xc5f6
$705 = 0xc61a
$706 = 0xc674
$707 = 0xc932
$708 = 0xcc86
$709 = 0xcf6a
$710 = 0xd162
$711 = 0xd2ee
$712 = 0xd57e
$713 = 0xd784
```

```
$714 = 0xdc78
#0  INT_TIMER1_OVERFLOW () at mcs_hal.c:548
#1  0x0000001b in ?? ()
#2  0x0000c18e in low_power_mode () at mcs_hal.c:284
#3  0x0000de9a in scheduler_copy1 () at ukernel.c:2072
#4  0x0000c0ce in main () at mcs.c:106
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por violacion de memoria ***
*** aplicacion no finalizada ***
```

La falla afecta una dirección de memoria de programa encargada de inicializar la tabla de recursos. Un posible motivo que explique el resultado de la falla puede asociarse al efecto que genera modificar parte del código del programa lo que puede derivar en el flujo inadecuado del mismo.

Resumen

Se despliega ahora un resumen de la segunda campaña de inyección de fallas en base al *Modelo FARM*.

Faults: Se inyectaron nuevamente 50 fallas permanentes a 1, 50 permanentes a 0 y 50 SEUs, tanto en memoria de datos como de programa, totalizando 300 casos distintos. Se incluyó el área de stack en el rango atacado.

Activation: El sistema al cual se le inyectan fallas es la *Aplicación sencilla* corriendo sobre el sistema µkernel, que este caso cuenta con algunos agregados: chequeo de overflow e inconsistencia de stack y triplicado de código.

Readouts: Se verificó para cada una de las 300 fallas si la aplicación finaliza correctamente, almacenando además en caso afirmativo el tiempo le toma. Además se registraron la cantidad de resets generados por software, por watchdog o por algún motivo diferente.

Measures: La siguiente tabla resume las medidas tomadas:

Tabla 18: Medidas tomadas en la segunda campaña

	Total	Sin efecto	No finaliza	Retardo	Time out	Reset por watchdog	Reset por otra causa
Perm datos	100	96	4	0	2	1	1
Perm prog	100	100	0	0	0	0	0
SEUs datos	50	49	1	0	0	0	1
SEUs progr	50	47	3	0	0	1	0

Prueba del control de consistencia para las múltiples copias de código

El µkernel cuenta con la posibilidad de triplicar el almacenamiento de algunas de sus funciones más importantes y testear la integridad de las mismas –mediante chequeo de CRC- antes de emplearlas. En caso de detectar una inconsistencia en alguna de ellas se pasa a la siguiente copia. En esta fase de pruebas se inyectarán fallas permanentes en direcciones correspondientes a funciones del µkernel para ver si este último responde adecuadamente.

En la Tabla 19 se muestran los rangos de direcciones donde se alojan cada una de copias de todas las funciones que se encuentran triplicadas. Cabe destacar que en caso de fallar la comprobación de las primeras dos copias se usa automáticamente la tercera sin que la misma sea chequeada.

Tabla 19: Direcciones de las diferentes copias de cada función

	COPIA 1	**COPIA 2**	**COPIA 3**
	Comienzo - Fin	Comienzo - Fin	Comienzo – Fin
stop	C27E – C290	C290 – C2A2	C2A2 -- C2B4
run	C2D8 -- C2EA	C2C6 – C2D8	C2B4 -- C2C6
set_buffer	C4C2 -- C5AE	C3D6 -- C4C2	C2E0 -- C3D6
get_resource_status	C5F6 -- C61A	C5D2 -- C5F6	C5A6 -- C5D2
tsleep	C61A -- C638	C638 -- C656	C656 -- C674
sleep	C674 -- C6D6	C6D6 -- C738	C866 -- C932
writei	C932 -- C9FE	C866 -- C932	C79A -- C866
write	CC86 -- CDCA	CB42 -- CC86	C9FE -- CB42
read	CF6A -- D03A	CE9A -- CF6A	CDCA -- CE9A
reset_buffer	D162 -- D1F6	D0CE -- D162	D03A -- D0CE
unlock	D2EE -- D36A	D272 -- D2EE	D1F6 -- D272
lock	D57E -- D688	D474 -- D57E	D36A -- D474
load	D784 -- D802	D706 -- D784	D688 -- D706
scheduler	DC78 -- DEAA	DA46 -- DC78	D814 -- DA46

En una primera instancia se correrá una aplicación sin ningún tipo de falla presente para poder comprobar que todas las funciones utilizadas corresponden a la copia 1. Para esto, se utilizará una copia modificada del script que se encarga de realizar las campañas de inyección de fallas para realizar los pasos de configuración necesarios para correr la aplicación. Entre los resultados, más allá de valores sobre cantidad de reset, captura de tiempo de los mismos y tiempo que duró la ejecución, se encontrará el valor de cada uno de los punteros destinados a almacenar el comienzo de cada función a utilizar. Vale decir que si el puntero load presenta el valor 0xD784 significa que el µkernel utilizó la copia 1. En cambio, si vale 0xD706 implica que utilizó la copia 2 y por último es claro que 0xD688 hace referencia a la copia 3.

A continuación se despliegan los resultados que arroja el script:

```
<<<<<<FALLA INDIVIDUAL, PARA ANALIZAR COMPORTAMIENTO>>>>>>
*** Direccion atacada***
$1 = 0xdc98 <scheduler_copy1+32>
*** Palabra de control***
```

```
$2 = 0x25
*** Mascara utilizada para la falla ***
$3 = 0xf00
*** Puntero a funciones***
$4 = 0xc27e
$5 = 0xc2d8
$6 = 0xc4c2
$7 = 0xc5f6
$8 = 0xc61a
$9 = 0xc674
$10 = 0xc932
$11 = 0xcc86
$12 = 0xcf6a
$13 = 0xd162
$14 = 0xd2ee
$15 = 0xd57e
$16 = 0xd784
$17 = 0xdc78
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset causa desconocida ***
*** aplicacion finalizada en el instante 666 ***
```

Cabe señalar que si bien hay indicada una dirección de falla “0xdc98”, la misma nunca se activa dado que la palabra de control empleada “0x25” mantiene deshabilitado al módulo de inyección. También se puede apreciar que la aplicación culminó correctamente como es de esperar en este caso. Por último, se analizaran los correspondientes punteros a función, los cuales si bien no aparecen con sus respectivos nombres sí lo hacen en el mismo orden en que se pueden apreciar en la Tabla 19.

Tabla 20: Dirección de la copia 1 y puntero utilizado por el ukernel

PUNTERO	COPIA 1	PUNTERO UTILIZADO POR EL UKERNEL
stop	C27E – C290	0xc27e
run	C2D8 -- C2EA	0xc2d8
set_buffer	C4C2 -- C5AE	0xc4c2
get_resource_status	C5F6 -- C61A	0xc5f6
tsleep	C61A -- C638	0xc61a
sleep	C674 -- C6D6	0xc674
writei	C932 -- C9FE	0xc932
write	CC86 -- CDCA	0xcc86
read	CF6A -- D03A	0xcf6a
reset_buffer	D162 -- D1F6	0xd162
unlock	D2EE -- D36A	0xd2ee
lock	D57E -- D688	0xd57e
load	D784 -- D802	0xd784
scheduler	DC78 -- DEAA	0xdc78

En la Tabla 20 vemos que todos los punteros tienen como contenido el comienzo de las copias 1 de las funciones a utilizar. Por lo tanto, se concluye que todas las copias 1 superaron el control de consistencia para poder ser utilizadas.

A continuación se insertará una falla en la copia 1 de la función load, por lo cual es de esperar que el puntero de la función load apunte al comienzo de la copia 2 de dicha función. Se muestra la salida del script implementado para insertar dicha falla:

