

Índice general

Resumen	I
1. Introducción general	1
1.1. El espacio como recurso estratégico	1
1.2. Radiación cósmica y sus efectos	2
1.3. Calificación espacial e iniciativa <i>new space</i>	4
1.4. Estado del arte	5
1.5. Alcance del trabajo	8
2. Introducción específica	9
2.1. Arquitectura del dispositivo bajo prueba	9
2.2. Servidores y sondas de depuración	11
2.3. Periféricos de interés	13
2.4. Entornos de desarrollo	14
2.5. Requerimientos del cliente	16
3. Diseño e implementación	17
3.1. Autoevaluación del dispositivo bajo prueba	17
3.2. Interfaz de programación de aplicaciones	22
3.3. Sistema de inyección de <i>soft-errors</i>	23
3.4. Biblioteca para el desarrollo de ensayos	26
4. Ensayos y resultados	31
4.1. Laboratorio remoto	31
4.2. Ensayos de inyector	33
4.3. Validación con el cliente	34
5. Conclusiones	37
5.1. Logros obtenidos	37
5.2. Trabajo futuro	38
Bibliografía	39

Índice general

Resumen	I
1. Introducción general	1
1.1. El espacio como recurso estratégico	1
1.2. Radiación cósmica y sus efectos	2
1.3. Calificación espacial e iniciativa <i>New Space</i>	4
1.4. Estado del arte	5
1.5. Alcance del trabajo	8
2. Introducción específica	9
2.1. Arquitectura del dispositivo bajo prueba	9
2.2. Servidores y sondas de depuración	11
2.3. Periféricos de interés	13
2.4. Entornos de desarrollo	14
2.5. Requerimientos del cliente	16
3. Diseño e implementación	17
3.1. Autoevaluación del dispositivo bajo prueba	17
3.2. Interfaz de programación de aplicaciones	22
3.3. Sistema de inyección de <i>soft-errors</i>	23
3.4. Biblioteca para el desarrollo de ensayos	26
4. Ensayos y resultados	31
4.1. Laboratorio remoto	31
4.2. Ensayos de inyector	33
4.3. Validación con el cliente	34
5. Conclusiones	37
5.1. Logros obtenidos	37
5.2. Trabajo futuro	38
Bibliografía	39

Índice de figuras

1.1. Satélite SAOCOM ¹	2
1.2. Capas magnéticas de la tierra y viento solar ²	3
1.3. Ejemplo simplificado de <i>bit flip</i> en un bloque <i>SDRAM</i> ³	3
1.4. Proyección de la constelación <i>Starlink</i> ⁴	5
1.5. Cámara de pruebas de iones pesados ⁵	6
1.6. Diagrama simplificado del dispositivo bajo prueba.	8
1.7. Diagrama simplificado del sistema de inyección de errores.	8
2.1. Diagrama de la arquitectura <i>Cortex M7</i> ⁶	10
2.2. Diagrama del módulo <i>CoreSight</i> ⁷	11
2.3. Conexión de una sesión de depuración.	12
2.4. Sonda de depuración <i>Segger J-32</i> ⁸	13
2.5. Ejemplo del flujo de trabajo Tmux-Neovim.	15
3.1. Diagrama de configuración de las señales de reloj.	17
3.2. Diagrama de <i>loopback</i> del periférico <i>CAN</i> ⁹	18
3.3. Fotografía del dispositivo bajo prueba.	19
3.4. Flujo del <i>firmware</i> de autoevaluación.	21
3.5. Flujo de una sesión de depuración.	24
3.6. Diagrama en bloques del sistema de inyección de soft-errors.	27
3.7. Flujo de tareas concurrentes.	28
4.1. Diagrama en bloques del laboratorio remoto.	32
4.2. Dispositivo alternativo <i>NUCLEO-F429ZI</i>	34
4.3. Demostración de acceso a memoria.	35

Índice de figuras

1.1. Satélite SAOCOM ¹	2
1.2. Capas magnéticas de la tierra y viento solar ²	3
1.3. Ejemplo simplificado de <i>bit flip</i> en un bloque <i>SDRAM</i> ³	3
1.4. Proyección de la constelación Starlink ⁴	5
1.5. Cámara de pruebas de iones pesados ⁵	6
1.6. Diagrama simplificado del dispositivo bajo prueba.	8
1.7. Diagrama simplificado del sistema de inyección de errores.	8
2.1. Diagrama de la arquitectura <i>Cortex M7</i> ⁶	10
2.2. Diagrama del módulo <i>CoreSight</i> ⁷	11
2.3. Conexión de una sesión de depuración.	12
2.4. Sonda de depuración <i>Segger J-32</i> ⁸	13
2.5. Ejemplo del flujo de trabajo Tmux-Neovim.	15
3.1. Diagrama de configuración de las señales de reloj.	17
3.2. Diagrama de <i>loopback</i> del periférico <i>CAN</i> ⁹	18
3.3. Fotografía del dispositivo bajo prueba.	19
3.4. Flujo del <i>firmware</i> de autoevaluación.	21
3.5. Flujo de una sesión de depuración.	24
3.6. Diagrama en bloques del sistema de inyección de soft-errors.	27
3.7. Flujo de tareas concurrentes.	28
4.1. Diagrama en bloques del laboratorio remoto.	32
4.2. Dispositivo alternativo <i>NUCLEO-F429ZI</i>	34
4.3. Demostración de acceso a memoria.	35

Índice de tablas

1.1. Cinturón de Van Allen	2
1.2. Efectos de la radiación cósmica	4
1.3. Proyección de <i>debris</i>	4
1.4. Comparación de métodos de simulación	7
2.1. Servidores de depuración	12
2.2. Resumen de periféricos	14
3.1. Estrategias de depuración	18
3.2. Funcionalidades abstraidas	23
4.1. Resumen del laboratorio remoto	33
4.2. Resumen de ensayos	34
4.3. Resumen de la validación con el cliente	35

Índice de tablas

1.1. Cinturón de Van Allen	2
1.2. Efectos de la radiación cósmica	4
1.3. Proyección de <i>debris</i>	4
1.4. Comparación de métodos de simulación	7
2.1. Servidores de depuración	12
2.2. Resumen de periféricos	14
3.1. Estrategias de depuración	20
3.2. Funcionalidades abstraidas	23
4.1. Resumen del laboratorio remoto	33
4.2. Resumen de ensayos	34
4.3. Resumen de la validación con el cliente	35

FIGURA 1.1. Satélite SAOCOM¹.

1.2. Radiación cósmica y sus efectos

El sol produce partículas de luz e iones pesados que de forma conjunta se denominan viento solar. Este fenómeno es atenuado antes de llegar a la superficie del planeta gracias al campo magnético terrestre [5]. Como se puede ver en la figura 1.2, las partículas son desviadas por el campo. Luego, este queda deformado por el viento solar y se genera una magnetosfera asimétrica. En la tabla 1.1 se puede observar las características de la asimetría.

TABLA 1.1. Cinturón de Van Allen [5].

Cinturón	Frontera	Partícula dominante
Interior	1,2 - 2,5 radios terrestres	Protones de alta energía
Exterior	2,8 - 12 radios terrestres	Electrones de alta energía

La electrónica de los satélites tiene un alto grado de exposición al viento solar. Esto significa que la probabilidad de incidencia de una partícula cargada en el circuito es mayor. La incidencia de una partícula genera una traza densa de pares electrón-hueco en los semiconductores [6]. Además, es posible que esta ionización cause un pulso transitorio de corriente.

Los efectos de la radiación cósmica sobre el circuito pueden ser transitorios o permanentes. Los permanentes se deben a la destrucción de una parte del circuito. Esta destrucción es producto de: el disparo de componentes activos parásitos o la generación de plasma dentro del encapsulado [7]. Finalmente, en la tabla 1.2 se puede ver un resumen de los tipos de errores generados por la radiación cósmica que de forma conjunta se denominan *Single Event Effects (SEE)*.

Los errores transitorios son denominados *soft-errors* y son el objeto de estudio de este trabajo. En la figura 1.3 se puede ver el efecto transitorio de la radiación sobre

¹Imagen tomada de la página oficial de INVAP S.E. [4].

FIGURA 1.1. Satélite SAOCOM¹.

1.2. Radiación cósmica y sus efectos

El sol produce partículas de luz e iones pesados que de forma conjunta se denominan viento solar. Este fenómeno es atenuado antes de llegar a la superficie del planeta gracias al campo magnético terrestre [5]. Como se puede ver en la figura 1.2, las partículas son desviadas por el campo. Luego, este queda deformado por el viento solar y se genera una magnetosfera asimétrica. En la tabla 1.1 se pueden observar las características de la asimetría.

TABLA 1.1. Cinturón de Van Allen [5].

Cinturón	Frontera	Partícula dominante
Interior	1,2 - 2,5 radios terrestres	Protones de alta energía
Exterior	2,8 - 12 radios terrestres	Electrones de alta energía

La electrónica de los satélites tiene un alto grado de exposición al viento solar. Esto significa que la probabilidad de incidencia de una partícula cargada en el circuito es mayor. La incidencia de una partícula genera una traza densa de pares electrón-hueco en los semiconductores [6]. Además, es posible que esta ionización cause un pulso transitorio de corriente.

Los efectos de la radiación cósmica sobre el circuito pueden ser transitorios o permanentes. Los permanentes se deben a la destrucción de una parte del circuito. Esta destrucción es producto de: el disparo de componentes activos parásitos o la generación de plasma dentro del encapsulado [7]. Finalmente, en la tabla 1.2 se puede ver un resumen de los tipos de errores generados por la radiación cósmica que de forma conjunta se denominan *Single Event Effects (SEE)*.

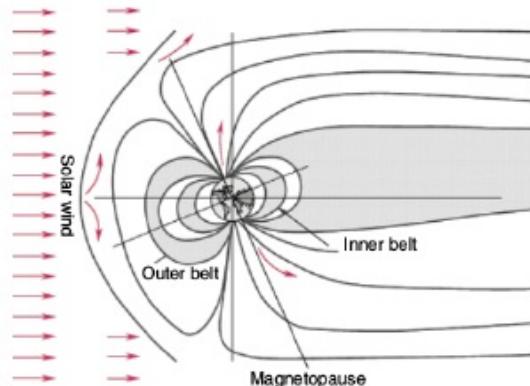
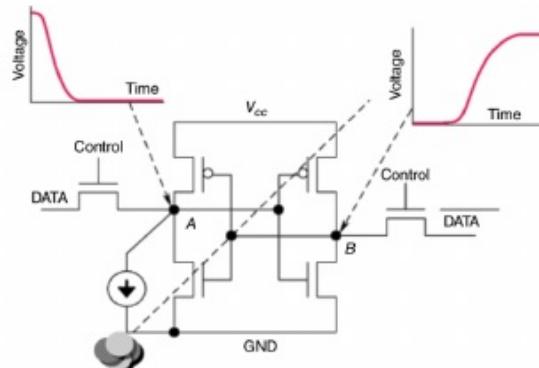
Los errores transitorios son denominados *soft-errors* y son el objeto de estudio de este trabajo. En la figura 1.3 se puede ver el efecto transitorio de la radiación sobre

¹Imagen tomada de la página oficial de INVAP S.E. [4].

1.2. Radiación cósmica y sus efectos

3

los transistores de un integrado. En particular, la inyección de un error denominado *bit flip*. Este se manifiesta como el cambio del valor de un bit en un registro o memoria. En la esquina inferior izquierda de la figura se pueden observar círculos grises claros y oscuros. Estos representan los huecos y electrones generados por el efecto Compton [8] y fotoeléctrico. La línea punteada oblicua representa la traza generada al perturbar los electrones de las uniones covalentes del semiconductor. Esta traza genera perturbaciones en las junturas y logra disparar las compuertas de los transistores. Luego, la activación de los transistores cambian los niveles de tensión en las señales de datos y control. Finalmente, el circuito se normaliza pero con valores invertidos de tensión.