```
<<<<<<FALLA INDIVIDUAL, PARA ANALIZAR COMPORTAMIENTO>>>>>>
*** Direccion atacada***
$35 = 0xd796 <load_copy1+18>
*** Palabra de control***
$36 = 0x8025
*** Mascara utilizada para la falla ***
$37 = 0x2
*** Puntero a funciones***
$38 = 0xc27e
$39 = 0xc2d8
$40 = 0xc4c2
$41 = 0xc5f6
$42 = 0xc61a
$43 = 0xc674
$44 = 0xc932
$45 = 0xcc86
$46 = 0xcf6a
$47 = 0xd162
$48 = 0xd2ee
$49 = 0xd57e
$50 = 0xd706
$51 = 0xdc78
*** ocurrieron 0 reset por watchdog ***
*** ocurrieron 0 reset por software ***
*** ocurrieron 0 reset por causa desconocida ***
*** aplicacion finalizada en el instante 667 ***
```

Esta vez un análisis del contenido del resultado del script nos permite ver que la falla introducida corresponde a una del tipo permanente a 1 con mascara personalizada. Esta mascara posee el valor 0x0002, lo cual provoca que el segundo bit menos significativo de la dirección afectada -0xD796 en este caso- quede “atascado” en 1. El propio script nos informa que la dirección de la falla se encuentra 18 lugares a partir del comienzo de la función load_copy1. Una lectura previa a esta dirección nos devolvió el valor que se almacenaba, 0x4139. De acuerdo a lo explicado anteriormente, el 9 del último nibble pasa a ser una “B” (1001 -> 1011). Por ende, mediante el análisis de los punteros debería corroborarse el uso de la copia 2, ya que nuestro módulo de CRC es capaz de detectar cambios de un bit en un 100% de los casos. Se procede entonces a analizar los punteros:

Tabla 21: Direcciones de las copias 1 y 2 y puntero utilizado por el ukernel

PUNTERO	COPIA 1	COPIA 2	PUNTERO UTILIZADO POR EL UKERNEL
stop	C27E – C290	C290 – C2A2	0xc27e
run	C2D8 -- C2EA	C2C6 – C2D8	0xc2d8
set_buffer	C4C2 -- C5AE	C3D6 -- C4C2	0xc4c2
get_resource_status	C5F6 -- C61A	C5D2 -- C5F6	0xc5f6
tsleep	C61A -- C638	C638 -- C656	0xc61a
sleep	C674 -- C6D6	C6D6 -- C738	0xc674
writei	C932 -- C9FE	C866 -- C932	0xc932
write	CC86 -- CDCA	CB42 -- CC86	0xcc86
read	CF6A -- D03A	CE9A -- CF6A	0xcf6a
reset_buffer	D162 -- D1F6	D0CE -- D162	0xd162
unlock	D2EE -- D36A	D272 -- D2EE	0xd2ee
lock	D57E -- D688	D474 -- D57E	0xd57e
load	D784 -- D802	D706 -- D784	0xd706
scheduler	DC78 -- DEAA	DA46 -- DC78	0xdc78

Efectivamente se observa lo previsto, es decir, la copia 1 al no verificar el código CRC calculado inicialmente será descartada y se pasará a analizar la copia 2. Dado que esta no presenta alteraciones, verifica el CRC y es utilizada. Es de destacar que la falla introducida no provoca ningún error de gravedad ya que el µkernel resuelve utilizar otra copia que no presenta errores de consistencia sin afectar el funcionamiento global del sistema.

Este fenómeno puede que se presente en más de una oportunidad y pase desapercibido para el caso de fallas en memoria de programa durante la segunda campaña de inyección. De hecho, si bien es interesante considerar puntualmente casos como el que se acaba de presentar, no se busca detectarlos en las campañas masivas dado que no derivan en un malfuncionamiento del sistema. Además, claro está que demandaría un tiempo considerablemente mayor.

Variables críticas detectadas

Se inyectaron fallas permanentes en algunas direcciones específicas de interés, previendo que podían tener efecto en el flujo del programa. Por ejemplo, se constató la importancia de la variable tics, utilizada para definir el flujo principal de la función scheduler(). En particular, se observó que si su valor queda estancado en un número mayor a 0 el scheduler() nunca ejecuta las aplicaciones. Lo mismo ocurre si se afecta la variable “process.timer”, que decrementa su valor en cada tic generado mientras un proceso queda a la espera de ser ejecutado cuando la misma llegue a 0.

8. CONCLUSIONES

Para evaluar el trabajo realizado se comentarán por separado los resultados de las dos campañas de inyección de fallas, para luego detenernos en algunos casos puntuales y finalizar con un balance global.

La primera campaña sirvió como etapa de aprendizaje, permitiendo pulir las técnicas para realizar campañas masivas sobre el µkernel. Posibilitó generar un método para automatizar la recolección de resultados y visualizar una serie de atajos a la hora de utilizar el módulo de inyección. Se adquirieron además nociones sobre cómo inyectar fallas con la precaución de no afectar el mecanismo de almacenamiento de resultados ni las variables involucradas en dicho proceso.

Tal vez esta primera etapa no derive en la obtención de resultados sustanciales para evaluar la robustez del µkernel, pero sí se sientan las bases de lo que podría llegar a ser un análisis más profundo para estudios de esta naturaleza. Pueden considerarse entonces más valiosos la experiencia y los conocimientos adquiridos en cuanto a la metodología para realizar una campaña de inyección de fallas que los resultados en sí.

De todos modos, cabe señalar se constataron ciertos hechos de interés, como la importancia que posee la variable tics, utilizada para definir el flujo principal de la función scheduler(). En particular, se observó que si su valor queda estancado en un número mayor a 0 el scheduler() nunca ejecuta las aplicaciones. Lo mismo ocurre si se afecta la variable “process.timer”, que decrementa su valor en cada tic generado mientras un proceso queda a la espera de ser ejecutado cuando la misma llegue a 0. Es de destacar que el sistema fue posteriormente fortalecido por el equipo del satélite en cuanto a su robustez para este tipo de casos. Según se comentó con Gustavo de Martino, la posible aparición de fallas permanentes en variables de este estilo no estaba considerada en las primeras versiones del µkernel usadas al comienzo de nuestro trabajo.

En la segunda campaña de inyección de fallas se incluyó detección de stack overflow e inconsistencia de datos en el área de stack de cada aplicación. A su vez se aumentó el nivel de rigurosidad al incluir el área de stack como posible zona a inyectar fallas. Otra característica incorporada fue la opción de triplicado de código, lo que permite al µkernel usar diferentes copias según el resultado del cálculo de CRC de las mismas. Esta propiedad agrega redundancia, permitiendo la recuperación del código ante su eventual corrupción ocasionada por una falla en memoria de programa.

Al observar los resultados obtenidos para el caso de memoria de datos, se distinguieron zonas de alta vulnerabilidad en las que un cambio de un bit en direcciones críticas altera el flujo normal del programa, impidiendo que la aplicación concluya exitosamente. Como se comentó previamente, las fallas consideradas críticas para el sistema son aquellas que afectan a las funciones que resultan vitales para el funcionamiento del módulo µkernel (scheduler, load, run, kernel_init) y a sus variables principales, como la tabla de procesos y sus variables asociadas.

El chequeo de CRC, el cual permite elegir entre tres posibles copias de código de las funciones del µkernel, provee en principio mayor robustez al sistema. Pese a ello siguen existiendo aéreas de programa vulnerables. Los resultados obtenidos denuncian una alta sensibilidad ante fallas en la implementación del código que configura las diferentes copias de programa

dentro de la función “kernel_init()”. Esta última no se encuentra triplicada, de modo que cualquier falla inyectada en estas direcciones de programa puede ser crítica para el kernel.

En resumen, si bien el número de fallas inyectado no es suficiente como sacar grandes conclusiones en aspecto estadístico, permitió igualmente el análisis de algunos casos de interés. Asimismo, el trabajo realizado provee las herramientas básicas para la realización de campañas de inyecciones de fallas, pudiendo a futuro introducir mejoras.

Un objetivo sería realizar mayor número de corridas en las campañas, además de ampliar el espacio de fallas en general, incluyendo por ejemplo registros del microprocesador. También podría pensarse en atacar los buses de comunicación I2C y SPI, lo cual iría de la mano con mejorar la emulación del hardware.

Otra de las metas que podrían fijarse a futuro sería independizar el mecanismo de inyección de fallas y registro de resultados del openMPS430. Además de la ventaja de ser menos invasivo al sistema bajo estudio, facilitaría su reutilización. Asimismo, queda pendiente usar un mecanismo de activación que se asemeje más a las tareas que pueda ejecutar el kernel del satélite.

ANEXOS

A continuación se adjunta una serie de apartados en los que se ahonda más detalladamente en algunos de los contenidos presentados hasta el momento, al tiempo que se describen ciertos procedimientos y se explica el uso de varias herramientas utilizadas.

Anexo 1: Periféricos del MSP430

MSP430 es una familia de microcontroladores fabricados por Texas Instruments para aplicaciones de bajo costo y poco consumo de energía. En el ANTEL-SAT se usan chips de las familias MSP430x5xx y MSP430x6xx. A continuación se describen los periféricos presenten en esta variedad de integrados.

Interfaz USCI

El MSP430 incluye 2 módulos diferentes de USCI, USCI_A y USCI_B.
Cada uno de ellos soporta los siguientes modos:

USCI_A:
- Modo UART
- Formas de pulso para comunicaciones IrDA
- Detección automática de ratios para comunicaciones LIN
- Modo SPI

USCI_B:
- Modo I2C
- Modo SPI

SPI

SPI (serial peripheral interface) es uno de los modos de comunicación serial que soporta la interfaz USCI (interfaz de comunicación serial universal).

En modo síncrono, la USCI conecta el dispositivo con un sistema externo por medio de 3 o 4 pines: UCxSIMO, UCxSOMI, UCxCLK, y UCxSTE. El modo SPI es seleccionado cuando el bit UCSYNC está en 1, y el modo de SPI (3 o 4 pines) es seleccionado con los bits UCMODEx.

El modo SPI incluye:
- Largo de palabra de 7 a 8 bits
- Transmisión y recepción de datos LSB-first o MSB-first.
- Modo de operación 3 ó 4 bit para SPI
- Modo Maestro o Esclavo
- Shift registers para transmisión y recepción
- Registros buffer de recepción y transmisión
- Operación de transmisión y recepción continua
- Selección de polaridad de reloj y control de fase
- Frecuencia de reloj programable en modo maestro
- Capacidad de interrupción independiente para recepción y transmisión
- Modo esclavo en LPM4

En el modo SPI, los datos seriales son transmitidos y recibidos por múltiples dispositivos usando un reloj compartido que provee el dispositivo maestro.
Un pin adicional, UCxSTE controlado por el maestro, es proveído para habilitar un dispositivo a recibir y trasmitir datos.
Las siguientes señales son usadas para el intercambio de datos en modo SPI:

• UCxSIMO – slave in, master out
Master mode: UCxSIMO es la línea de salida de datos.
Slave mode: UCxSIMO es la línea de entrada de datos.
• UCxSOMI – slave out, master in
Master mode: UCxSOMI es la línea de entrada de datos.
Slave mode: UCxSOMI es la línea de salida de datos.
• UCxCLK – USCI SPI clock
Master mode: UCxCLK es una salida.
Slave mode: UCxCLK es una entrada.
• UCxSTE – habilitación para transmisión de esclavo.

La USCI en modo SPI soporta largos de palabras de 7 y 8 bits seleccionadas mediante el bit UC7BIT.

SPI 4-Pin en modo maestro

En el modo maestro de 4-pin, el registro UCxSTE se usa para prevenir conflictos con otro dispositivo maestro. Si se escriben datos al buffer UCxTXBUF mientras el dispositivo maestro está estancado en modo inactivo por UCxSTE, no se trasmitirá el dato hasta que UCxSTE cambie el estado a master-active. Si una transmisión activa es cancelada por una transición de UCxSTE al modo master-active, los datos deben ser reescritos en el buffer UCxTXBUF para ser transmitidos cuando UCxSTE cambie al estado master-active.
La señal de entrada UCxSTE no es usada en el modo *3-pin master mode*.

SPI 4-Pin en modo esclavo

En el modo esclavo de 4-pin, el registro UCxSTE proveído por el SPI maestro, es usado por el esclavo para habilitar la operación transmisión y recepción de datos.
La señal de entrada UCxSTE no es usada en el modo *3-pin slave mode.*

La operación de transmisión y recepción es indicada por la bandera UCBUSY = 1.

Transmisión habilitada

En modo maestro, escribir en el registro UCxTXBUF activa el bit de reloj generador, y la transmisión de datos comienza.
En modo esclavo, la transmisión comienza cuando un dispositivo maestro provee un reloj y, en modo 4-pin, cuando el registro UCxSTE está en estado esclavo-activo.

Recepción habilitada

El SPI recibe datos cuando la transmisión esta activa. Las operaciones de recepción y transmisión ocurren al mismo tiempo.

Interrupciones de transmisión

La bandera de interrupción UCTXIFG se setea por el transmisor para indicar que UCxTXBUF está pronto para aceptar datos. La respuesta a la interrupción es generada si UCTXIE y GIE están ambas seteadas. UCTXIFG se resetea automáticamente cuando un carácter se guarda en el buffer UCxTXBUF.

Interrupciones de recepción

La bandera de interrupción UCRXIFG se setea cada vez que un carácter es recibido y es cargado al registro UCxRXBUF. La respuesta a la interrupción es generada si UCRXIE y GIE están ambas seteadas.

Registros USCI SPI

Table 35-2. USCI_A SPI Mode Registers

Offset	Acronym	Register Name	Type	Access	Reset	Section
00h	UCAxCTLW0	USCI_Ax Control Word 0	Read/write	Word	0001h	
00h	UCAxCTL1	USCI_Ax Control 1	Read/write	Byte	01h	Section 35.4.2
01h	UCAxCTL0	USCI_Ax Control 0	Read/write	Byte	00h	Section 35.4.1
06h	UCAxBRW	USCI_Ax Bit Rate Control Word	Read/write	Word	0000h	
06h	UCAxBR0	USCI_Ax Bit Rate Control 0	Read/write	Byte	00h	Section 35.4.3
07h	UCAxBR1	USCI_Ax Bit Rate Control 1	Read/write	Byte	00h	Section 35.4.4
08h	UCAxMCTL	USCI_Ax Modulation Control	Read/write	Byte	00h	Section 35.4.5
0Ah	UCAxSTAT	USCI_Ax Status	Read/write	Byte	00h	Section 35.4.6
0Bh		Reserved - reads zero	Read	Byte	00h	
0Ch	UCAxRXBUF	USCI_Ax Receive Buffer	Read/write	Byte	00h	Section 35.4.7
0Dh		Reserved - reads zero	Read	Byte	00h	
0Eh	UCAxTXBUF	USCI_Ax Transmit Buffer	Read/write	Byte	00h	Section 35.4.8
0Fh		Reserved - reads zero	Read	Byte	00h	
1Ch	UCAxICTL	USCI_Ax Interrupt Control	Read/write	Word	0200h	
1Ch	UCAxIE	USCI_Ax Interrupt Enable	Read/write	Byte	00h	Section 35.4.9
1Dh	UCAxIFG	USCI_Ax Interrupt Flag	Read/write	Byte	02h	Section 35.4.10
1Eh	UCAxIV	USCI_Ax Interrupt Vector	Read	Word	0000h	Section 35.4.11

Figura 18: Registros USCI_A SPI del MSP430

Table 35-14. USCI_B SPI Mode Registers

Offset	Acronym	Register Name	Type	Access	Reset	Section
00h	UCBxCTLW0	USCI_Bx Control Word 0	Read/write	Word	0101h	
00h	UCBxCTL1	USCI_Bx Control 1	Read/write	Byte	01h	Section 35.5.2
01h	UCBxCTL0	USCI_Bx Control 0	Read/write	Byte	01h	Section 35.5.1
06h	UCBxBRW	USCI_Bx Bit Rate Control Word	Read/write	Word	0000h	
06h	UCBxBR0	USCI_Bx Bit Rate Control 0	Read/write	Byte	00h	Section 35.5.3
07h	UCBxBR1	USCI_Bx Bit Rate Control 1	Read/write	Byte	00h	Section 35.5.4
08h	UCBxMCTL	USCI_Bx Modulation Control	Read/write	Byte	00h	Section 35.5.5
0Ah	UCBxSTAT	USCI_Bx Status	Read/write	Byte	00h	Section 35.5.6
0Bh		Reserved - reads zero	Read	Byte	00h	
0Ch	UCBxRXBUF	USCI_Bx Receive Buffer	Read/write	Byte	00h	Section 35.5.7
0Dh		Reserved - reads zero	Read	Byte	00h	
0Eh	UCBxTXBUF	USCI_Bx Transmit Buffer	Read/write	Byte	00h	Section 35.5.8
0Fh		Reserved - reads zero	Read	Byte	00h	
1Ch	UCBxICTL	USCI_Bx Interrupt Control	Read/write	Word	0200h	
1Ch	UCBxIE	USCI_Bx Interrupt Enable	Read/write	Byte	00h	Section 35.5.9
1Dh	UCBxIFG	USCI_Bx Interrupt Flag	Read/write	Byte	02h	Section 35.5.10
1Eh	UCBxIV	USCI_Bx Interrupt Vector	Read	Word	0000h	Section 35.5.11

Figura 19: Registros USCI_B SPI del MSP430 [16]

[16] Ambas figuras extraídas de MSP430x5xx / MSP430x6xx Family User's Guide", Texas Instruments, 2008.

I2C

En modo I2C, el módulo USCI provee una interfaz entre el dispositivo y los dispositivos I2C-compatibles conectados mediante el bus serie I2C.

El modo I2C incluye:

- Modos de direccionamiento de 7 y 10 bits
- Llamada general
- START/RESTART/STOP
- Modo multi-master de transmisión/recepción
- Modo esclavo de transmisión/recepción
- Soporte de modo estándar hasta 100 kbps y rápido hasta 400 kbps
- Frecuencia UCxCLK programable en modo maestro
- Diseño para bajo consumo
- Detección de START en recepción de esclavo para despertarse de cualquiera de los modos LPMx (Low Power Mode).
- Operación de esclavo en LPM4 (Low Power Mode)

El modo I2C soporta cualquier dispositivo maestro/esclavo compatible con I2C.
Cada dispositivo I2C es reconocido con una dirección única y puede operar como transmisor y receptor. El maestro inicia la transmisión de datos y genera la señal de reloj SCL. Cualquier dispositivo direccionado por el maestro es considerado como esclavo.

La comunicación I2C se realiza a través del pin de datos seriales (SDA) y el pin de reloj serial (SCL). Ambos son bidireccionales y deben ser conectados a una fuente de voltaje positiva usando un resistor de pullup.

Comunicación I2C

El maestro genera un pulso de reloj por cada bit de datos transferido. El modo I2C opera con datos de bytes. Primero se envía el MSB. El primer byte luego de la condición START consiste en 7-bits de dirección del esclavo y el bit R/W. Cuando R/W=0, el maestro transmite datos al esclavo. Cuando R/W=1, el maestro recibe datos del esclavo. El bit ACK es enviado desde el receptor luego de cada byte on the ninth SCL clock.

Las condiciones START y STOP son generadas por el maestro de la siguiente manera:
- La condición de START es una transición high-to-low en la línea SDA mientras SCL está en nivel alto.
- La condición STOP es una transición low-to-high en la línea SDA mientras SCL está en nivel alto.
- El bit de ocupado UCBBUSY se setea luego de un START y se borra luego de un STOP.

Condición de RESTART

El dispositivo maestro puede cambiar la dirección de datos sin necesidad de usar stop and transfer, simplemente usando RESTART (condición de START repetitiva). Luego de un RESTART la dirección del esclavo es reenviada con la nueva dirección de datos.

Modo Esclavo

El módulo USCI se configura en modo esclavo I2C seleccionando el modo I2C con UCMODEx = 11 y UCSYNC = 1 y borrando el bit UCMST.
Inicialmente, el módulo USCI debe estar configurado en modo recepción borrando el bit UCTR para recibir la dirección I2C. Luego de esto las operaciones de transmisión y recepción son controladas automáticamente, dependiendo de los bits R/W recibidos con la dirección del esclavo.

La dirección del esclavo USCI se programa con el registro UCBxI2COA.
Cuando UCA10 = 0, el modo de direccionamiento de 7-bits queda seleccionado.
Cuando UCA10 = 1, el modo de direccionamiento de 10-bits queda seleccionado.
El bit UCGCEN selecciona si el esclavo responde a una llamada general.
Cuando se detecta una condición de START en el bus, el módulo USCI recibe la dirección de transmisión y la compara con su dirección almacenada en UCBxI2COA. La bandera UCSTTIFG se setea cuando la dirección recibida coincide con la dirección del esclavo USCI.

Modo Esclavo – transmisión

La transmisión en modo esclavo queda habilitada cuando la dirección del esclavo transmitida por el maestro es idéntica a la dirección almacenada, setenándose el bit R/W. El esclavo que transmite borra los datos del puerto serie del SDA con un pulso de reloj generado por el dispositivo maestro. El dispositivo esclavo no genera el reloj, pero esto hace que se mantenga SCL en nivel bajo mientras la intervención del CPU es requerida luego que un byte fue transmitido. Si el dispositivo maestro requiere datos del esclavo, el módulo USCI se configura automáticamente como un transmisor y se setean UCTR y UCTXIFG. La línea SCL se mantiene baja hasta que el primer dato a ser transmitido es escrito en el buffer de transmisión UCBxTXBUF.
Luego que la dirección es reconocida, se borra la bandera UCSTTIFG, y el dato es transmitido.
En el instante en que el dato es transferido al shift register, UCTXIFG se setea nuevamente.
Luego que el dato es reconocido por el dispositivo maestro, el próximo dato escrito en el buffer UCBxTXBUF es transmitido, o si el buffer está vacío, el bus queda estancado durante el ciclo de reconocimiento manteniendo el SCL en nivel bajo hasta que un nuevo dato sea escrito en el buffer UCBxTXBUF.
Si el maestro envía un NACK seguido de una condición de STOP, la bandera UCSTPIFG se setea.
Si el NACK es seguido de un RESTART, la máquina de estados del USCI I2C retorna a su estado de *address-reception.*

Modo esclavo – recepción

La recepción en modo esclavo queda habilitada cuando la dirección del esclavo transmitida por el maestro es idéntica a su dirección y un bit bajo de R/W es recibido. En el modo esclavo de recepción, los datos seriales del SDA son borrados con los pulsos de reloj generados por el dispositivo maestro. El dispositivo esclavo no genera pulsos de reloj, pero esto puedo mantener la señal SCL en nivel bajo si la intervención del CPU es requerida luego que un byte ha sido recibido.
Si el esclavo debe recibir datos del maestro, el módulo USCI se configura automáticamente como un receptor y la señal UCTR se resetea. Luego que se recibe el primer byte de datos, se setea la bandera de interrupción de recepción de datos UCRXIFG. El módulo de recepción USCI reconoce automáticamente el dato recibido y se prepara para recibir el próximo byte de datos.

Si el dato previo no fue leído del buffer UCBxRXBUF al final de la recepción, el bus queda atascado manteniendo la señal SCL en nivel bajo. En el instante en que el buffer es leído, el nuevo dato es transferido a UCBxRXBUF, se envía un reconocimiento al dispositivo maestro, y de queda esperando al próximo dato.
Para evitar perdida de datos, el buffer UCBxRXBUF debe ser leído antes que sea seteada la bandera UCTXNACK (causa que un NACK sea transmitido durante el próximo ciclo de reconocimiento).
Si el dispositivo maestro genera un RESTART, la máquina de estados del USCI I2C retorna a su estado de address-reception.

Modo Maestro

El módulo USCI se configura en modo maestro I2C seleccionando el modo I2C con UCMODEx = 11 y UCSYNC = 1 y sesteando el bit UCMST.
Cuando el maestro es una parte de un sistema multi-master, UCMM debe estar seteado y su dirección debe estar programada en el registro UCBxI2COA. Cuando UCA10 = 0, el modo de direccionamiento de 7 bits queda seleccionado. En el caso de UCA10 = 1, el modo de direccionamiento de 10 bits quedará seleccionado. El bit UCGCEN selecciona si el módulo SCI debe responder o no a una llamada general.

Modo maestro – transmisión

Luego de la inicialización, el modo maestro de transmisión queda habilitado al escribir la dirección de esclavo deseada al registro UCBxI2CSA, seleccionando el tamaño de direcciones de esclavos con el bit UCSLA10, seteando UCTR para modo de transmisión, y seteando UCTXSTT para generar la condición de START.
El módulo USCI chequea si el bus está habilitado, genera la condición START, y transmite la dirección del esclavo. El bit UCTXIFG queda seteado cuando la condición de START es generada y el primer dato a ser transmitido puede ser escrito en el buffer UCBxTXBUF.
En el instante en que el esclavo reconoce la dirección, el bit UCTXSTT es borrado.

Los datos escritos en el buffer UCBxTXBUF se transmiten si el arbitraje no se pierde durante la transmisión de la dirección del esclavo.
El bit UCTXIFG se setea nuevamente al transferirse el dato del buffer al shift register. Si no hay datos cargados en el buffer antes del ciclo de reconocimiento, el bus queda estancado durante el ciclo de reconocimiento con la señal SCL en nivel bajo hasta que algún dato sea escrito en el buffer UCBxTXBUF.
Al setear UCTXSTP se genera una condición de STOP luego del siguiente ciclo de reconocimiento del esclavo. Si UCTXSTP esta seteado durante la transmisión de la dirección del esclavo o mientras el módulo USCI espera por datos a ser escritos en UCBxTXBUF, una condición de STOP será generada, aunque no se hayan transmitido datos al esclavo. Cuando se transmite un solo byte de datos, el bit UCTXSTP debe estar seteado mientras el byte está siendo transmitido, sin escribir nuevos datos en el buffer.
Cuando un dato es transferido desde el buffer al shift register, UCTXIFG queda seteado, indicando que una transmisión de datos ha comenzado, y el bit UCTXSTP debe estar seteado.
Si el esclavo no reconoce el dato transmitido, la bandera de interrupción por no-reconocimiento UCNACKIFG queda seteada. El maestro debe reaccionar con una condición de STOP o un RESTART. Si el dato está escrito en el buffer UCBxTXBUF, se lo descarta. Si el dato debe ser transmitido luego de la condición de RESTART, será escrito en el buffer nuevamente.

Modo maestro – recepción

Luego de la inicialización, el modo maestro de recepción se habilita al escribir la dirección deseada del esclavo en el registro UCBxI2CSA, seleccionando el tamaño del direccionamiento de esclavos con el bit UCSLA10, borrando UCTR para el modo de recepción, y seteando UCTXSTT para generar una condición de START.
El módulo USCI chequea si el bus está disponible, genera la condición de START, y transmite la dirección del esclavo. En el instante en que el esclavo reconoce la dirección, se resetea UCTXSTT.
Luego del reconocimiento de la dirección del esclavo, el primer byte de datos del esclavo es recibido y reconocido y la bandera UCRXIFG se setea.
Los datos son recibidos hasta que UCTXSTP o UCTXSTT no están seteados. Si el buffer UCBxRXBUF no es leído, el maestro sostiene el bus durante la recepción del último bit de datos y hasta que el buffer sea leído.
Si el esclavo no reconoce la dirección de transmisión, la bandera de interrupción por no-reconocimiento UCNACKIFG se setea.
El maestro debe reaccionar con una condición de STOP o RESTART.
Luego de setear UCTXSTP, un NACK seguido de una condición de STOP es generado luego de la recepción de datos desde el esclavo, o inmediatamente si el módulo USCI está esperando por un dato del buffer UCBxRXBUF a ser leído.

Interrupciones de USCI en modo I2C

El módulo USCI tiene solo un vector de interrupciones compartido para transmisión, recepción, y el cambio de estados.
USCI_Ax y USCI_Bx no comparten el mismo vector de interrupciones.
Cada bandera de interrupción tiene su propio bit de habilitación de interrupción.
Cuando una interrupción queda habilitada y el bit GIE se setea, la bandera de interrupción genera una petición de interrupción.

Operación de interrupciones de transmisión I2C

La bandera de interrupción UCTXIFG se setea por el dispositivo a transmitir para indicar que el buffer UCBxTXBUF esta pronto para aceptar otro carácter. Una petición de interrupción se genera si UCTXIE y GIE están siempre seteados. UCTXIE se resetea automáticamente cuando un carácter se escribe en el buffer UCBxTXBUF o si un NACK se recibe. UCTXIFG se setea cuando UCSWRST = 1 y el modo I2C se ha seleccionado.

Operación de interrupciones de recepción I2C

La bandera de interrupción UCTXIFG se setea cuando se recibe un carácter y se carga al buffer UCBxRXBUF. Una petición de interrupción se genera si UCRXIE y GIE están siempre seteados.

Registros I2C

Table 36-2. USCI_B Registers

Offset	Acronym	Register Name	Type	Access	Reset	Section
00h	UCBxCTLW0	USCI_Bx Control Word 0	Read/write	Word	0101h	
00h	UCBxCTL1	USCI_Bx Control 1	Read/write	Byte	01h	Section 36.4.2
01h	UCBxCTL0	USCI_Bx Control 0	Read/write	Byte	01h	Section 36.4.1
06h	UCBxBRW	USCI_Bx Bit Rate Control Word	Read/write	Word	0000h	
06h	UCBxBR0	USCI_Bx Bit Rate Control 0	Read/write	Byte	00h	Section 36.4.3
07h	UCBxBR1	USCI_Bx Bit Rate Control 1	Read/write	Byte	00h	Section 36.4.4
0Ah	UCBxSTAT	USCI_Bx Status	Read/write	Byte	00h	Section 36.4.5
0Bh		Reserved - reads zero	Read	Byte	00h	
0Ch	UCBxRXBUF	USCI_Bx Receive Buffer	Read/write	Byte	00h	Section 36.4.6
0Dh		Reserved - reads zero	Read	Byte	00h	
0Eh	UCBxTXBUF	USCI_Bx Transmit Buffer	Read/write	Byte	00h	Section 36.4.7
0Fh		Reserved - reads zero	Read	Byte	00h	
10h	UCBxI2COA	USCI_Bx I2C Own Address	Read/write	Word	0000h	Section 36.4.8
12h	UCBxI2CSA	USCI_Bx I2C Slave Address	Read/write	Word	0000h	Section 36.4.9
1Ch	UCBxICTL	USCI_Bx Interrupt Control	Read/write	Word	0200h	
1Ch	UCBxIE	USCI_Bx Interrupt Enable	Read/write	Byte	00h	Section 36.4.10
1Dh	UCBxIFG	USCI_Bx Interrupt Flag	Read/write	Byte	02h	Section 36.4.11
1Eh	UCBxIV	USCI_Bx Interrupt Vector	Read	Word	0000h	Section 36.4.12

Figura 20: Registros I2C del MSP430 [17]

UART

En modo UART, la USCI transmite y recibe caracteres a una tasa de bits asíncrona a los demás dispositivos. Las funciones de transmisión y recepción usan la misma tasa de transferencia. En modo asíncrono, el módulo USCI_Ax conecta el dispositivo a un sistema externo a través de dos pines externos, UCAxRXD y UCAxTXD. El modo UART se selecciona cuando el bit UCSYNC está en cero.

El modo UART incluye:
Datos de 7 o 8 bits, con paridad par, impar y sin paridad
Registros de transmisión y recepción independientes
Registros buffer de transmisión y recepción separados
Transmisión y recepción de datos en modo LSB-first y MSB-first
Detección de flancos de transmisión para que el receptor despierte del modo LPMx
Baud rate programable con soporte de modulación para baud rates fraccionadas
Banderas de estado de detección de errores y supresión
Banderas de estado de detección de direcciones
Capacidad independiente de interrupción de recepción y transmisión

Formato de carácter

El formato de carácter consiste de un bit de start, 7 o 8 bits de datos, un bit de paridad par, impar o de no paridad, un bit de direcciones, y uno o dos bits de parada.

Modo de comunicación asíncrona

Cuando 3 o más dispositivos se comunican, el módulo USCI soporta los formatos de comunicación multiprocesos de tipo idle-line and adress-bit.

[17] Figura extraída de MSP430x5xx / MSP430x6xx Family User's Guide, Texas Instruments, 2008.

Formato de multiprocesos idle-line

Los bloques de datos son separados por un tiempo inactivo (idle-time) puesto en las líneas de transmisión y recepción. Se detecta una línea inactiva en recepción cuando se reciben 10 o más bits en alto contiguos luego del bit de parada del carácter.
El primer bit recibido luego de un periodo inactivo es un carácter de direcciones.
El bit UCDORM se usa para controlar la recepción de datos en el formato de línea inactiva. Cuando UCDORM = 1, todos los caracteres que no son direcciones son ensamblados pero no transferidos al buffer UCAxRXBUF, y no se generan interrupciones. Cuando se recibe un carácter de direcciones, el carácter es transferido al buffer UCAxRXBUF, se setea UCRXIFG, y alguna bandera de error aplicable se setea cuando UCRXEIE = 1. Cuando UCRXEIE = 0 y el carácter de direcciones es recibido pero presenta un error de paridad o de trama, el carácter no se transfiere al buffer UCAxRXBUF y UCRXFIG no se setea.

Si se recibe una dirección, el software usuario puede validar la dirección y debe resetear UCDORM para continuar recibiendo datos. Si UCDORM se mantiene seteado, solo los caracteres de dirección son recibidos.

Transmisión de una Idle Frame