FIGURA 1.2. Capas magnéticas de la tierra y viento solar².FIGURA 1.3. Ejemplo simplificado de *bit flip* en un bloque SDRAM.²Imagen tomada del artículo *Structure of space radiation* [5].

1.2. Radiación cósmica y sus efectos

3

los transistores de un integrado. En particular, la inyección de un error denominado *bit flip*. Este se manifiesta como el cambio del valor de un bit en un registro o memoria. En la esquina inferior izquierda de la figura se pueden observar círculos grises claros y oscuros. Estos representan los huecos y electrones generados por el efecto Compton [8] y fotoeléctrico. La línea punteada oblicua representa la traza generada al perturbar los electrones de las uniones covalentes del semiconductor. Esta traza genera perturbaciones en las junturas y logra disparar las compuertas de los transistores. Luego, la activación de los transistores cambian los niveles de tensión en las señales de datos y control. Finalmente, el circuito se normaliza pero con valores invertidos de tensión.

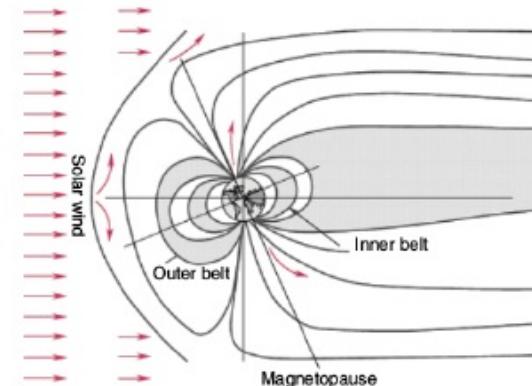
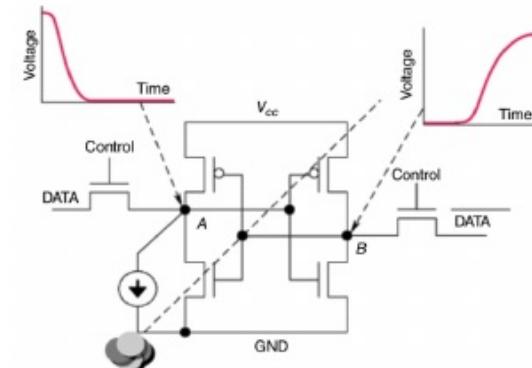
FIGURA 1.2. Capas magnéticas de la tierra y viento solar².FIGURA 1.3. Ejemplo simplificado de *bit flip* en un bloque SDRAM.²Imagen tomada del artículo *Structure of space radiation* [5].

TABLA 1.2. Efectos producidos por la radiación cósmica [5].

Evento	Acrónimo	Efecto
Latch-up	SEL	Pico de corriente
Upset	SEU	Alteración de datos
Functional Interrupt	SEFI	Cambios en la configuración
Transient	SET	Pico de tensión
Burnout	SEB	Activación de transistores parásitos
Gate Rapture	SEGR	Generación de plasma de alta densidad

1.3. Calificación espacial e iniciativa *new space*

A los efectos vistos en la sección 1.2 se suman: el estrés mecánico del lanzamiento y los cambios de temperatura en la órbita. Este ambiente genera la necesidad de utilizar componentes con calificación espacial. Para que un componente alcance la calificación espacial se debe someter a un largo y costoso proceso de acreditación. Luego, estos componentes adolecen de un elevado precio y atraso tecnológico frente a los del mercado masivo [9].

La irrupción del sector privado vista en la sección 1.1, trajo una nueva iniciativa comercial denominada *new space*. Esta iniciativa busca bajar los costos al utilizar componentes no calificados para su uso espacial. Además, existe la ventaja adicional de introducir tecnología de vanguardia.

El caso de *Starlink* es un ejemplo de *new space* particular. Su volumen de satélites lanzados permite realizar conclusiones estadísticas significativas. En particular, su poca capacidad para cumplir sus objetivos si se mantiene la actual tasa de mortalidad de sus satélites [10]. En la figura 1.4 se puede observar que la constelación no logrará alcanzar la población deseada.

Al problema de población de *Starlink* se suma la gran cantidad de polución generada. Los satélites fuera de servicio no pueden ser desorbitados y persisten en forma de *debris*. Como se puede ver en la tabla 1.3, el volumen de basura generado es significativo.

TABLA 1.3. Proyección de *debris* de *Starlink* [10].

Lanzamientos	Satélites	Total lanzados	Población	Debris
12	60	7200	2704	4046
12	180	21600	8105	12146
12	400	48000	18007	26994
180	60	108000	40000	61200
60	180	108000	40000	61200
27	400	108000	40000	61200

TABLA 1.2. Efectos producidos por la radiación cósmica [5].

Evento	Acrónimo	Efecto
<i>Latch-up</i>	SEL	Pico de corriente
<i>Upset</i>	SEU	Alteración de datos
<i>Functional Interrupt</i>	SEFI	Cambios en la configuración
<i>Transient</i>	SET	Pico de tensión
<i>Burnout</i>	SEB	Activación de transistores parásitos
<i>Gate Rapture</i>	SEGR	Generación de plasma de alta densidad

1.3. Calificación espacial e iniciativa *New Space*

A los efectos vistos en la sección 1.2 se suman: el estrés mecánico del lanzamiento y los cambios de temperatura en la órbita. Este ambiente genera la necesidad de utilizar componentes con calificación espacial. Para que un componente alcance la calificación espacial se debe someter a un largo y costoso proceso de acreditación. Luego, estos componentes adolecen de un elevado precio y atraso tecnológico frente a los del mercado masivo [9].

La irrupción del sector privado vista en la sección 1.1, trajo una nueva iniciativa comercial denominada *New Space*. Esta iniciativa busca bajar los costos al utilizar componentes no calificados para su uso espacial. Además, existe la ventaja adicional de introducir tecnología de vanguardia.