El procedimiento a continuación envía una trama inactiva para indicar que un carácter de direcciones le sigue:
Se setea UCTXADDR, luego se escribe la dirección del carácter al buffer UCAxTXBUF. UCAxTXBUF debe estar listo para recibir nuevos datos (UCTXIFG=1).

Se escriben los datos deseados al registro UCAxTXBUF. UCAxTXBUF debe estar listo para guardar nuevos datos (UCTXIFG=1). Los datos escritos en el buffer son transferidos al shift register y transmitidos tan pronto como el shift register queda listo para recibir nuevos datos.

La línea inactiva no debe exceder el espacio entre direcciones y transmisión de datos o entre transmisiones de datos. De otra manera, la transmisión de datos será interpretada como una dirección.

Formato de direcciones de bit de Multiprocesador

Cuando UCMODEx = 10, queda seleccionado el formato de direcciones de bit de Multiprocesador. Cada carácter procesado contiene un extra bit que es usado como un indicador de direcciones. El primer carácter en un bloque de caracteres contiene el bit de direcciones seteado, lo que indica que el carácter es una dirección.

El bit UCDORM se usa para controlar la recepción de datos en el formato adress-bit. Cuando UCDROM está seteado, los caracteres de datos con el bit de direcciones igual a 0 son aceptados por el receptor pero no se transfieren al buffer UCAxRXBUF y no se generan interrupciones. Cuando se recibe un carácter con el bit de direcciones en 1, el carácter se transfiere a UCAxRXBUF, se setea UCRXIFG, y se levanta la bandera de error correspondiente UCRXEIE = 1. Cuando UCRXEIE = 0 y se recibe el carácter con el bit de direcciones en 1 pero con un error de trama o paridad, el carácter no se transfiere al buffer y UCRXIFG no queda seteada.

Si se recibe una dirección, el software usuario puede validar la dirección reseteando UCDORM para continuar recibiendo datos. Si UCDROM se mantiene encendido, solo los caracteres de direcciones con el bit de direcciones en 1 son recibidos.

Cuando UCDORM = 0, todos los caracteres recibidos setean la bandera de interrupción de recepción UCRXIFG. Si UCDORM está en 0 durante la recepción de carácter, la bandera de interrupción de recepción se setea al finalizar la recepción.

Fin de recepción y generación

Cuando UCMODEx = 00, 01, o 10, el receptor detecta un fin de palabra cuando todos los bits de datos, paridad y Stop están bajos. Cuando se detecta un fin, se setea el bit UCBRK.

Para transmitir un fin, se setea el bit UCTXBRK, luego se escribe 0h a UCAxTXBUF. UCAxTXBUF debe estar lista para recibir nuevos datos. Esto genera un fin cuando todos los bits están en nivel bajo.

Tasa de transferencia de bits

La UART ofrece un modo de detección automática de baudios a través de la transmisión de un campo break/sync.
Puede generar también frecuencias de transmisión estándar a través de fuentes no estándar mediante la selección de dos modos de operación.

Registros

Table 34-6. USCI_A UART Mode Registers

Offset	Acronym	Register Name	Type	Access	Reset	Section
00h	UCAxCTLW0	USCI_Ax Control Word 0	Read/write	Word	0001h	
00h	UCAxCTL1	USCI_Ax Control 1	Read/write	Byte	01h	Section 34.4.2
01h	UCAxCTL0	USCI_Ax Control 0	Read/write	Byte	00h	Section 34.4.1
06h	UCAxBRW	USCI_Ax Baud Rate Control Word	Read/write	Word	0000h	
06h	UCAxBR0	USCI_Ax Baud Rate Control 0	Read/write	Byte	00h	Section 34.4.3
07h	UCAxBR1	USCI_Ax Baud Rate Control 1	Read/write	Byte	00h	Section 34.4.4
08h	UCAxMCTL	USCI_Ax Modulation Control	Read/write	Byte	00h	Section 34.4.5
09h		Reserved - reads zero	Read	Byte	00h	
0Ah	UCAxSTAT	USCI_Ax Status	Read/write	Byte	00h	Section 34.4.6
0Bh		Reserved - reads zero	Read	Byte	00h	
0Ch	UCAxRXBUF	USCI_Ax Receive Buffer	Read/write	Byte	00h	Section 34.4.7
0Dh		Reserved - reads zero	Read	Byte	00h	
0Eh	UCAxTXBUF	USCI_Ax Transmit Buffer	Read/write	Byte	00h	Section 34.4.8
0Fh		Reserved - reads zero	Read	Byte	00h	
10h	UCAxABCTL	USCI_Ax Auto Baud Rate Control	Read/write	Byte	00h	Section 34.4.11
11h		Reserved - reads zero	Read	Byte	00h	
12h	UCAxIRCTL	USCI_Ax IrDA Control	Read/write	Word	0000h	
12h	UCAxIRTCTL	USCI_Ax IrDA Transmit Control	Read/write	Byte	00h	Section 34.4.9
13h	UCAxIRRCTL	USCI_Ax IrDA Receive Control	Read/write	Byte	00h	Section 34.4.10
1Ch	UCAxICTL	USCI_Ax Interrupt Control	Read/write	Word	0000h	
1Ch	UCAxIE	USCI_Ax Interrupt Enable	Read/write	Byte	00h	Section 34.4.12
1Dh	UCAxIFG	USCI_Ax Interrupt Flag	Read/write	Byte	00h	Section 34.4.13
1Eh	UCAxIV	USCI_Ax Interrupt Vector	Read	Word	0000h	Section 34.4.14

Figura 21: Registros UART del MSP430 [18]

[18] Figura extraída de MSP430x5xx / MSP430x6xx Family User's Guide, Texas Instruments, 2008.

TIMER A

El timer_A es un timer/contador de 16-bits con 7 registros capture/compare. El timer_A cuenta con extensas capacidades de interrupción. La interrupción puede ser generada desde el contador debido a una condición de overflow y desde cada uno de los registros de capture/compare.
El timer_A incluye:

Contador y timer asíncronos de 16-bit con 4 modos de operación
Selección y configuración de reloj externo
Hasta 7 registros capture/compare configurables
Salidas configurables con pulso de modulación PWM
Latcheo asíncrono de entrada y salida
Registro de vector de interrupción para decodificar rápidamente rodas las interrupciones del Timer_A

Operación de Timer_A

El módulo de Timer_A se configura con un software de usuario.
Los diferentes modos de configuración del timer se presentan a continuación.

16-bit Timer Counter

El registro timer/counter de 16 bits, TAxR, incrementa o decrementa (dependiendo el modo de operación) con cada flanco de subida de las señal de reloj. TAxR puede ser leído o escrito con el software. Adicionalmente, el timer puede generar una interrupción cuando desborda.

Selección y divisor de reloj externo

El reloj del timer puede ser generado por ACLK, SMCLK, o externamente vía TAxCLK o INCLK. La fuente de reloj se selecciona mediante los bits TASSEL. La fuente de reloj seleccionada debe pasar directamente al timer o por el divisor de 2, 4 o 8 usando los bits de ID. La fuente de reloj seleccionada puede ser también dividida por 2, 3, 4, 5, 6, 7, o 8 usando los bits TAIDEX.

Arranque del Timer

El timer debe ser reseteado o arrancado de la siguiente manera:
El timer cuenta cuando MC > 0 y la fuente de reloj esta activa.
El timer debe parar escribiendo cero a TAxCCR0. Luego el timer debe ser reseteado escribiendo un valor diferente a 0 a TAxCCR0.

Modo de control de Timer

EL timer tiene 4 modos de operación: stop, up, continuado, y up/down. El modo de operación es seleccionado con los bits MC.

Up Mode

El modo Up se usa si el periodo del timer es diferente a 0FFFFh cuentas. El timer cuenta repetidamente ascendiendo hasta llegar al valor del registro comparador TAxCCR0, el cual

define el periodo. Cuando el valor del timer alcanza el valor de TAxCCR0, el timer resetea la cuenta comenzando desde cero.

Cambiando el periodo del registro

Al cambiar TAxCCR0 mientras el timer está corriendo, si el nuevo periodo es mayor o igual que el periodo anterior o mayor que el valor de conteo presente, el timer cuenta hasta el nuevo periodo. Si el nuevo periodo es menor al valor que presenta el contador, el timer vuelve a cero. Sin embargo una cuenta adicional ocurre cuando el contador vuelve a cero.

Modo Continuo

En el modo continuo, el timer cuenta hasta el valor 0FFFFh repetidamente y se resetea desde el cero. La bandera de interrupción TAIFG se setea cuando el contador pasa de 0FFFFh a cero.

Modo Up/Down

El modo Up/Down se usa si el periodo del timer debe ser diferente a 0FFFFh cuentas.
El timer cuenta repetidamente hasta alcanzar el valor del registro comparador TAxCCR0 y pasa a decrementar hasta llegar a cero. El periodo es el doble del valor cargado en TAxCCR0.

La bandera de interrupción CCIFG se setea cuando el contador del timer llega al valor de TAxCCR0, y TAIFG se setea cuando el timer completa la cuenta descendente al llegar al valor cero.

Bloque Capture/Compare

El Timer_A contiene hasta 7 bloques capture(compare, TAxCCRn (con n=0 a 7)

Modo Capture

El modo captura queda seleccionado cuando CAP = 1. Este modo se usa para medir tiempo.
Si una captura ocurre:
El valor del Timer se copia al registro TAxCCRn
Se setea la bandera de interrupción CCIFG

Modo Compare

El modo de comparación es utilizado para interrumpir en intervalos de tiempo específicos y generar señales de salida PWM. Cuando TAxR cuenta hasta el valor de TAxCCRn donde n representa el número de registro usado.

Unidad de salida

Cada bloque capture/compare contiene una unidad de salida. La unidad de salida es utilizada para generar señales de salida, por ejemplo señales PWM.

Interrupciones del Timer_A

Son dos los vectores de interrupción que están asociados con el módulo Timer_A de 16 bits:

- Vector de interrupción TAxCCR0 de la bandera TAxCCR0 CCIFG

- Vector de interrupción TAxIV para todas las otras banderas CCIFG y TAIFG

Interrupción de TAxCCR0

La bandera de interrupción TAxCCR0 CCIFG tiene la más alta prioridad de interrupción del Timer_A y tiene un vector de interrupción dedicado.

TAxIV, Vector Generador de Interrupciones

El registro de vector de interrupciones TAxIV se usa para determinar que bandera (TAxCCIFG CCIFG y TAIFG) solicita la interrupción.
La interrupción que tiene la prioridad genera un número en el registro TAxIV. Este número es agregado al *Program Counter* para ejecutar de manera automática la rutina pendiente.

TIMER B

Similitudes y diferencias respecto al Timer_A

- El largo del Timer_B es programable pudiendo ser de 8, 19, 12 o 16 bits
- Los registros TBxCCRn del Timer_B son doublé-buffered y pueden ser agrupados
- Todas las salidas del Timer_B pueden ponerse en estado de alta impedancia
- La función del bit SCCI no está implementada en el Timer_B

Anexo 2: Conexión de la placa DE0 a la PC

La conexión entre la DE0 y la computadora se realiza por medio de un cable USB, el cual requiere para su funcionamiento una previa instalación de drivers. A continuación se detallan los pasos a seguir en Windows XP/Vista/7.

Paso 1: Reconocimiento del nuevo hardware conectado

Al conectar la placa a la computadora, esta última detectará el nuevo hardware conectado al puerto USB pero será incapaz de reconocerlo si no se poseen instalados los correspondientes drivers. Windows arrojará entonces la ventana de nuevo hardware encontrado, mostrada en la figura 21.

Figura 22: Nuevo hardware encontrado

Paso 2: Especificar la ruta de los drivers

Debido a que el controlador que se desea no está disponible en la web de Windows Update, se debe elegir "No, not this time" y a continuación dar click en "Next". Esto lleva a la ventana que se muestra en la figura 22, donde se seleccionará la opción de especificar la locación y un nuevo clik en "Next" nos llevará a la ventana de la figura 23.

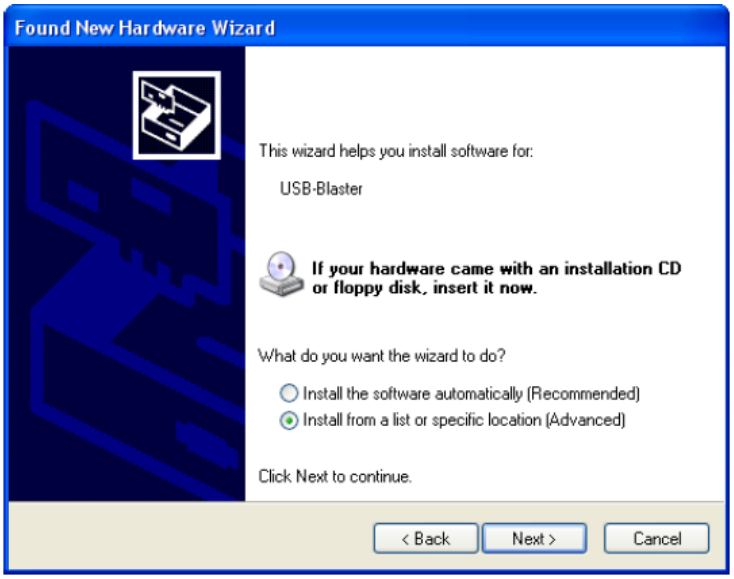


Figura 23: elegir desde dónde instalar

Paso 3. Seleccionar apropiadamente la versión del driver a instalar

A continuación se selecciona la opción de buscar el mejor controlador en estas ubicaciones (Search for the best driver in these locations) y se da click en "Browse".

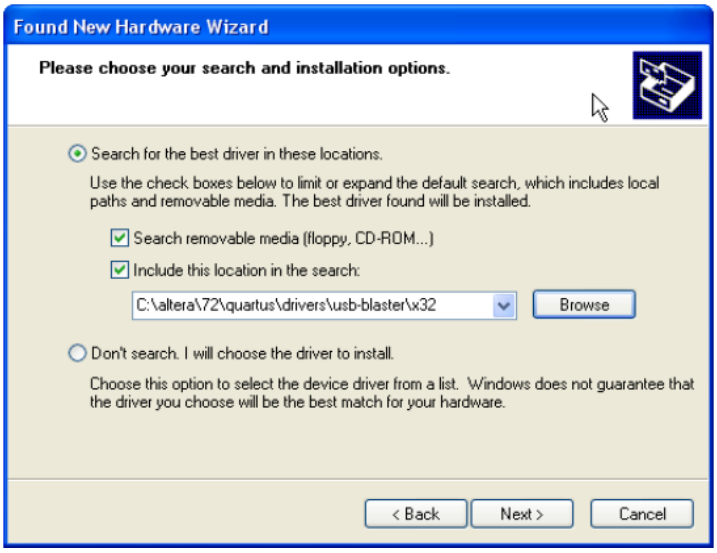


Figura 24: Elegir ubicación para búsqueda de drivers

Aparecerá entonces el cuadro de pop-up en la Figura 24, donde se debe navegar hasta encontrar la ruta. Haga click en “Ok” y luego retornará a la figura 23, donde se deberá se clikear en “Next para iniciar la instalación.

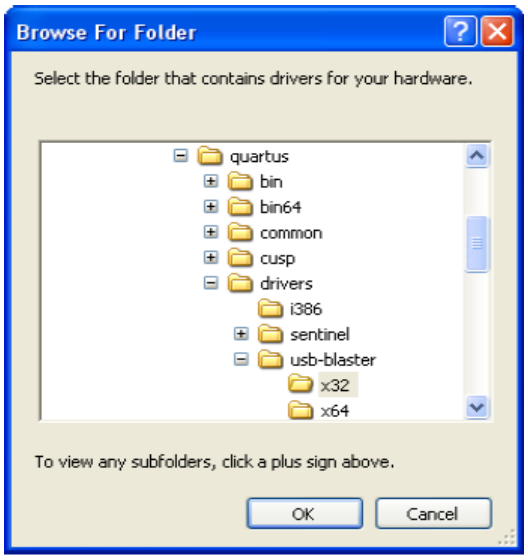


Figura 25: Menú de navegación

Al finalizar emergerá la ventana de la figura 25, donde se nos notifica que el driver no ha superado la prueba del logotipo de Windows. Se dará click en continuar de todos modos.

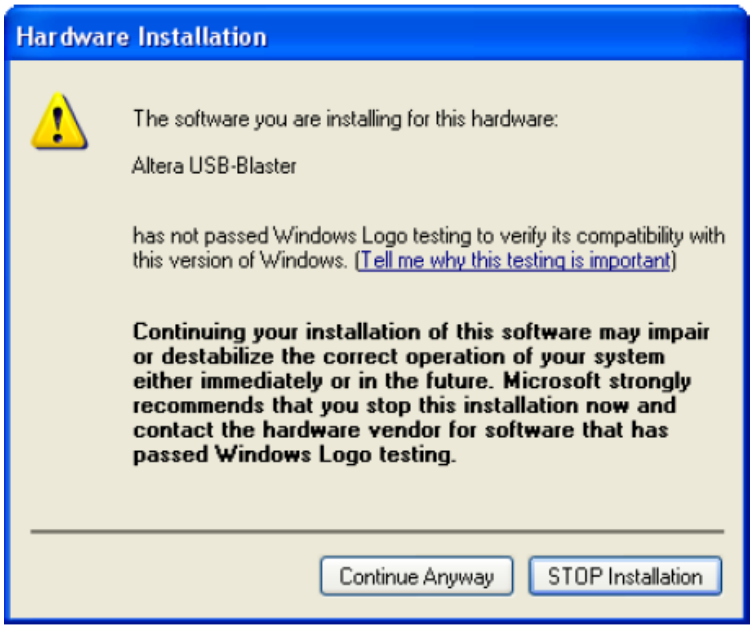


Figura 26: Notificación de incompatibilidad con la versión de Windows

Paso 4. El cable USB está listo para ser utilizado

El driver quedará instalado como se muestra en la figura 26. Tras hacer click en “Finalizar” se podrá comenzar a utilizar la placa DE0.

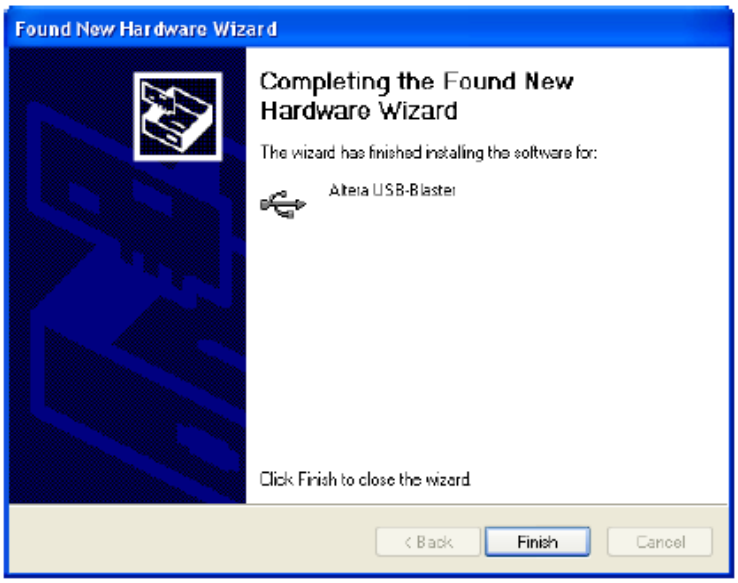


Figura 27: Fin de la instalación

Anexo 3: Instalación de driver del cable serial RS232

1. Se procede a iniciar el proceso de instalación del driver, dando doble click o ENTER al archivo “PL-2303 Driver Installer”**.** Aparece inmediatamente la ventana mostrada a continuación.

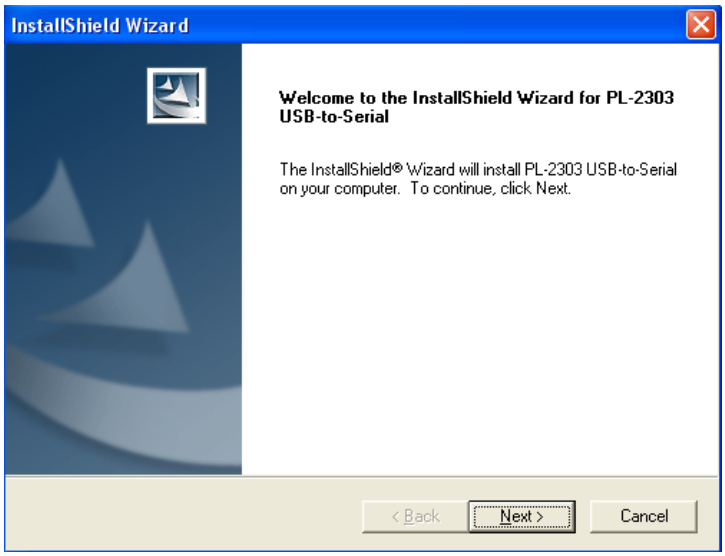


Figura 28: Bienvenido a la instalación USB-to-Serial

2. A continuación se da click en la opción “Next”, lo cual iniciará el proceso de instalación. Al finalizar aparecerá una ventana donde se nos notifica que la instalación ha finalizado y que en caso de tener el cable conectado procedamos a la desconexión del mismo para luego volver a conectarlo.

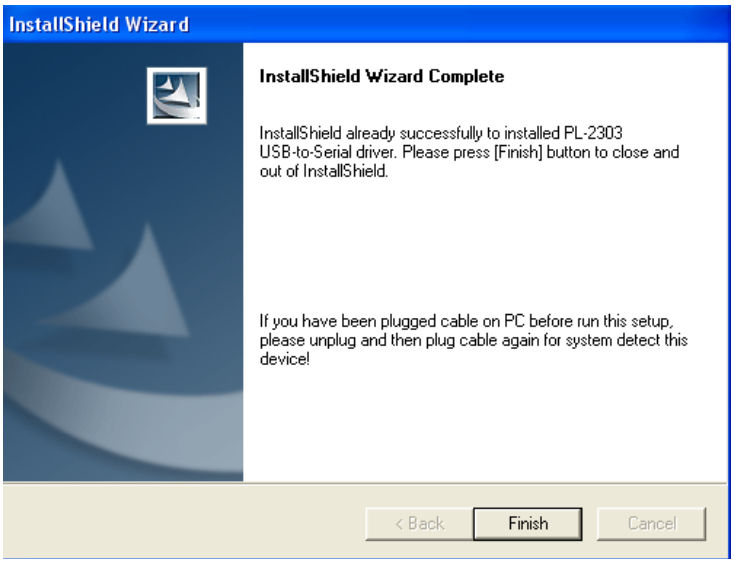


Figura 29: Instalación exitosa

3. Por último nos cercioramos de que en el administrador de dispositivos de Windows el cable haya sido reconocido por el sistema y este le haya asignado un COM para su utilización.

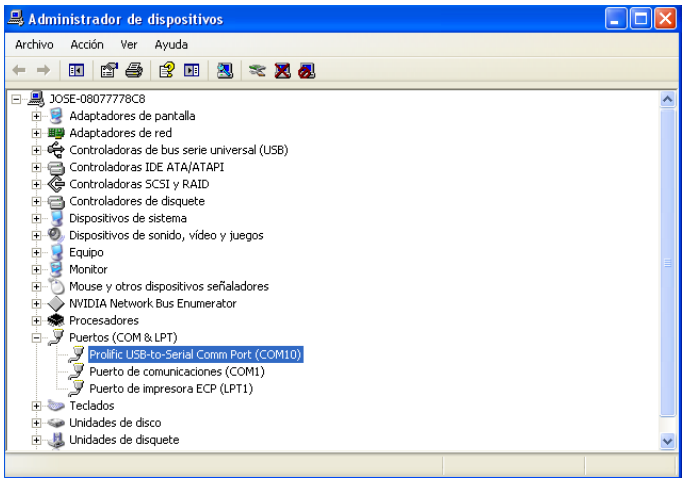


Figura 30: Administrador de dispositivos

En algunos sistemas operativos puede ocurrir que una vez finalizada la instalación aparezca el siguiente mensaje:

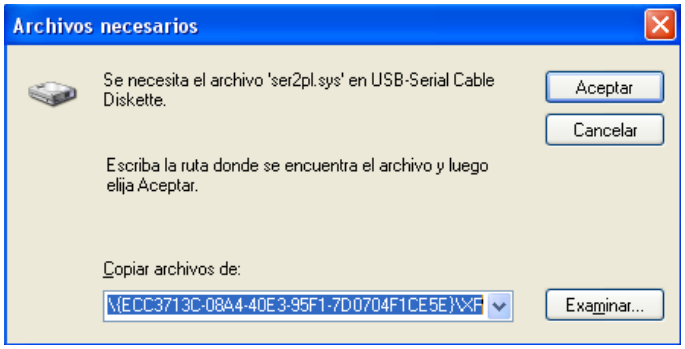


Figura 31: Especificar ruta

Para resolverlo basta con especificar la siguiente ruta del CD de instalación provisto por el fabricante del cable.

$$E{:}\backslash PL2303\ all\ driver\backslash 98\_ME\_2K$$

Tras dar click en “Aceptar” aparecerá aviso de que el software que intentamos instalar no supera el logotipo de Windows. Damos en continuar y culminamos la instalación.

Figura 32: Notificación de incompatibilidad con la versión de Windows

Anexo 4: Herramientas de software del openMSP430

Teniendo en cuenta las capacidades de depuración (conexión serie) que presenta el openMSP430, el diseñador del módulo principal ha adjuntado una serie de programas que facilitan distintas tareas. Los mismos son:

- **openMSP430-loader:** es un programa que funciona en línea de comandos y permite cargar archivos *.elf* y *.hex* a la memoria de programas.
- **openMSP430-minidebug:** se trata de un debugger extremadamente simple con una GUI (Graphical User Interface) muy intuitiva.
- **openMSP430-gdbproxy:** consiste en un servidor GDB-Proxy para ser utilizado junto con el debugger MSP430-GDB y el Eclipse.

Estas herramientas de software se han desarrollado en Tcl/Tk y fueron probadas con éxito en Windows (XP/7/8). Cabe destacar que el diseñador también logró utilizarlas en Linux sin especificar la distribución utilizada.

Para ejecutar estos scripts escritos en tcl es necesario un intérprete. En nuestro caso se utilizó Tcl/Tk. Para conectar el PC a la interfaz serial de depuración openMSP430 se requiere una UART o cable serial I2C. En nuestro caso optamos por la interfaz UART debido a las facilidades que nos ofrecía la placa DE0.

A continuación se presenta una descripción de cada una de las herramientas de software mencionadas anteriormente.

openMSP430-loader

Este simple programa permite cargar la memoria de programa del openMSP430 con un archivo ejecutable *(.elf* o *.hex*) que será proporcionado como argumento en la consola de comandos cuando se llame al programa. Si bien se recomienda utilizar el programa conjuntamente con "make" para cargar automáticamente el archivo ejecutable en la memoria de programa luego de una compilación, en nuestro caso fue utilizado por separado.

El programa puede ser llamado con la siguiente sintaxis:

```
USAGE   : openmsp430-loader.tcl [-device   <communication port>]
                                [-adaptor  <adaptor type>]
                                [-speed    <communication speed>]
                                [-i2c_addr <cpu address>]          <elf/ihex-file>

DEFAULT : <communication port>  = /dev/ttyUSB0
          <adaptor type>        = uart_generic
          <communication speed> = 115200 (for UART) / I2C_S_100KHZ (for I2C)
          <core address>        = 42

EXAMPLES: openmsp430-loader.tcl -device /dev/ttyUSB0 -adaptor uart_generic -speed 9600  leds.elf

          openmsp430-loader.tcl -device COM2:        -adaptor i2c_usb-iss  -speed I2C_S_100KHZ
                                -i2c_addr 75 ta_uart.ihex
```

Figura 33: Llamando al openMSP430-loader

A continuación se muestra una captura de pantalla del programa siendo utilizado en Windows XP.

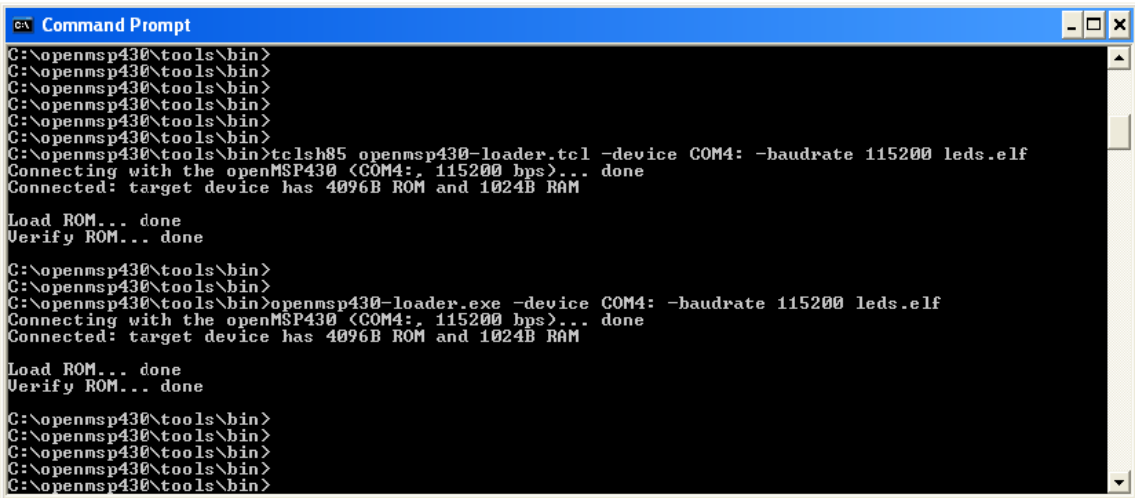


Figura 34: openMSP430-loader en Windows XP

openMSP430-debug

Es un debugger extremadamente simplificado si lo comparamos con otros más conocidos (como Eclipse por ejemplo) pero permite interactuar con el openMSP430 de una forma simple y cómoda.

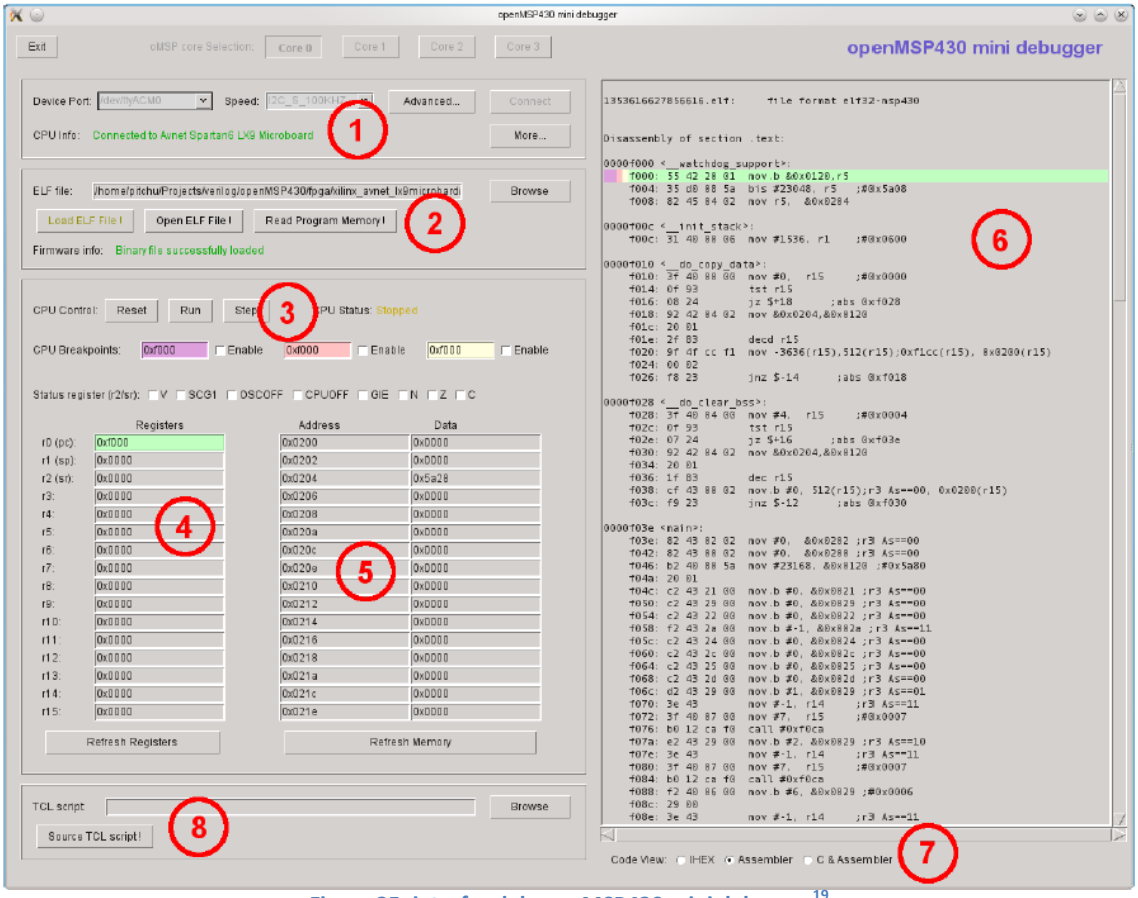


Figura 35: interfaz del openMSP430 mini debugger[19]

[19] Figura extraída de OpenMSP430 Datasheet, Olivier Girard, 2013.

1. Establece conexión con la interfaz serie de depuración del openMSP430.
2. Carga a la memoria de programa el archivo *.elf* o *hex* generado.
3. Control de la CPU: Reset, Stop, Run, Paso a Paso. Permite también la creación de breakpoints en el programa.
4. Acceso de lectura/escritura de los registros de la CPU.
5. Acceso de lectura/escritura de todo el rango de memoria (programa, datos, periféricos).
6. Vista básica del desensamblado cargado en la memoria de programa. La ubicación actual del PC es resaltada en verde, los breakpoints de software en amarillo, rosado y violeta.
7. Elección del tipo de representación del código desensamblado.
8. Fuente de un script Tcl personalizado.

openMSP430-gdbproxy

El propósito de este programa es reemplazar la utilidad "msp430-gdbproxy" proporcionada por el toolchain de mspgcc. Típicamente, un proxy GDB crea un puerto local para que el GDB pueda conectarse, y su vez se ocupa de la comunicación con el hardware deseado. En nuestro caso, se trata básicamente de un puente entre el protocolo de comunicación RSP de GDB y la interfaz de depuración serie del openMSP430.

A continuación se muestra un diagrama del flujo de comunicación:

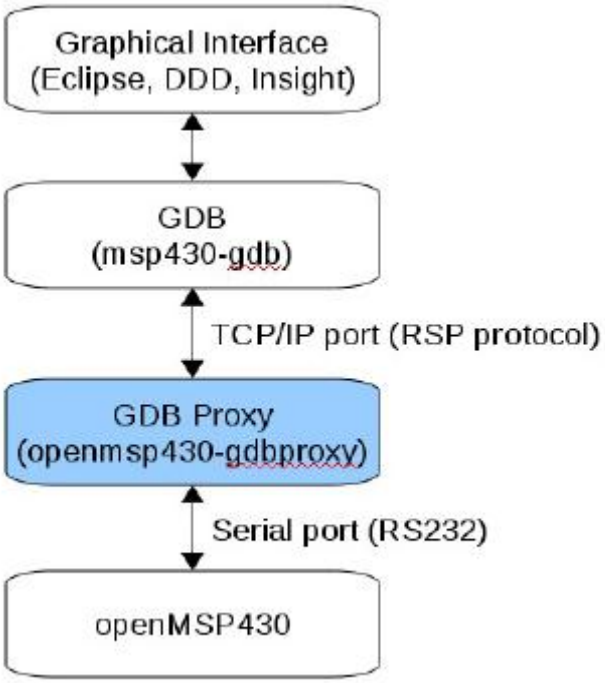


Figura 36: Diagrama del flujo de comunicación entre el openMSP430 y la interfaz gráfica

Al igual que el programa original “msp430-gdbproxy”, el “openmsp430-gdbproxy” se puede controlar desde la línea de comandos. Sin embargo, también proporciona una interfaz gráfica sencilla, que fue la opción empleada por nosotros.

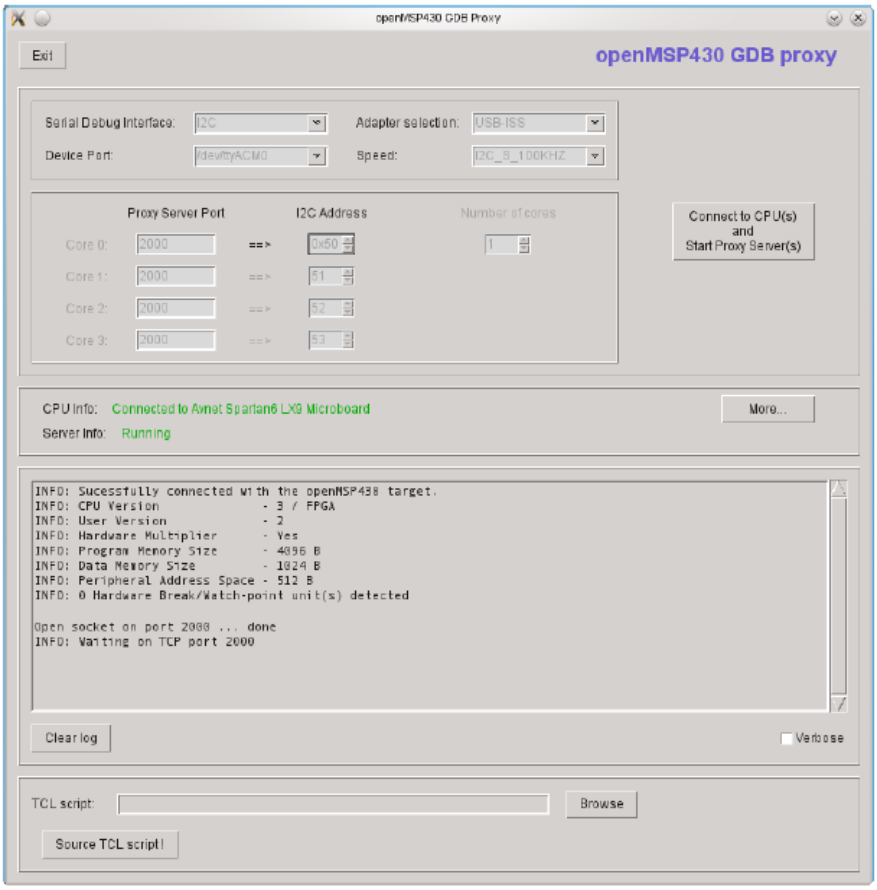


Figura 37: Interfaz gráfica del openmsp430-gdbproxy

Anexo 5: Carga y ejecución de aplicación

A continuación se incluye el código de la versión básica del módulo mcs.c utilizada para ambas campañas de inyección de fallas. Se debe tener en cuenta no se incluye el código agregado para evaluar el resultado de la inyección de fallas.

Como se comentó anteriormente, lo que hace la aplicación es incrementar la cuenta de un contador en 1 cada vez es llamada. También enciende el primer LED de la placa para mostrar que corrió. Al finalizar cada ejecución le solicita al kernel ser despertada al transcurrir los próximos 2 tics (200 ms) del timer A0 para incrementar la cuenta nuevamente (cont = cont + 1). A medida que esto sucede se va guardando el valor de 2^cont en una variable llamada num; es decir que num=2^cuenta. Cuando el contador llega a 3 la aplicación comunica al kernel que no se ejecutará más, llamando a la función stop() y finalizando para siempre.

En nuestro caso la aplicación se incluyó antes del comienzo de la función main() del módulo mcs.c pero puedo haber sido incluida en un módulo independiente. En caso de haber optado por esta otra opción, se deberá incluir también el archivo .h de la aplicación en el módulo “mcs.c”. A continuación se adjunta el código, que seguidamente se comentará.

```
#include "hardware.h"
#include "ukernel_hal.h"
#include "ukernel.h"
#include "common.h"

volatile unsigned int num;
volatile unsigned int cont;

void f (void *p)
{

for(;;){

//prendo led
P3OUT = 0x01;
cont = cont + 1;

      if (num == 0){
            num= 0x0002;
      }else num = num << 1;

      if (cont == 0x03){
      stop();
      } else tsleep(2);

      };
}

int main( void )

{
      stack_t f_stack[0x0100];
      hardware_init();
      system_timer_config ();
      kernel_init ();
      load( f, 0, f_stack, 0x100, WDTIS0 );
      run();
      return 0;
}
```

Se usó la función for(;;) al inicio para que la aplicación se ejecute nuevamente desde la primera línea cuando vuelve a ser llamada. De lo contrario, cada vez que el µkernel ceda el micro a la aplicación se ejecutaría sólo la última línea, en la cual quedó la aplicación al concluir su ejecución anterior.

Luego resta definir un espacio de stack para la aplicación e incluirla en el µkernel. Para esto se define la variable stack_t f_stack[0x0100], la cual reside en el stack de main y es usada para llamar la función load(f, 0, f_stack, 0x100, WDTIS0). En ella se incluyen los parámetros necesarios para agregar la aplicación a la tabla de procesos del ukernel. Obsérvese que el

primer parámetro pasado a la función “load()” es el nombre de la aplicación, en este caso “f”. El parámetro 0x100 es el largo del espacio de stack reservado para la aplicación, el cual debe coincidir con el usado para definir a la variable “f_stack”. Por último, se setea el tiempo con el que desea configurar el watchdog para que resetee al sistema en caso que la aplicación quede congelada. En ese sentido, el valor usado fue de 1 segundo.

Anexo 6: Uso del minidebugger

El “openmsp430-minidebug” es una alternativa a GDB disponible entre las herramientas de debugging incluidas en el proyecto openMSP430. La misma presenta una interfaz gráfica que facilita la tarea del usuario, permitiéndole observar el flujo del programa paso a paso, así como también el contenido de la memoria en tiempo de ejecución.

Para cargar un programa debe previamente generarse el archivo de extensión “.elf”, ejecutando del comando “make” desde consola de comandos. Para que el comando “make” sea reconocido deberá crearse una variable path con la dirección de la carpeta que incluye todas las herramientas de compilación de GCC para openMSP430, donde se encuentra el archivo make.exe, entre otros. Una vez obtenido el archivo de extensión .elf, lo usaremos para abrir el código desde el minidebugger, siguiendo mediante los siguientes pasos:

1- Grabar el openMSP430 en la placa DE0 con el programador lógico de Quartus 13.

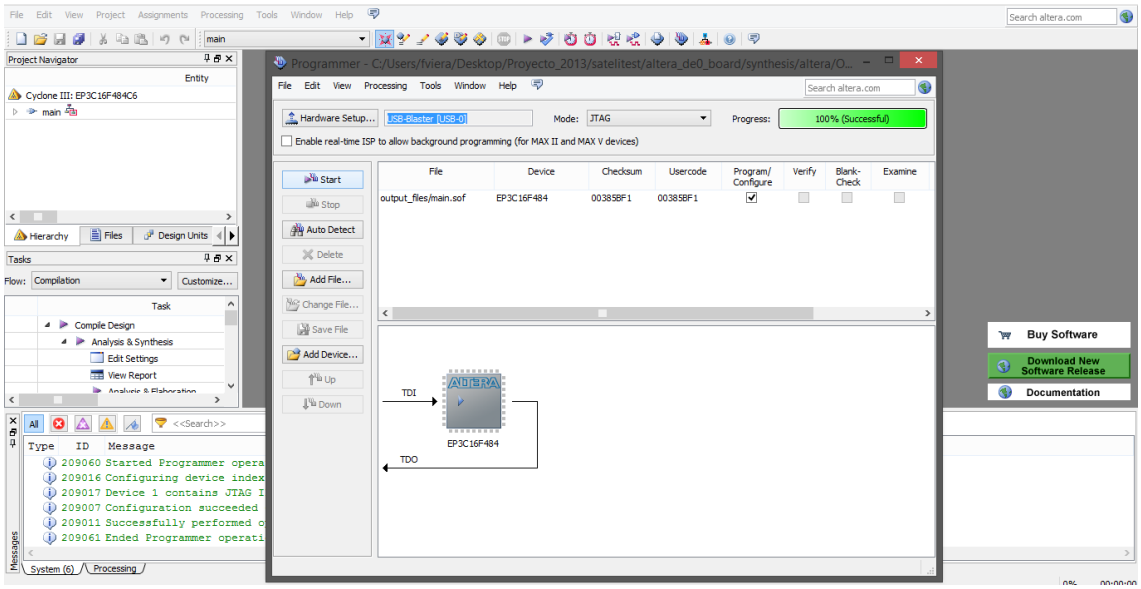


Figura 38: Programación de la placa

2- Abrir el programa “openmsp430-minidebug.tcl” y conectar con la placa. Recordar que se debe elegir el puerto COM adecuado, habiendo previamente instalado los drivers del cable serial de acuerdo al procedimiento detallado en el *Anexo 3: Instalación de driver del cable serial RS232*.

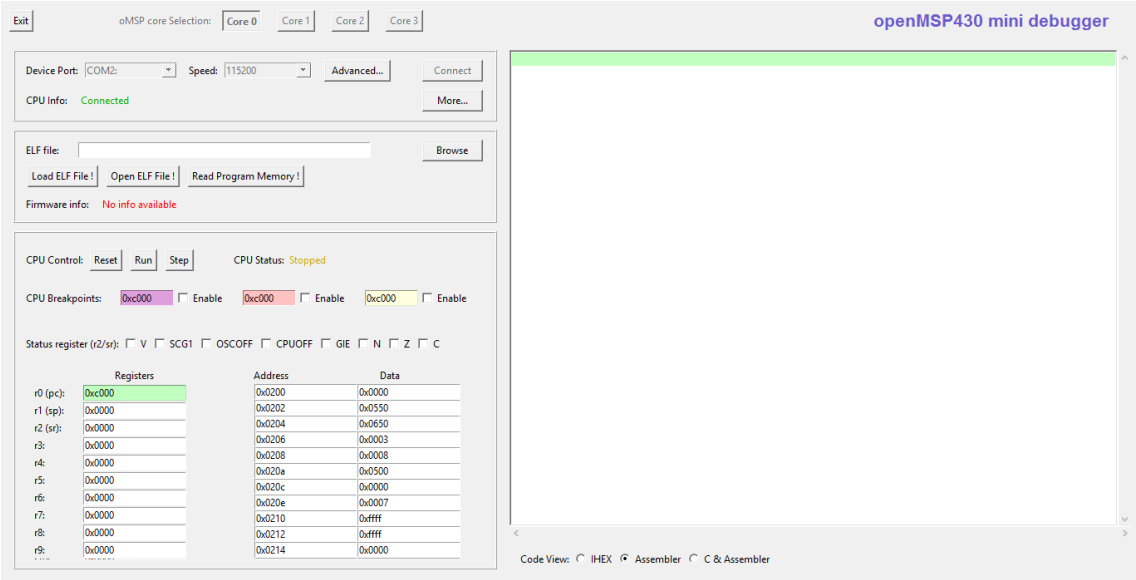


Figura 39: Conexión con la placa desde mini debugger

3- Cargar el archivo de extensión ".elf" generado al compilar nuestro programa. En el caso de ejemplo, el archivo generado es "mcs.elf". Tras un click en el botón "Browse" se busca el archivo correspondiente.

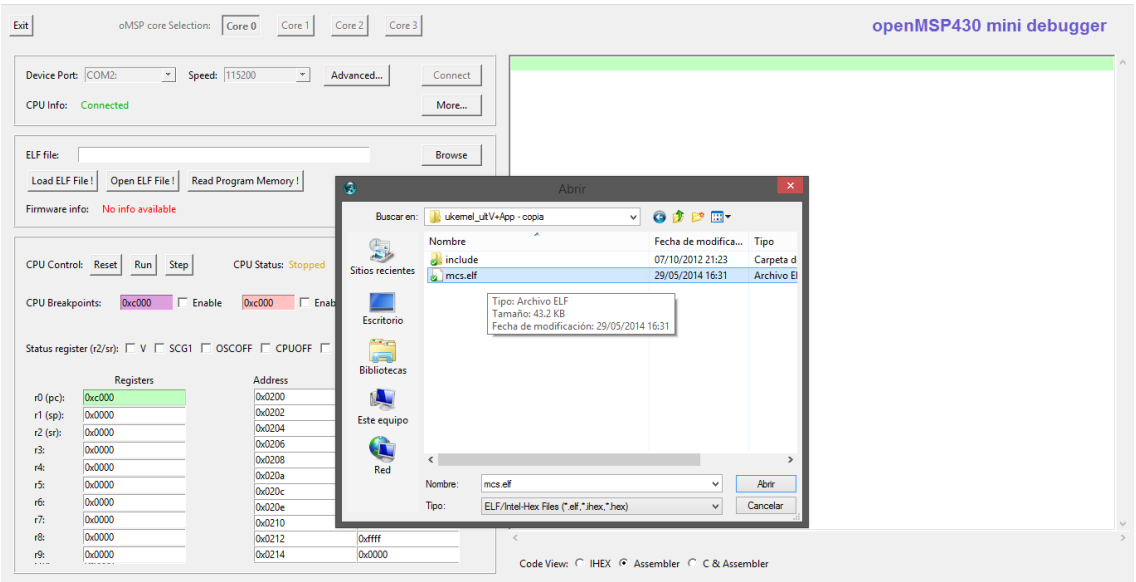


Figura 40: Cargar el archivo de extensión "elf"

4- Una vez abierto el archivo lo cargamos al micro openMSP430 programado en la placa dando un click en el botón "Load". Al cargarse satisfactoriamente el archivo de compilación aparecerá en color verde el mensaje "Binary file successfully loaded" en la sección de "Firmware info". Téngase en cuenta que no siempre aparecerá el mensaje satisfactorio, pudiendo informarse un error al cargar el firmware. De cualquier modo, esto no afecta la sección de debugging, al menos para el alcance de nuestro proyecto.

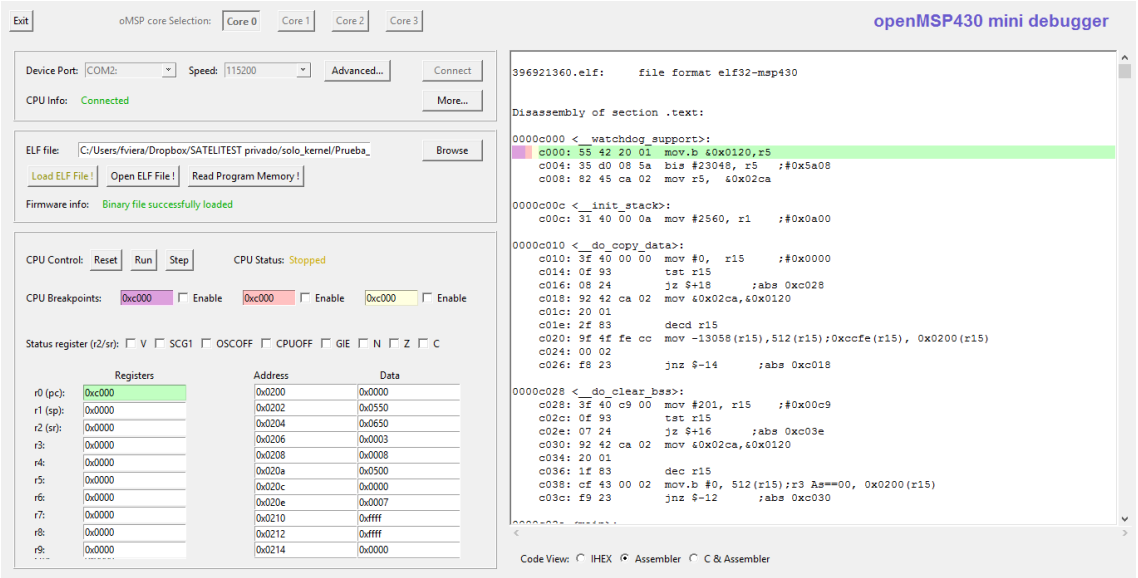


Figura 41: Programa listo para ser corrido paso a paso

5- El programa queda pronto para ser debuggeado “step-by-step” o dando un click en la opción “run”, que provoca la ejecución completa hasta el final.

6- Al finalizar la sesión de debugging, para salir de la herramienta basta con hacer click en el botón de “Exit”.

Anexo 7: GCC inline assembler

Es una herramienta que propone el compilador GCC que permite agregar código de bajo nivel escrito en assembler para ser compilado junto a código de alto nivel.

Sintaxis de GCC inline assembler

1. Nombre de registros: los nombres de los registros son precedidos por el símbolo %, por ejemplo %eax, %ax, etc.
2. Orden de los operando: las variables origen van escritas primero, luego le sigue el destino. Por ejemplo: “ mov %edx, %eax".
3. Tamaño de la operación: los sufijos utilizados para indicar el tamaño en bits de la operación es el siguiente: b para (8-bit) byte, w para (16-bit) word, y l para (32-bit) long. Por ejemplo, la sintaxis correcta para la instrucción de arriba sería "movl %edx, %eax".
4. Operadores inmediatos: los operadores inmediatos como ser números enteros van precedidos del símbolo $ como en "addl $5, %eax", lo que significa que el numero largo 5 se carga en el registro %eax.
5. Memoria de los operando: si se olvida el prefijo $ de una variable ocurre lo siguiente, por ejemplo, si "movl $bar, %ebx" pone la dirección de la variable bar en el registro %ebx, la instrucción "movl bar, %ebx" pondría el contenido de la variable bar en el registro.
6. Indexación: la indexación se realiza mediante la utilización de paréntesis curvos que encierran a una variable, por ejemplo "movl 8(%ebp), %eax" (mueve el contenido con offset 8 de la celda apuntada por %ebp en el registro %eax).

Ejemplo 1