El caso de *Starlink* es un ejemplo de *New Space* particular. Su volumen de satélites lanzados permite realizar conclusiones estadísticas significativas. En particular, su poca capacidad para cumplir sus objetivos si se mantiene la actual tasa de mortalidad de sus satélites [10]. En la figura 1.4 se puede observar que la constelación no logrará alcanzar la población deseada.

Al problema de población de *Starlink* se suma la gran cantidad de polución generada. Los satélites fuera de servicio no pueden ser desorbitados y persisten en forma de *debris*. Como se puede ver en la tabla 1.3, el volumen de basura generado es significativo.

TABLA 1.3. Proyección de *debris* de *Starlink* [10].

Lanzamientos	Satélites	Total lanzados	Población	Debris
12	60	7200	2704	4046
12	180	21600	8105	12146
12	400	48000	18007	26994
180	60	108000	40000	61200
60	180	108000	40000	61200
27	400	108000	40000	61200

⁵Imagen tomada del artículo *Effects of space radiation on electronic devices* [7].

⁵Imagen tomada del artículo *Effects of space radiation on electronic devices* [7].

1.4. Estado del arte

7

Donde:

- $d(R_i)$ es el ciclo de trabajo del elemento de memoria R_i .
- σ_{R_i} es la sección trasversal de memoria obtenida por la metodología estática [6].

Como se puede ver en la ecuación 1.1 diseñar un ensayo dinámico por radiación es un proceso largo y costoso. Demanda personal capacitado e instalaciones específicas. Además, la electrónica de vuelo no suele estar presente durante el desarrollo del proyecto. Finalmente, este método no permite un control preciso del ensayo.

Otra técnica disponible son los ensayos basados en software. Este método se basa en introducir instrucciones espúreas que generen errores. Esto genera la problemática de estimar la tasa de error. Con esta tasa se puede determinar en qué posiciones del programa corresponde introducir las instrucciones de error. La tasa de error se estima como:

$$\tau_{SEU} = \sigma_{SEU} \times \tau_{inj} \quad (1.2)$$

Donde τ_{inj} es la tasa de incidencia de una partícula cargada.

Esta técnica demanda ciclos de la unidad de proceso y por lo tanto consume más tiempo de ejecución. Además, cada vez que se altera el código fuente de la aplicación se necesita volver a diseñar una serie nueva de ensayos.

Existe una tercera técnica de ensayo de errores transitorios. Esta técnica está basada en hardware; y en el caso de los microprocesadores, se utiliza una sonda de depuración. La principal ventaja de este método frente al basado en software es que un mismo ensayo puede ser utilizado para múltiples iteraciones del código fuente. Esto es posible al suponer un flujo constante de partículas cargadas, de esta manera, los errores siguen un proceso Poisson homogéneo. Luego, el intervalo de tiempo entre dos errores transitorios expresa una distribución exponencial. Este razonamiento está sustentado en la siguiente ecuación:

$$P(N_{SEU}(t + \Delta t) = N_{SEU}(t)) = e^{-\sigma \times \phi \times \Delta t} \quad (1.3)$$

El trabajo realizado es una solución del tipo ensayo por hardware. Además, se propuso superar el estado del arte de este método al crear una abstracción para el diseño de ensayos. Los métodos mencionados presentan compromisos de ingeniería y están resumidos en la tabla 1.4.

TABLA 1.4. Comparación de métodos de simulación [6].

Método	Eficiencia	Costo	Limitación
Software	Baja	Bajo	Ciclos de CPU
Hardware	Media	Medio	Acceso al integrado
Radiación	Alta	Alto	Control del ensayo

1.4. Estado del arte

7

Donde:

- $d(R_i)$ es el ciclo de trabajo del elemento de memoria R_i .
- σ_{R_i} es la sección trasversal de memoria obtenida por la metodología estática [6].

Diseñar un ensayo dinámico por radiación es un proceso largo y costoso, demanda personal capacitado e instalaciones específicas. Además, la electrónica de vuelo no suele estar presente durante el desarrollo del proyecto. Finalmente, este método no permite un control preciso del ensayo.

Otra técnica disponible son los ensayos basados en software, cuyo método se basa en introducir instrucciones espúreas que generen errores. Esto genera la problemática de estimar la tasa de error. Con esta tasa se puede determinar en qué posiciones del programa corresponde introducir las instrucciones de error. La tasa de error se estima como:

$$\tau_{SEU} = \sigma_{SEU} \times \tau_{inj} \quad (1.2)$$

Donde τ_{inj} es la tasa de incidencia de una partícula cargada.

Esta técnica demanda ciclos de la unidad de proceso y por lo tanto consume más tiempo de ejecución. Además, cada vez que se altera el código fuente de la aplicación se necesita volver a diseñar una serie nueva de ensayos.

Existe una tercera técnica de ensayo de errores transitorios. Esta técnica está basada en hardware; y en el caso de los microprocesadores, se utiliza una sonda de depuración. La principal ventaja de este método frente al basado en software es que un mismo ensayo puede ser utilizado para múltiples iteraciones del código fuente. Esto es posible al suponer un flujo constante de partículas cargadas, de esta manera, los errores siguen un proceso Poisson homogéneo. Luego, el intervalo de tiempo entre dos errores transitorios expresa una distribución exponencial. Este razonamiento está sustentado en la siguiente ecuación:

$$P(N_{SEU}(t + \Delta t) = N_{SEU}(t)) = e^{-\sigma \times \phi \times \Delta t} \quad (1.3)$$

El trabajo realizado es una solución del tipo ensayo por hardware. Además, se propuso superar el estado del arte de este método al crear una abstracción para el diseño de ensayos. Los métodos mencionados presentan compromisos de ingeniería y están resumidos en la tabla 1.4.

TABLA 1.4. Comparación de métodos de simulación [6].

Método	Eficiencia	Costo	Limitación
Software	Baja	Bajo	Ciclos de CPU
Hardware	Media	Medio	Acceso al integrado
Radiación	Alta	Alto	Control del ensayo

1.5. Alcance del trabajo

El trabajo realizado se divide en dos partes:

1. *Firmware* para el dispositivo bajo prueba.
2. Inyector de *soft-errors* por consola de comandos.

El *firmware* en el dispositivo bajo prueba tiene la misión de validar su funcionamiento. Esto se logró al verificar cada periférico de interés dentro del integrado. Luego, se generan reportes periódicos que se envían al inyector por consola de comandos. En la figura 1.6 se puede observar un diagrama en bloques simplificado.

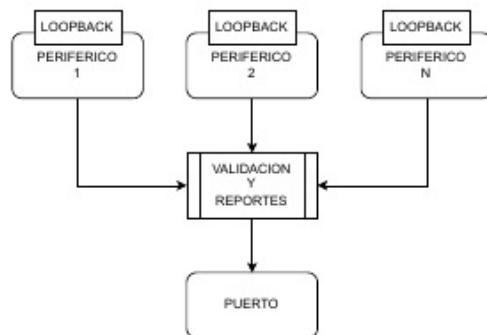


FIGURA 1.6. Diagrama simplificado del dispositivo bajo prueba.

El inyector por consola de comandos tiene la función de planificar los ensayos y gestionar la introducción de errores. En la figura 1.7 se puede ver como interactúan las partes del sistema.

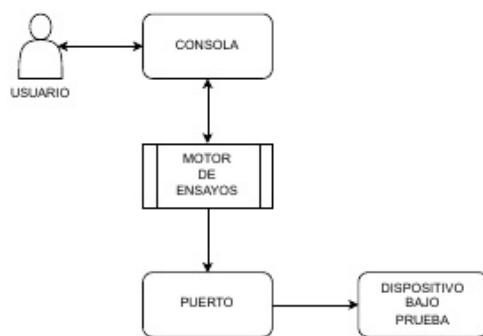


FIGURA 1.7. Diagrama simplificado del sistema de inyección de errores.

1.5. Alcance del trabajo

El trabajo realizado se divide en dos partes:

1. *Firmware* para el dispositivo bajo prueba.
2. Inyector de *soft-errors* por consola de comandos.

El *firmware* en el dispositivo bajo prueba tiene la misión de validar su funcionamiento. Esto se logró al verificar cada periférico de interés dentro del integrado. Luego, se generan reportes periódicos que se envían al inyector por consola de comandos.

En la figura 1.6 se puede observar un diagrama en bloques simplificado.

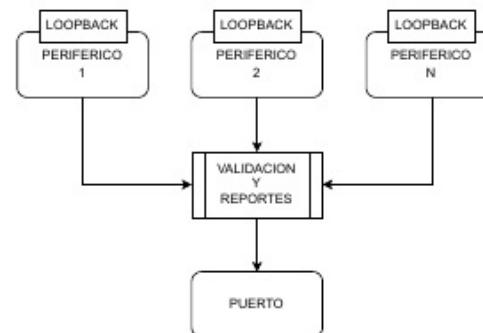


FIGURA 1.6. Diagrama simplificado del dispositivo bajo prueba.

El inyector por consola de comandos tiene la función de planificar los ensayos y gestionar la introducción de errores. En la figura 1.7 se puede ver como interactúan las partes del sistema.

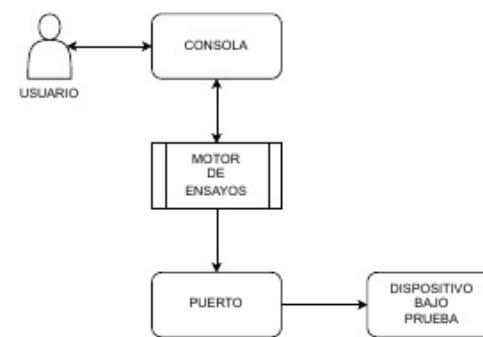


FIGURA 1.7. Diagrama simplificado del sistema de inyección de errores.

Capítulo 2

Introducción específica

En este capítulo se detallan las tecnologías que forman parte del trabajo. Son productos de terceros que se integran en las herramientas entregadas al cliente.

2.1. Arquitectura del dispositivo bajo prueba

El trabajo fue realizado para un tipo de microcontrolador específico. Su diseño forma parte de la familia *Cortex M7* de la empresa *ARM*. En la figura 2.1 se puede observar un diagrama en bloques de la arquitectura.

El dispositivo bajo prueba es el microcontrolador *SAM V71* diseñado por la empresa *Atmel* y comercializado por *Microchip*. El integrado fue pensado para aplicaciones automotrices según el estándar *ISO-TS-16949*. Además, el circuito puede operar con un reloj de 300 MHz y almacenar un programa de 2048 kB. Las estructuras de datos del programa pueden aprovechar la memoria *cache* dual de 16 kB [12]. Las principales características del dispositivo bajo prueba son:

- Núcleo:
 - Unidad de punto flotante de precisión simple y doble.
 - Unidad de protección de memoria con 16 zonas.
 - Instrucciones para el procesamiento digital de señales.
- Memorias:
 - ROM de 16 kB con rutinas de inicialización. Esto permite iniciar el sistema desde los periféricos *UART0* y *USB*.
 - Controlador de memoria estática para el uso de memorias externas.
- Sistema:
 - Reloj de tiempo real con gestión de calendario gregoriano.
 - Reinicio por alimentación, detección de caída de tensión y doble *Watchdog*.
 - Puerto dual de 24 canales para la gestión de acceso a memoria.
 - Compensación por variaciones de reloj.

Capítulo 2

Introducción específica

En este capítulo se detallan las tecnologías que forman parte del trabajo. Son productos de terceros que se integran en las herramientas entregadas al cliente.

2.1. Arquitectura del dispositivo bajo prueba

El trabajo fue realizado para un tipo de microcontrolador específico. Su diseño forma parte de la familia *Cortex M7* de la empresa *ARM*. En la figura 2.1 se puede observar un diagrama en bloques de la arquitectura.