```
asm ("movl %%eax, %0;" : "=r" ( val ));
```

En el ejemplo, la variable “val” se guarda en un registro, el valor del registro eax se copia en el registro, y el valor de “val” se actualiza en la memoria.

Ejemplo 2

```
int no = 100, val ;
asm ("movl %1, %%ebx;"
    "movl %%ebx, %0;"
     : "=r" ( val )       /* salida */
     : "r" ( no )        /* entrada */
     : "%ebx"         /* registro usado */
       );
```

En el ejemplo de arriba, "val" es el operador de salida, referenciado mediante %0 y "no" es el operador de entrada, referenciado por %1. "r" es una restricción en la operación, que le dice al compilador GCC que use alguno de los registros para guardar los operadores.

Los abreviados de los registros son los siguientes:

a %eax, %ax, %al
b %ebx, %bx, %bl
c %ecx, %cx, %cl
d %edx, %dx, %dl
S %esi, %si
D %edi, %di

Anexo 8: Procedimiento para agregar un periférico

Una eventual necesidad que puede tener con frecuencia cualquier usuario del openMSP430 es la de incorporar nuevos periféricos al microcontrolador, por lo cual a continuación se presenta un método simple y conciso para llevar a cabo esta tarea.

Antes que nada se destaca la existencia de módulos predefinidos que ofician como bancos de registros y que al ser instanciados por el módulo principal del proyecto pueden ser accedidos tanto para escritura como para la lectura de forma sencilla por parte del procesador. Los mismos fueron diseñados por el autor del openMSP430 y pueden ser utilizados como base de cualquier módulo que se diseñe externamente. Aparecen en forma de “templates” en el directorio “…satelitest\altera_de0_board\rtl\verilog\openmsp430\periph”.

Estos bancos de registros se presentan en dos variedades, una para periféricos de 8 bits y otra para 16, quedando a gusto del usuario cuál de ellas utilizar. Es altamente recomendable hacer uso de estos templates para evitarse el trabajo de hacer uno mismo la decodificación de las direcciones de memoria para las diferentes entradas de datos.

A continuación se muestra la instanciación de un módulo que funciona como banco de registros, sobre el cual uno podría diseñar cualquier periférico.

```
Banco_registro    banco (
```

```
        .per_dout    ( per_dout_BR ),          // Peripheral data output
        .mclk ( mclk ),                        // Main system clock
        .per_addr    ( per_addr ),             // Peripheral address
        .per_din     ( per_din ),              // Peripheral data input
        .per_en      ( per_en ),         // Peripheral enable (high active)
        .per_we      ( per_we ),  // Peripheral write enable (high active)
        .puc_rst     ( puc_rst )
);
```

Luego de trabajar sobre un banco como los que se tiene del template, se deben realizar las interconexiones correspondientes de los buses de datos de entrada y direcciones de periféricos, señal de habilitación de periférico, reset, clock y habilitación de escritura. La señal de salida se conecta al openMSP430 mediante un OR (|) con las salidas de todos los demás periféricos. Cabe señalar que cuando el periférico esté activo los otros tendrán su salida en cero. El siguiente es un ejemplo de la señal que se conecta al microprocesador:

```
assign per_dout    =    per_dout_dio  |
                        per_dout_tA   |
                        per_dout_tA1  |
                        per_dout_7seg |
                        per_dout_MF   |
                        per_dout_BR   |
                        per_pout_MF   |
                        per_dout_CRC;
```

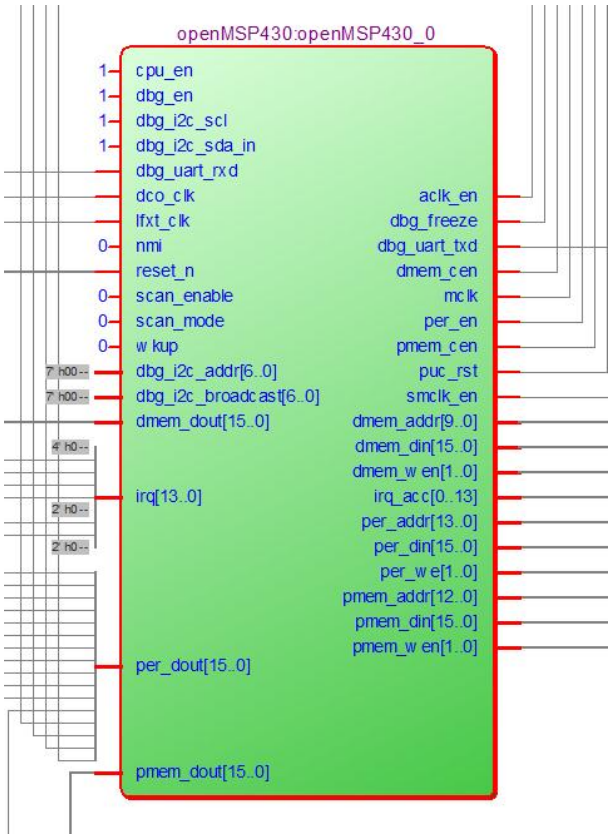


Figura 42: Bloque openMSP430

Al efectuar la interconexión del banco es preciso cerciorarse de que las direcciones establecidas para los registros estén disponibles, para lo cual es recomendable realizar previamente un mapa de las direcciones utilizadas, que facilite encontrar un lugar disponible.

También puede resultar de interés definir un nombre para cada registro, de modo que luego en el programa sea posible referirse a ellos por su nombre y no por su ubicación, facilitando así el posterior depurado del código. Para asociar la dirección del registro con el nombre que se le quiera dar basta con agregar en el archivo omsp_system.h una línea como la que se muestran a continuación para cada registro.

```
#define  CRCINIRES       (*(volatile unsigned char  *) 0x0142)
```

Anexo 9: Mecanismo para la ampliación de memoria

El mapeo de memoria en el openMSP430 es ampliamente configurable, soportando dos métodos que mantienen compatibilidad del 100% con el linker provisto por MSPGCC. A continuación se describirá el procedimiento a seguir para cada uno de ellos.

Método de configuración básica

El primer método consiste en modificar la sección “basic system configuration” del archivo “openMSP430_undefines.v” ubicado en “…\altera_de0_board\rtl\verilog\openmsp430.” Lo que se debe hacer es cambiar el tamaño de las memorias de programa y datos para luego recompilar el proyecto en Quartus. Para modificar el tamaño de las memorias basta con dejar habilitadas las definiciones de constantes que hacen referencia al tamaño deseado, teniendo especial cuidado de verificar que la suma no supere los 64 kB que el openMSP430 admite como máximo. A continuación se muestra la configuración utilizada en nuestro caso así como también un esquema del mapeo de memoria que ilustra las posibilidades de configuración.[20]

```
`define PMEM_SIZE_16_KB
`define DMEM_SIZE_2_KB
```

[20] Como se ve en la imagen, el cambio introducido en tamaño de la memoria de programa modificará su comienzo por lo cual se debe editar el link.ld.

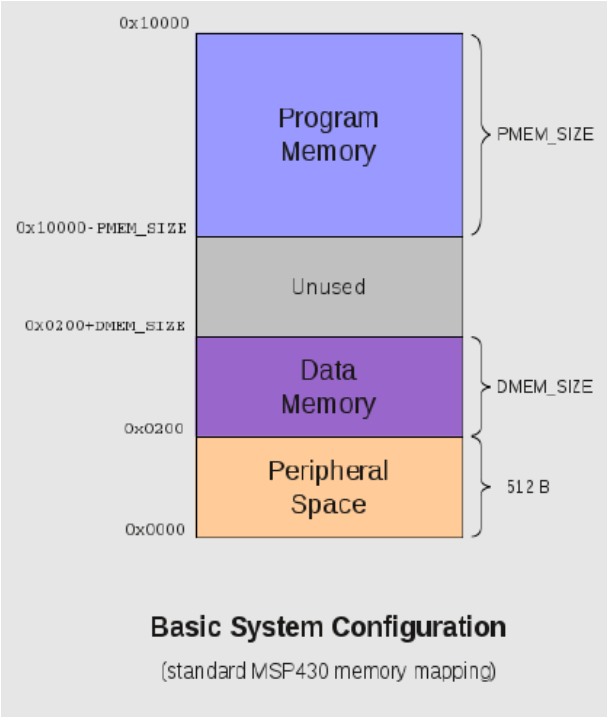


Figura 43: Mapa de memoria según configuración básica

Para una memoria de programa de 16 kB se deben realizar las siguientes modificaciones en el archivo link.hd:

```
MEMORY
{
  data     (rwx)    : ORIGIN = 0x0200,       LENGTH = 0x400
  text     (rx)     : ORIGIN = 0xf000,       LENGTH = 0x1000-0x20
  vectors  (rw)     : ORIGIN = 0xffe0,       LENGTH = 0x20
}
```

Método de configuración avanzada

El segundo método implica también la edición del archivo “openMSP430_undefines.v” pero en la sección de “advanced system configuration”. Esta opción es útil para aquellos que vean el espacio de memoria de periféricos disponible (512 bytes) como una limitante para sus necesidades, dado que permite ampliar este rango introduciendo un “offset” para el resto de las direcciones de memoria de datos y programa. Esto implicará una ligera modificación del linker, donde se deberán detallar los nuevos comienzos de las memorias. Cabe señalar que el espacio de direccionamiento de periféricos puede crecer hasta un máximo de 32 kB, a dividirse entre módulos de 8 y de 16 bits. Se despliega ahora un esquema del mapeo de direcciones que detalla las posibilidades de configuración avanzada.

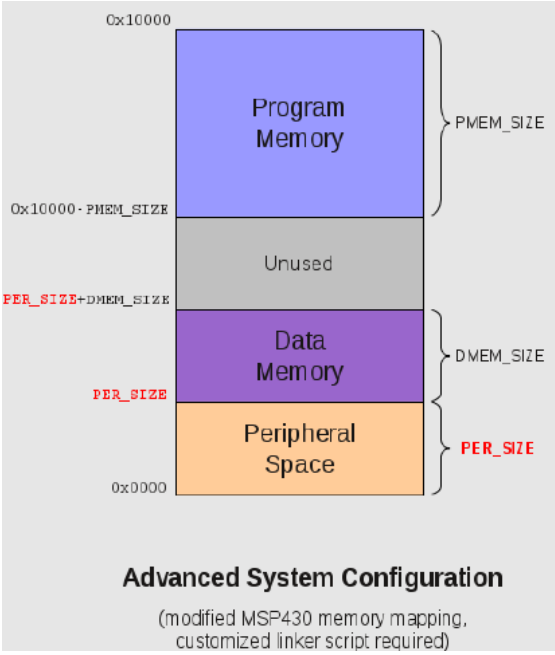


Figura 44: Mapa de memoria según configuración avanzada

Instanciación de memorias

Luego de la elección del método para la ampliación de memoria y compilación, veremos que el ancho del bus saliente del procesador del openMSP430 dedicado al direccionamiento va a crecer en el caso de que la memoria haya aumentado o de lo contrario disminuir si se redujo. Por lo tanto, se deberá corregir el direccionamiento dentro de la instanciación de las memorias así como crear nuevas memorias con el Megawizard de Quartus que se correspondan con el tamaño definido en el archivo anterior. A continuación se detalla el método para crear memorias con Quartus que se utilizarán para emular las memorias de datos y programa.

En primer lugar, nos dirigimos a la opción “MegaWizard Plug-In Manager” dentro de la pestaña “Tools” del Quartus.

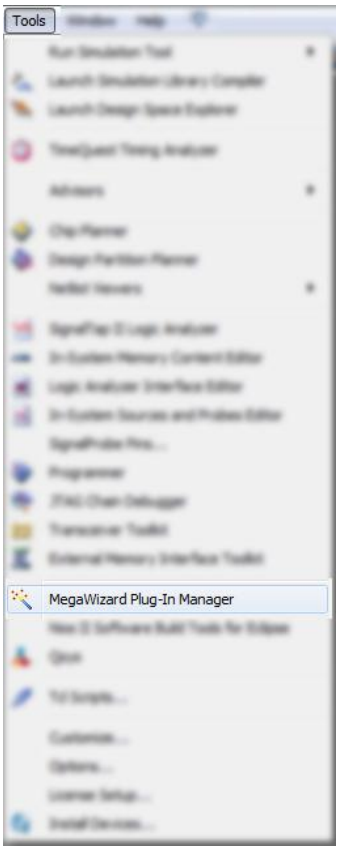


Figura 45: Menú de la pestaña "Tools"

A continuación elegimos la opción “Create a new custom megafunction variation” y se da click en “Next”, apareciendo la siguiente ventana:

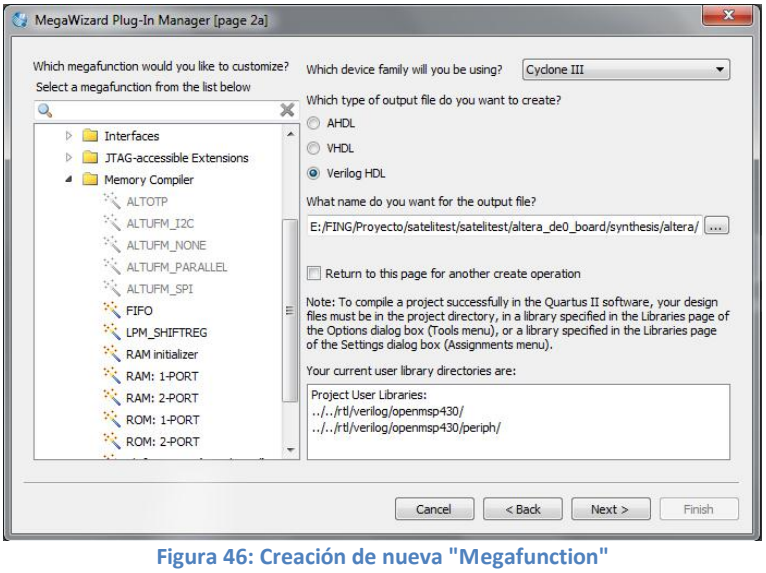


Figura 46: Creación de nueva "Megafunction"

Ahora se deberá elegir dentro de la sección “Memory Compiler” la opción RAM: 1-PORT, otorgar un nombre a la instancia de la memoria a crear y escoger el lenguaje en el cual pretendemos la salida. En nuestro caso se optará por la opción “Verilog”, al ser el que se empleó durante todo el proyecto.

A continuación emergerá una nueva ventana, en la que se configuran los parámetros que caracterizan la memoria a crear. En este ejemplo se definirá una memoria de 512 lugares con palabras de ancho 16 bits.

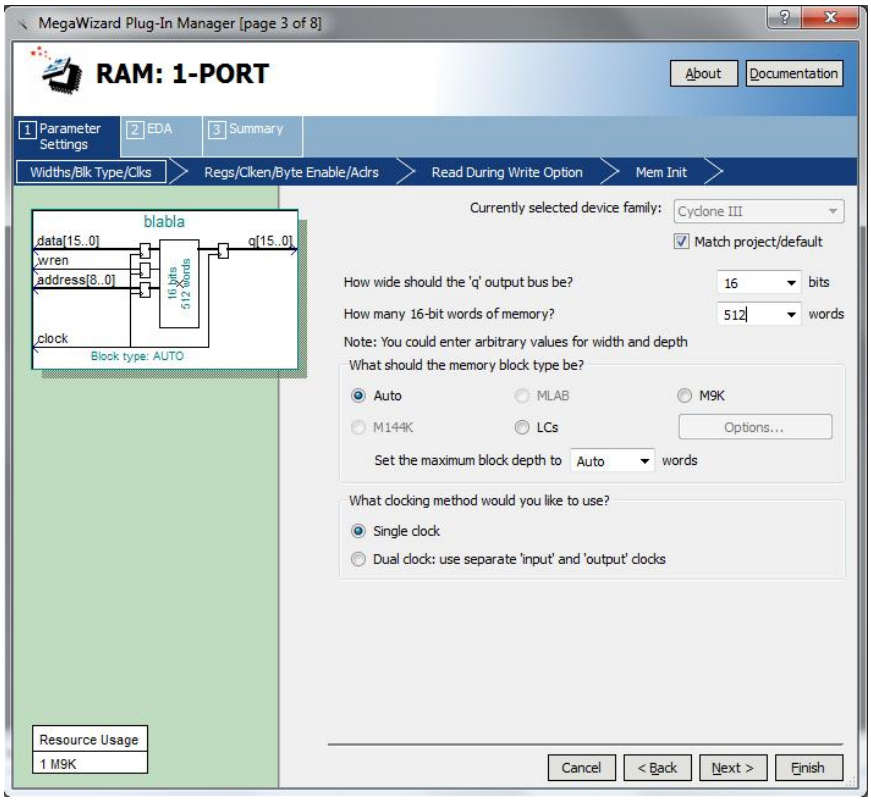


Figura 47: Dimensionado de la memoria

En la próxima ventana se elegirán los puertos a ser registrados y se presentan algunas opciones de señales adicionales. En nuestro caso seleccionamos la opción de “Create byte enable for port A”, dado que el openMSP430 tiene prevista la opción de escribir o leer en 8 bits para ciertos casos especiales.[21]

[21] Si bien en nuestro proyecto las memorias cuentan con esta opción, no se hace uso de dicha posibilidad.

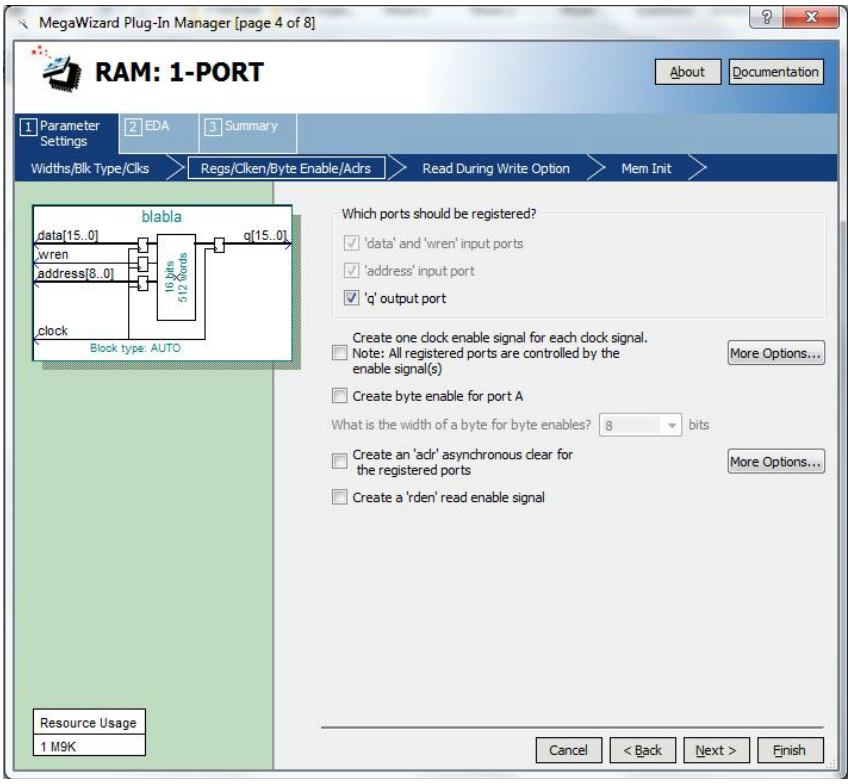


Figura 48: Elección de entradas y salidas

Luego en la sección “Mem Init” tendremos la posibilidad de cargarle a la memoria un contenido de inicialización que se grabará al transferir el proyecto de Quartus a la FPGA con la que se esté trabajando.

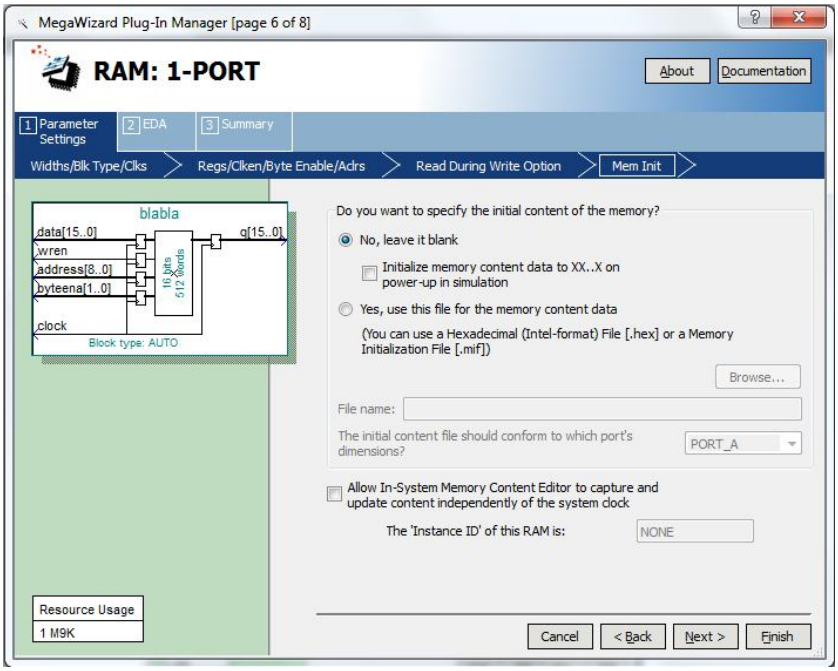


Figura 49: Contenido de inicialización de la memoria

Se optará por dejar la memoria en blanco, ya que luego se la inicializará y borrará reiteradamente durante las campañas de inyección de fallas. Ahora basta con dar “Next” hasta llegar al final del asistente.

Una vez concluida esta etapa resta únicamente instanciar el módulo en el main del proyecto. Dado que ya existen memorias en el mismo, la tarea consistirá en cambiar la instancia actual por la correspondiente a la nueva memoria, para luego finalmente corregir el ancho de los buses de direccionamiento en caso de ser necesario. A continuación se muestra una instanciación a modo de ejemplo:

```
ram16x1024 ram (   //1024
        .address (dmem_addr_wire[9:0]), // ANTES ERA DE 8 -> 0
        .clken   (~dmem_cen),
        .clock   (clk_sys),
        .data    (dmem_din[15:0]),
        .q       (dmem_dout_wire[15:0]),
        .wren    ( ~(&dmem_wen[1:0]) ),
        .byteena ( ~dmem_wen[1:0] )
);
```

Anexo 10: Manual de uso de GDB

¿Por qué GDB?

La anterior es una interrogante válida que le puede surgir al lector dado que ya se ha presentado al minidebugger como una herramienta sencilla que permite al usuario debuggear con una facilidad que aparente GDB nunca podrá igualar. El hecho es que si bien el minidebugger permite realizar algunas tareas de manera muy simple, sólo lo logra a cambio de una restricción enorme con respecto a las capacidades de GDB. De hecho, estrictamente el minidebugger no es más que una pequeña implementación de un entorno gráfico para un número muy limitado de comandos de GDB. Por lo tanto, en ocasiones optaremos por este último en virtud de su potencia, pese a ser menos amigable.

Este manual detalla las funciones más utilizadas a la hora de debuggear y servirá como referencia para aquellos que deseen detenerse en algunos de los scripts utilizados en el proyecto. Cabe aclarar que se hablará indistintamente de GDB o MPS430-GDB, siendo esta última la herramienta que efectivamente se empleó. Si bien no son la misma cosa, la segunda basa su funcionamiento en la primera, y no se presentará en este manual la necesidad de distinguir entre comandos de una y otra.

Introducción

GDB o GNU Debugger es el depurador estándar para el compilador GNU. [19] Se trata de un software libre y portable, que funciona en varias plataformas “Unix” para diferentes lenguajes de programación, tales como C, C++ y Fortran. Ofrece la posibilidad de trazar y modificar la ejecución de un programa, permitiendo al usuario controlar y alterar los valores de las variables internas de un programa. GDB no posee interfaz gráfica propia y por defecto se usa mediante línea de comandos.

Invocación del Debugger

El debugger se puede ejecutar de alguna de las siguientes formas: [20]

```
$ gdb
```

(o bien como en nuestro caso

```
$ msp430-gdb)
```

para entrar al modo interactivo

```
$ gdb programa
```

para cargar el programa y entrar en el modo interactivo. En este caso el programa no comienza hasta que sea indicado con un comando.

Comandos más frecuentes

Los comandos más frecuentemente usados son:

```
list [archivo:]funcion
list [archivo:]linea[,linea]
list
list–
```

Para listar el fuente a partir de una función o una línea. Si se escribe únicamente “list”, se continúa el listado previo (page down). Mientras tanto, “list –“ trae las líneas anteriores (page up).

```
break [archivo:]función
break [archivo:]línea
```

Para colocar un breakpoint al comienzo de la función o al comienzo de la línea indicada.

```
run [argumentos]
```

Para comenzar la ejecución del programa desde el principio. Los argumentos son los pasados al ejecutable.

```
backtrace (backtrace)
bt (backtrace)
```

Muestra el stack del programa, indicando las funciones invocadas y en qué lugares fueron llamadas.

```
printexpr
p/xexpr
```

Muestra el valor de una expresión.

```
c
```

Para continuar la ejecución del programa después de que ha sido detenido con un “signal” o un “breakpoint”.

```
next
```

Ejecuta la próxima línea del programa sin ingresar dentro de las funciones. Se puede aprovechar el hecho que GDB repite el último comando con ENTER para ejecutar varias líneas seguidas.

```
step
```
Ejecuta la próxima línea del programa entrando en funciones. Se puede aprovechar el hecho que GDB repite el último comando con ENTER para ejecutar varias líneas seguidas.

```
jump línea
```
Salta los comandos siguientes y comienza la ejecución a partir de "línea". Útil para continuar un programa que ha terminado con "core", arreglando la situación y saltando las líneas defectuosas.

```
help [item]
```
Ayuda en línea.

```
Quit
```
Sale de GDB.

Los comandos desplegados hasta ahora posibilitan básicamente la ejecución de un programa para su depurado. De hecho, permiten realizar más operaciones que las que ofrece el minidebugger. Cabe destacar que para poder hacer uso de GDB es necesario compilar previamente con alguna herramienta compatible, que a su vez genere información de debugging, algo que IAR se comprobó que no hace.

Uso de Breakpoints

Un breakpoint hace que un programa se frene al pasar por cierto punto del mismo, pudiendo fijar además condiciones de detención para ser tan meticuloso como se quiera. Para setear un breakpoint existen diferentes maneras de configuración, algunas de las cuales se describen a continuación.

Los breakpoints se pueden setear con el comando "break" o también con su abreviatura "b".

```
break [archivo:]funcion
break [archivo:]linea
break ubicacion
```
Establece un breakpoint en una ubicación determinada. Se debe especificar un nombre de función, un número de línea o una dirección de una instrucción. El programa se detendrá en el breakpoint justo antes de que se ejecute el código en la ubicación especificada. También es posible introducir breakpoints que produzcan la detención del programa cuando es una cierta tarea específica la que paso por el punto de interrupción establecido.

```
break
```
Cuando se invoca sin argumentos, el breakpoint se establece en la siguiente instrucción a ejecutar en el stackframe seleccionado. Esto produce que el programa se detenga tan pronto como se devuelve el control del programa al frame seleccionado. GDB normalmente ignora los breakpoints cuando reanuda la ejecución del programa, hasta que al menos se haya ejecutado una instrucción. De no ser así nunca se podría continuar más allá de un punto de interrupción sin antes de deshabilitar dicho breakpoint. Esta regla se aplica independientemente de si el breakpoint ya existía cuando el programa se detuvo.

```
break … if cond
```
Establece un punto de interrupción con la condición "cond", la cual se evalúa cada vez que se alcanza el breakpoint y el programa sólo se detendrá en caso de cumplirse.

```
clearlinea
clearfuncion
```
Para eliminar un breakpoint de la línea indicada.

```
delete numero
```
Para eliminar un breakpoint por número.

```
disablebreakpoint
enable breakpoint
```
Para habilitar o deshabilitar temporalmente un breakpoint. A diferencia de “delete”, no se pierde la referencia de la línea donde se encuentra, sino que simplemente es ignorado.

```
ignore breakpoint [count]
```
Ignora <count> pasadas sobre el breakpoint <breakpoint>.

Examinar memoria y modificar datos

Se puede utilizar el comando “x” para examinar el contenido de la memoria en múltiples formatos independientemente del tipo de datos del programa.

```
x [/fmt]addr
xaddr
x           Use the x command to examine memory.

      o     octal             f     float
      x     hexadecimal       a     address
      d     decimal                 u     unsigned decimal
      t     binary            s     string
      c     char
```

Las letras de tamaño son:

```
            b     byte              h     halfword
            w     word              g     giant (8 bytes)
```

El contador indica cuantos elementos imprimir, de modo que:

```
            x /10xb addr
```
Imprime los 10 siguientes bytes del arreglo en hexadecimal.

```
      set variable=expresion
```
Cambia el valor de una variable al resultado de la expresión.

```
      set*((int)[addr])= expression
```
Cambia el valor contenido en la dirección pasada como parámetro.

Comandos para impresión de información de estado

Los siguientes comandos imprimen información variada de estado del debugger y del programa depurado:

```
info files
```

Muestra los archivos y procesos que se están depurando.

```
info program
```

Estado del programa al momento.

```
info sources
```

Muestra los archivos fuentes en debugging.

```
info breakpoints
```

Muestra todos los breakpoints en efecto.

Otros comandos útiles

```
file
```

Para cargar un nuevo ejecutable y tabla de símbolos dentro del debugger.

```
cd
```

Para cambiar de directorio.

```
pwd
```

Para ver el directorio actual

```
make
```

Para correr el programa make.