El dispositivo bajo prueba es el microcontrolador *SAM V71* diseñado por la empresa Atmel y comercializado por Microchip. El integrado fue pensado para aplicaciones automotrices según el estándar *ISO-TS-16949*. Además, el circuito puede operar con un reloj de 300 MHz y almacenar un programa de 2048 kB. Las estructuras de datos del programa pueden aprovechar la memoria *caché* dual de 16 kB [12].

Las principales características del dispositivo bajo prueba son:

- Núcleo:
 - Unidad de punto flotante de precisión simple y doble.
 - Unidad de protección de memoria con 16 zonas.
 - Instrucciones para el procesamiento digital de señales.
- Memorias:
 - ROM de 16 kB con rutinas de inicialización. Esto permite iniciar el sistema desde los periféricos *UART0* y *USB*.
 - Controlador de memoria estática para el uso de memorias externas.
- Sistema:
 - Reloj de tiempo real con gestión de calendario gregoriano.
 - Reinicio por alimentación, detección de caída de tensión y doble *Watchdog*.
 - Puerto dual de 24 canales para la gestión de acceso a memoria.
 - Compensación por variaciones de reloj.

Este integrado fue sometido a una prueba por radiación donde se evaluaron SEE y la calificación de dosis total de ionización (TID). El microcontrolador mantuvo su funcionamiento en todo el rango de temperatura de calificación militar. Además, el dispositivo es inmune a *Single Event Latch-up* con una tolerancia de 60,0 MeV.cm²/mg. Esta sensibilidad fue probada con una cámara de iones pesados. En cuanto a la calificación de dosis total de ionización, el lote fue sometido a un ensayo de 30 krad(Si) y la prueba fue superada. Finalmente, los ensayos suministrados por el fabricante permiten concluir que no es necesario realizar inyecciones de errores en la memoria *flash* [13].

Al fabricante del dispositivo bajo prueba se le impone respetar el mapa de memoria y registros del núcleo. Esto permitió construir un inyector de *soft-errors* genérico. Finalmente, la herramienta entregada funciona para cualquier integrado de la familia Cortex M.

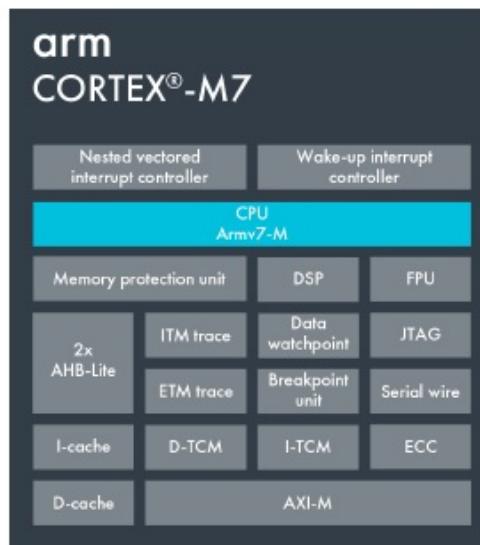


FIGURA 2.1. Diagrama de la arquitectura Cortex M7¹.

La arquitectura tiene un módulo que permite programar y depurar el integrado. Este módulo se denomina *CoreSight* y es propio de los dispositivos ARM. En la figura 2.2 se muestra un diagrama en bloques del módulo. Sus partes principales son:

- **Cross Triggering**: permite conectar y encaminar las señales que utilizan las sondas de depuración. En la figura 2.2 está representada en los bloques *CTI*. Además, se unen a través del *Cross Trigger Matrix (CTM)*.
- **Debug Access Port (DAP)**: es el puerto físico para conectar la sonda de depuración. Es una implementación de la interfaz de depuración ARM.

¹Imagen tomada de la página oficial de ARM Developers. [14]

Este integrado fue sometido a una prueba por radiación donde se evaluaron SEE y la calificación de dosis total de ionización (TID). El microcontrolador mantuvo su funcionamiento en todo el rango de temperatura de calificación militar. Además, el dispositivo es inmune a *Single Event Latch-up* con una tolerancia de 60,0 MeV.cm²/mg. Esta sensibilidad fue probada con una cámara de iones pesados.

En cuanto a la calificación de dosis total de ionización, el lote fue sometido a un ensayo de 30 krad(Si) y la prueba fue superada. Finalmente, los ensayos suministrados por el fabricante permiten concluir que no es necesario realizar inyecciones de errores en la memoria *flash* [13].

Al fabricante del dispositivo bajo prueba se le impone respetar el mapa de memoria y registros del núcleo. Esto permitió construir un inyector de *soft-errors* genérico. Finalmente, la herramienta entregada funciona para cualquier integrado de la familia Cortex M.

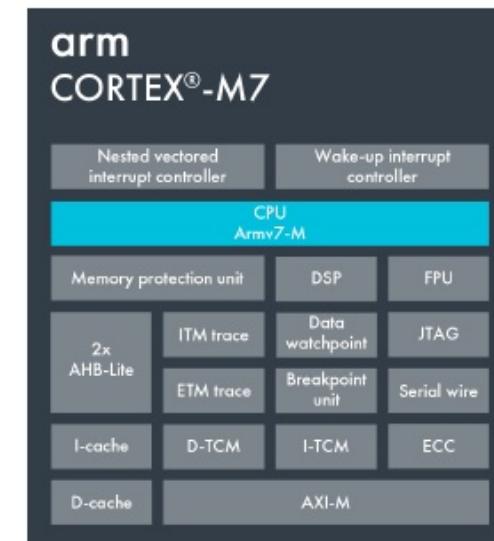


FIGURA 2.1. Diagrama de la arquitectura Cortex M7¹.

La arquitectura tiene un módulo que permite programar y depurar el integrado. Este módulo se denomina *CoreSight* y es propio de los dispositivos ARM.

En la figura 2.2 se muestra un diagrama en bloques del módulo. Sus partes principales son:

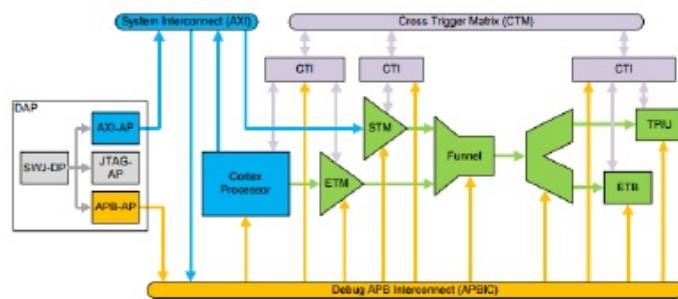
- **Cross Triggering**: permite conectar y encaminar las señales que utilizan las sondas de depuración. En la figura 2.2 está representada en los bloques *CTI*. Además, se unen a través del *Cross Trigger Matrix (CTM)*.

¹Imagen tomada de la página oficial de ARM Developers. [14]

2.2. Servidores y sondas de depuración

11

- *Embedded Trace Macrocells*: permite extraer información y controlar el núcleo del dispositivo.
- *Instrumentation Trace Units*: permite que una sonda de depuración se conecte con las *Embedded Trace Macrocells*.
- *ROM Tables*: sirven para que la sonda de depuración identifique al integrado.
- *Self Hosted Debug*: son instrucciones específicas de depuración controladas por un procesador secundario.
- *Trace Interconnect*: provee puentes para compartir señales de reloj, alimentación y otras señales comunes.

FIGURA 2.2. Diagrama del módulo CoreSight².

2.2. Servidores y sondas de depuración

Una sesión de depuración sirve para observar y modificar el estado de ejecución de un programa. Esto se logra al leer y modificar los valores en registros del procesador y periféricos. Además, se necesita de un sistema de disparos por eventos y supervisión de recursos. Finalmente, la sesión debe detener la ejecución del núcleo de ser necesario. En la figura 2.3 se puede observar un esquema simplificado de una sesión de depuración.

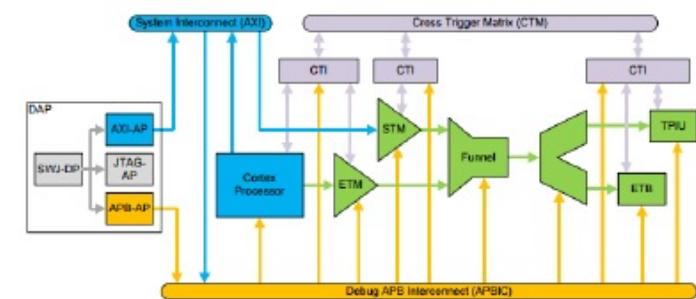
Un servidor *On-chip debugger* (OCD) tiene la misión de abstraer la conexión de la sonda de depuración. Además, facilita el manejo del ciclo de vida de la sesión y permite usar un *software* como *GNU Project debugger* (GDB). Finalmente, es la base de una pila de tecnologías que permite el uso de herramientas como *GNU Emacs* (*Emacs*) [16]. En la tabla 2.1 se puede observar un resumen de los servidores evaluados en el trabajo.

El servidor OCD utilizado en este trabajo es PyOCD. La principal característica que lo diferencia es el uso de Python 3 como lenguaje de *scripting*. Además, provee un servidor GDB, permite la programación de memoria *flash* y ofrece una interfaz por consola de comandos [17]. Finalmente, los datos más relevantes son:

2.2. Servidores y sondas de depuración

11

- *Debug Access Port (DAP)*: es el puerto físico para conectar la sonda de depuración. Es una implementación de la interfaz de depuración ARM.
- *Embedded Trace Macrocells*: permite extraer información y controlar el núcleo del dispositivo.
- *Instrumentation Trace Units*: permite que una sonda de depuración se conecte con las *Embedded Trace Macrocells*.
- *ROM Tables*: sirven para que la sonda de depuración identifique al integrado.
- *Self Hosted Debug*: son instrucciones específicas de depuración controladas por un procesador secundario.
- *Trace Interconnect*: provee puentes para compartir señales de reloj, alimentación y otras señales comunes.

FIGURA 2.2. Diagrama del módulo CoreSight².

2.2. Servidores y sondas de depuración

Una sesión de depuración sirve para observar y modificar el estado de ejecución de un programa. Esto se logra al leer y modificar los valores en registros del procesador y periféricos. Además, se necesita de un sistema de disparos por eventos y supervisión de recursos. Finalmente, la sesión debe detener la ejecución del núcleo de ser necesario. En la figura 2.3 se puede observar un esquema simplificado de una sesión de depuración.

Un servidor *On-chip debugger* (OCD) tiene la misión de abstraer la conexión de la sonda de depuración. Además, facilita el manejo del ciclo de vida de la sesión y permite usar un *software* como *GNU Project debugger* (GDB). Finalmente, es la base de una pila de tecnologías que permite el uso de herramientas como *GNU Emacs* [16].

En la tabla 2.1 se puede observar un resumen de los servidores evaluados en el trabajo.

²Imagen tomada del artículo *How to debug: CoreSight basis* [15].

²Imagen tomada del artículo *How to debug: CoreSight basis* [15].

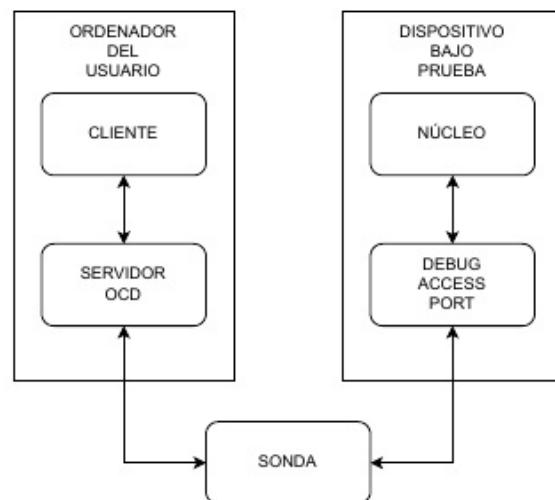


FIGURA 2.3. Conexión de una sesión de depuración.