```
target extended-remote localhost:port
```

Establece conexión con el GDB-proxy por el puerto “port”

Anexo 11: Pasos para realizar una campaña de inyección de fallas

1. Grabar el openMSP430 en la placa DE0 mediante el Quartus II 13.0.0 o superior.

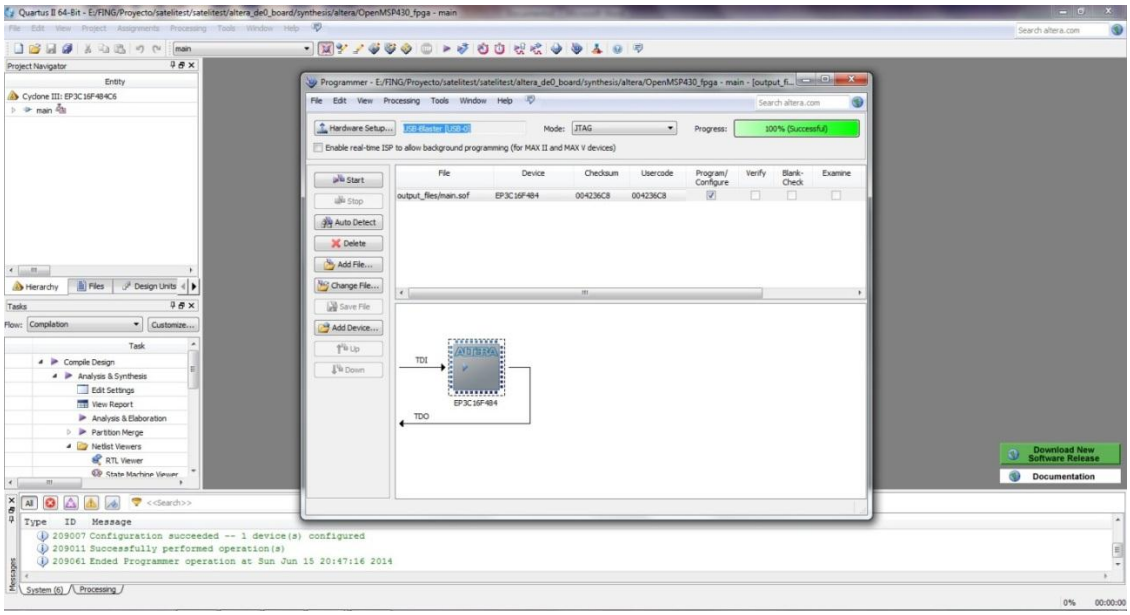


Figura 50: Programación de la placa

2. Iniciar el openmsp430-gdbproxy mediante línea de comandos y establecer la conexión entre la CPU y el GDB-proxy.

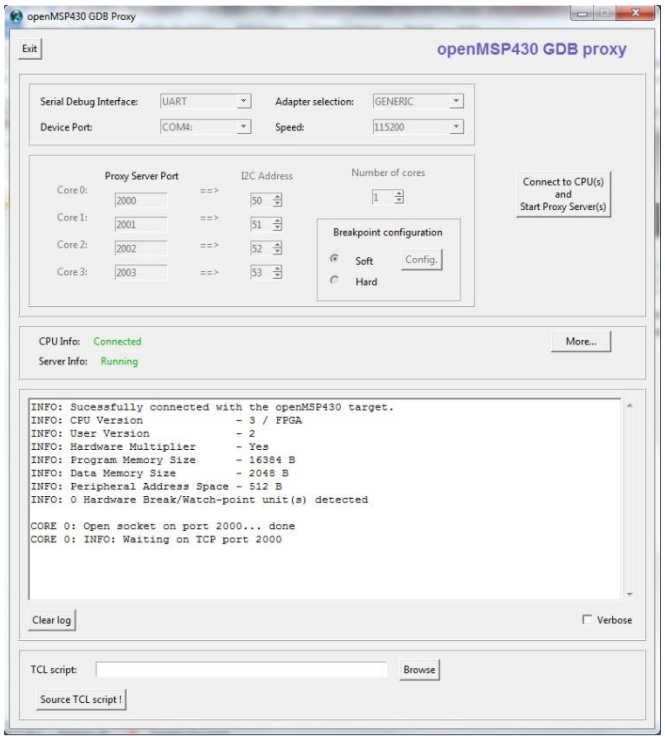


Figura 51: Conexión con GDB-proxy

3. Dirigirnos mediante línea de comandos al directorio donde se encuentra los archivos del kernel y los necesarios para realizar las campañas. Luego se procede a iniciar el MSP430-gdb.exe.

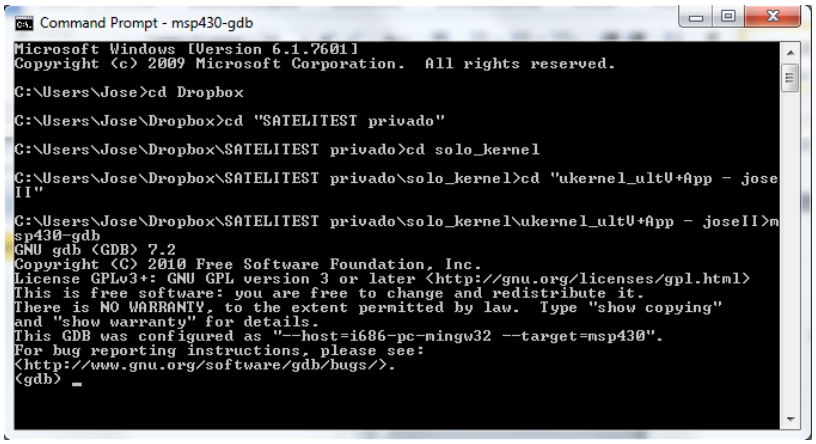


Figura 52: Elección del directorio

Verificamos que en el directorio se encuentren todos los archivos necesarios.

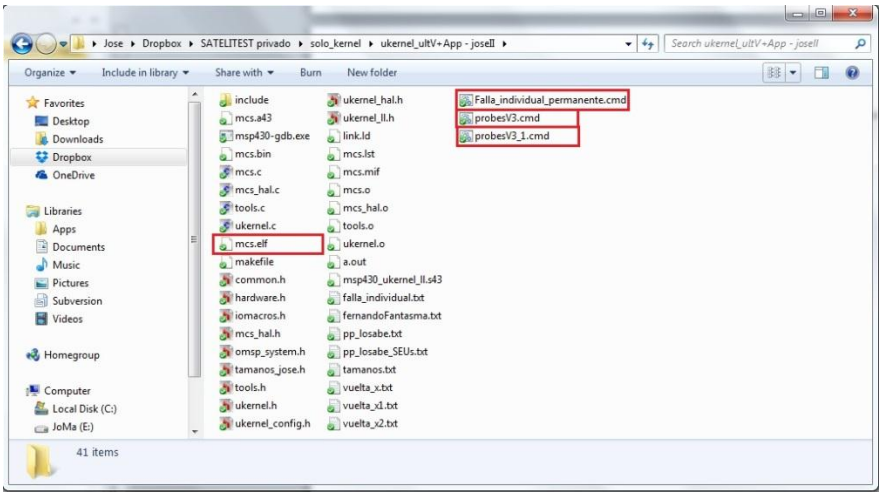


Figura 53: Verificación de la presencia de archivos necesarios

El archivo de extensión “elf” es el que se procederá a cargar en la placa, mientras que lo de tipo “cmd” son los que contienen los scripts que implementan las diferentes campañas. El resto de los archivos, si bien no se utilizan para las campañas, son los responsables de generar el “elf” en el proceso de compilación. Los “txt” son las salidas de los scripts.

Fallas permanentes

Para el comienzo de las campañas de fallas permanentes, una vez parados en el directorio indicado anteriormente, se introduce el siguiente comando:

```
source –s probeV3.cmd
```

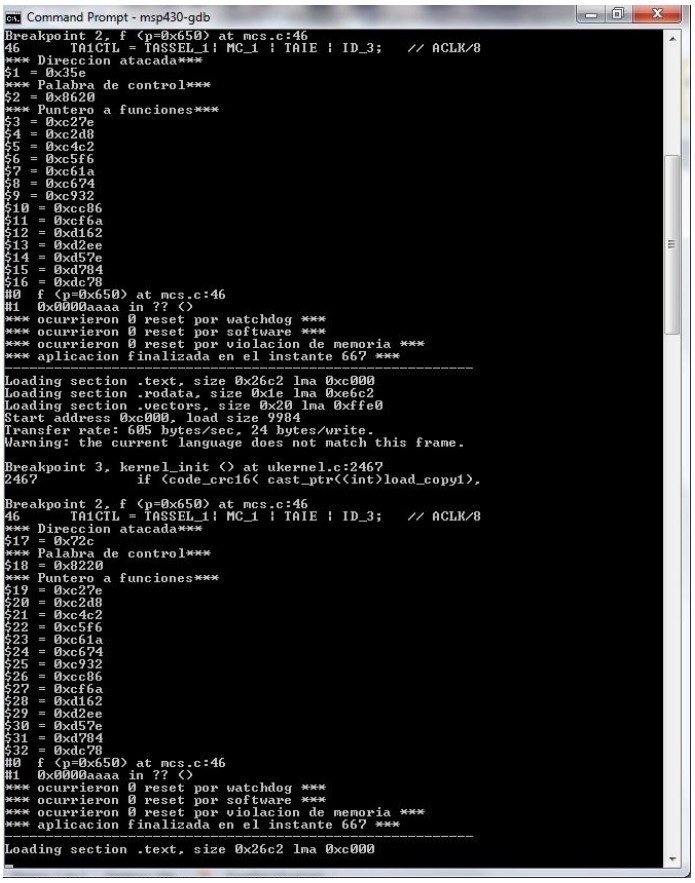


Figura 54: Procesos en ejecución y salidas

Se obtiene en modo “verbose” los procesos que está ejecutando la PC. Sin embargo, en el script se selecciona de forma más específica lo que interesa almacenar en cada corrida. La salida se guarda en un “txt” en el directorio de trabajo.

SEUs

En este caso se deben tratar por un lado las fallas que afectarán la memoria de datos y por otro las que atacan la memoria de programa. Se explicará un único procedimiento ya que el mismo es análogo para ambos casos.

Dado que el instante en el cual se inyecta la falla pasa ahora a ser una variable más, las fallas deben activarse luego del proceso de cálculo de CRC de las funciones que se encuentran triplicadas, para no afectar dicho proceso. Se optó entonces por realizar la inicialización del módulo de fallas desde el kernel, tal como se muestra a continuación:

Figura 55: Inicialización del módulo de fallas desde programa

Luego de esta modificación se debe compilar y por último a iniciar la campaña de fallas mediante el siguiente comando:

```
Source –s probeV3_1.cmd
```

Al igual que en el caso de fallas permanentes, la consola de comandos mostrará en modo “verbose” lo que está realizando, al tiempo que los resultados de interés son almacenados en un archivo de texto.

Anexo 12: Etapas del trabajo realizado y evaluación de la gestión

Al definir el alcance del proyecto se fijaron una serie de tareas hacia las cuales dirigir nuestros esfuerzos. Las mismas se fueron modificando según se avanzaba en el trabajo. A continuación se comentan las diferentes etapas que se llevaron a cabo, las dificultades enfrentadas y los cambios en la planificación.

Estudio de conceptos de lenguaje C y verilog

Como tarea inicial y teniendo en cuenta la escasa experiencia que el grupo tenía en cuanto a lenguaje C y Verilog se dedicó un tiempo a familiarizarnos con ellos, tanto de manera autodidacta como curricular, a través de cursos de grado de la carrera: Sistemas Embebidos y Diseño Lógico 2.

Familiarización con el openMSP430

Esta etapa incluyó varias tareas, tales como estudio del core openMSP430, pruebas en FPGAs (DE1 y posteriormente DE0) con aplicaciones provistas por el fabricante y familiarización con las herramientas básicas de debugger (Minidebugger y GDB). En esta etapa no se generaron compilaciones, dado que los archivos ".elf" utilizados fueron los incluidos en las aplicaciones originales. Hasta aquí, más allá de algunos inconvenientes para migrar el core a la placa DE0, no se enfrentaron mayores dificultades.

Comprensión de la arquitectura y funcionamiento del satélite

Se realizó un repaso general de las características principales del satélite. Se investigó acerca la función que cumple cada módulo del mismo, periféricos y protocolos de comunicación utilizados. No se indagó mucho más que esto, dejando de lado todo lo que implica el código del módulo kernel, al cual dedicamos un exhaustivo estudio en meses posteriores.

Definición de subsistemas a evaluar

Para encarar esta etapa fue necesario coordinar una reunión con parte del equipo del satélite con el fin de conocer en detalle los modelos de chip MSP430 que son usados, periféricos utilizados y en qué módulos son empleados, propósito e importancia relativa. La información recabada sirvió para acotar el espacio de estudio y tener una idea de qué tan lejos se estaba de aproximar el comportamiento del satélite a partir del openMSP430.

Definición de periféricos a agregar

Se decidió en un principio implementar los periféricos SPI e I2C en lenguaje VHDL, correspondientes a las familias 5 y 6 del chip MSP430 para complementar el openMSP430. Ambos proyectos fueron encomendados a grupos de proyecto del curso de Diseño Lógico 2. Además se decidió integrar la versión VHDL del periférico Watchdog presente en las familias 5 y 6 del MSP430. En la instancia del primer hito ya teníamos cierto atrasado con respecto a la planificación, pero se consideró que para recuperar el tiempo perdido sería suficiente usar los buffers previstos y ser menos exigentes en la primera campaña de inyección en cuanto a volumen de fallas y módulos testeados.

Diseño del mecanismo de inyección de fallas

Se comenzó por diseñar un periférico a incluir en el openMSP430 que permitiera la inyección de fallas. En una primera etapa se trabajó sólo con fallas de tipo permanente, realizando

pruebas individuales mediante escrituras y lecturas en memoria. El agregado de SEUs conllevó mayor dificultad dado que implicó el manejo de máquinas de estados. Paralelamente se realizó la adaptación del módulo kernel y diseño de una aplicación de prueba para la primera campaña, lo cual se comentará a continuación.

Adaptación de módulo kernel

Este proceso implica la migración del módulo kernel original y se dividió en cuatro etapas:

1- Modelo en IAR del openMSP430 y migración a plataforma de 16 bits
2- Adaptación de los módulos principales para compilar con GCC y ensamblador “as”
3- Compilación y generación de archivo de extensión “elf”
4- Diseño de aplicaciones de prueba

Replanificación

Al llegar al segundo hito fue necesario modificar el cronograma previsto para las etapas restantes. Se descartó la inclusión de los periféricos SPI e I2C y el objeto de estudio pasó a ser entonces una aplicación similar al módulo de control principal del satélite corriendo sobre el kernel. Se elaboró un nuevo diagrama de Gantt para las tareas a realizar en los 90 días restantes, previendo la posibilidad de solicitar una prórroga de un mes.

Primera campaña de inyección de fallas

Finalmente se optó por usar como activación de las fallas únicamente la *Aplicación sencilla* corriendo sobre el kernel. Para posibilitar el mecanismo de registro de resultados se incorporó el timer A1 y un banco de registros. Se redactó un script GDB para automatizar la inyección de fallas, realizando luego la primera campaña de inyección, de momento sólo con fallas de tipo permanente. Se iteró varias veces dentro de esta etapa corrigiendo errores, enfrentando la dificultad de que cada campaña toma alrededor de 40 minutos. Se analizaron resultados en forma global e individualmente para ciertos casos de interés. Dado el trabajo que quedaba por delante, algunos días antes de concluir esta etapa se optó por solicitar una prórroga de 30 días.

Segunda campaña de inyección de fallas

En esta etapa se incluyó la posibilidad de inyectar fallas de tipo SEU. Asimismo, se incorporó el módulo CRC, que permite el chequeo de consistencia de código y del área de stack. Además se depuraron el script y el método de registro de resultados. Se realizó la segunda campaña de inyección de fallas, analizando resultados en forma general y particular.

Documentación

Si bien en forma paralela al desarrollo del proyecto se fue generando documentación, más de la mitad de la misma se redactó en las últimas semanas. Este proceso tomó muchas horas de trabajo individual y grupal, concluyendo que hubiera sido beneficioso no llevarlo a cabo tan cerca del final.

9. REFERENCIAS

[1] P. universitarios, «El primer satélite uruguayo,» 2013. [En línea]. Available: http://pro-universitarios.com/investigacion-2/el-primer-satelite-uruguayo/.

[2] ANTEL, «Últimas etapas de prueba del satélite Antel Sat,» 2013. [En línea]. Available: http://www.antel.com.uy/antel/institucional/sala-de-prensa/eventos/2013/ultimas-etapasu-de-prueba-del-satelite-antel-sat.

[3] J. K. Bekkeng, «Radiation effects on space electronics,» Universidad de Oslo, [En línea]. Available: http://www.uio.no/studier/emner/matnat/fys/FYS4220/h11/undervisningsmateriale/foresinger-vhdl/Radiation%20effects%20on%20space%20electronics.pdf.

[4] A. Pérez Rodríguez, «Estudio y ensayo de un circuito digital para una prueba de radiación ionizante,» Escuela Técnica Superior de Ingeniería - Universidad de Sevilla, [En línea]. Available: http://bibing.us.es/proyectos/abreproy/11495/direccion/PFC_ARP%252F.

[5] Wikipedia, «Cubesat,» [En línea]. Available: http://es.wikipedia.org/wiki/CubeSat.

[6] Texas Instruments, «MSP430x5xx / MSP430x6xx Family User's Guide,» 2008. [En línea]. Available: http://staging.lierda.com/upload/ftp/201102151342.pdf.

[7] O. Girard, «openMSP430,» 2013. [En línea]. Available: http://opencores.org/project,openmsp430,core.

[8] Texas Instruments , «Msp430x1xx Family User's Guide,» 2006. [En línea]. Available: http://www.ti.com/lit/ug/slau049f/slau049f.pdf.

[9] O. Girard, «OpenMSP430 Datasheet,» 2013. [En línea]. Available: http://opencores.org/websvn,filedetails?repname=openmsp430&path=%2Fopenmsp430%2Ftrunk%2Fdoc%2FopenMSP430.pdf.

[10] G. de Martino, "Manual de usuario µKernel Versión 6.14", INCO - IIE, FING - UdelaR, 2013.

[11] O. Miranda Gómez y P. Mejía Álvarez, «Kernel de Tiempo Real para Control de Procesos,» [En línea]. Available: http://delta.cs.cinvestav.mx/~pmejia/miranda-mejia.pdf.

[12] «The GNU Assembler,» [En línea]. Available: https://web.eecs.umich.edu/~prabal/teaching/eecs373-f11/readings/Assembler.pdf.

[13] L. Entrena, *Fast Fault Injection Techniques using FPGAs,* Universidad Carlos III de Madrid, 2013.

[14] J. Pérez Acle, «"Prototipado en FPGAs para inyección de fallas. Aplicación a sistemas distribuidos sobre bus CAN.",» 2005.

[15] O. Girard, «16 bit peripheral template,» 2009. [En línea]. Available: ...openmsp430\trunk\fpga\altera_de1_board\rtl\verilog\openmsp430\periph\template_periph_16b.v.

[16] Wikipedia, «"CRC",» [En línea]. Available: http://es.wikipedia.org/wiki/Comprobaci%C3%B3n_de_redundancia_c%C3%ADclica.

[17] E. Stavinov, «CRC Generator for Verilog or VHDL,» [En línea]. Available: http://outputlogic.com/?page_id=321.

[18] Universidad de Umeå, Suecia, «Calculated 16-bit CRC,» [En línea]. Available: www8.cs.umu.c/~isak/snippets/.

[19] Wikipedia, «GNU Debugger,» [En línea]. Available: http://es.wikipedia.org/wiki/GNU_Debugger.

[20] N. Aparicio, «Manual de uso del debugger de GNU GDB,» 1995. [En línea]. Available: http://ldc.usb.ve/~figueira/cursos/ci3825/taller/material/gdb.html.