TABLA 2.1. Comparativa entre servidores de depuración.

Servidor	API	Acceso	Licencia
OpenOCD	tcl	Registros y SDRAM	MIT
PyOCD	Python 3	Registros y SDRAM	Apache-2.0

- Requerimientos:
 - Python 3.6.0 o superior.
 - Una versión reciente de libusb.
 - macOS, GNU Linux, Windows 7 o FreeBSD.
- Sondas de depuración soportadas:
 - Atmel EDBG/nEDBG.
 - Atmel-ICE.
 - Cypress KitProg3 o MiniProg4.
 - DAPLink.
 - Keil ULINKplus.
 - NXP LPC-LinkII
 - NXP MCU-Link
 - PE Micro Cyclone y Multilink.
 - Raspberry Pi Picoprobe.

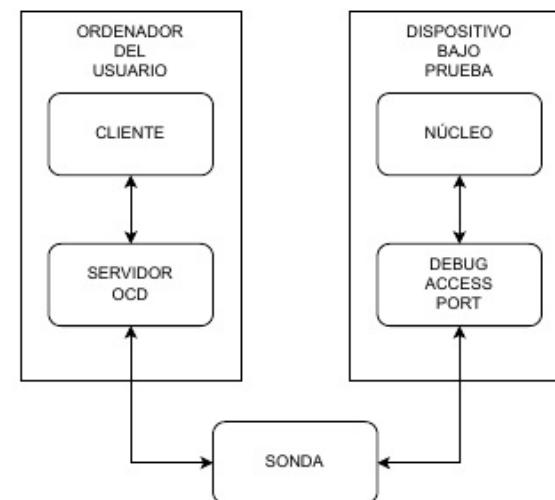


FIGURA 2.3. Conexión de una sesión de depuración.

TABLA 2.1. Comparativa entre servidores de depuración.

Servidor	API	Acceso	Licencia
OpenOCD	tcl	Registros y SDRAM	MIT
PyOCD	Python 3	Registros y SDRAM	Apache-2.0

El servidor OCD utilizado en este trabajo es PyOCD. La principal característica que lo diferencia es el uso de Python 3 como lenguaje de scripting. Además, provee un servidor GDB, permite la programación de memoria flash y ofrece una interfaz por consola de comandos [17]. Finalmente, los datos mas relevantes son:

- Requerimientos:
 - Python 3.6.0 o superior.
 - Una versión reciente de libusb.
 - macOS, GNU Linux, Windows 7 o FreeBSD.
- Sondas de depuración soportadas:
 - Atmel EDBG/nEDBG.
 - Atmel-ICE.
 - Cypress KitProg3 o MiniProg4.
 - DAPLink.
 - Keil ULINKplus.
 - NXP LPC-LinkII

2.3. Periféricos de interés

13

- SEGGER J-Link.
- STLinkV2 y SRLinkV3.

Las sondas de depuración tienen el objetivo de conectar el *Debug Access Port* con el puerto del ordenador del usuario. Adaptan los niveles de tensión y los protocolos involucrados. Luego, permiten realizar una sesión de depuración, programar el dispositivo o verificar el estado de los componentes en la placa. En la figura 2.4 se puede ver la sonda provista por el cliente.



FIGURA 2.4. Sonda de depuración Segger J-32³.

2.3. Periféricos de interés

El dispositivo bajo prueba ofrece una variedad de periféricos para el desarrollo de aplicaciones. Sin embargo, el cliente manifestó interés solo en los que se nombran a continuación:

- CAN: este periférico permite al microcontrolador ser el dispositivo principal en una *Controller Area Network*. La red es de grado industrial y fue diseñada para gestionar una red de sensores en un ambiente automotriz.
- PIO: es el puerto de entradas y salidas digitales de propósito general. En el caso del dispositivo bajo prueba, el periférico permite usar circuitos anti rebote, *pull-up* y *pull-down* internos.
- SPI: el periférico permite realizar una conexión del tipo *Serial Peripheral Interface*. Esta conexión es sincrónica y solo apta para distancias cortas.
- UART: es un periférico que permite conectarse a puertos y controlar dispositivos serie.
- Watchdog: el periférico sirve para detectar un error de ejecución y reiniciar el microporcesador.

En la tabla 2.2 se resume la funcionalidad de cada uno de ellos.

2.3. Periféricos de interés

13

- NXP MCU-Link
- PE Micro Cyclone y Multilink.
- Raspberry Pi Picoprobe.
- SEGGER J-Link.
- STLinkV2 y SRLinkV3.

Las sondas de depuración tienen el objetivo de conectar el *Debug Access Port* con el puerto del ordenador del usuario. Adaptan los niveles de tensión y los protocolos involucrados. Luego, permiten realizar una sesión de depuración, programar el dispositivo o verificar el estado de los componentes en la placa. En la figura 2.4 se puede ver la sonda provista por el cliente.



FIGURA 2.4. Sonda de depuración Segger J-32³.

2.3. Periféricos de interés

El dispositivo bajo prueba ofrece una variedad de periféricos para el desarrollo de aplicaciones. Sin embargo, el cliente manifestó interés solo en los que se nombran a continuación:

- CAN: este periférico permite al microcontrolador ser el dispositivo principal en una *Controller Area Network*. La red es de grado industrial y fue diseñada para gestionar una red de sensores en un ambiente automotriz.
- PIO: es el puerto de entradas y salidas digitales de propósito general. En el caso del dispositivo bajo prueba, el periférico permite usar circuitos anti rebote, *pull-up* y *pull-down* internos.
- SPI: el periférico permite realizar una conexión del tipo *Serial Peripheral Interface*. Esta conexión es sincrónica y solo apta para distancias cortas.
- UART: es un periférico que permite conectarse a puertos y controlar dispositivos serie.
- Watchdog: el periférico sirve para detectar un error de ejecución y reiniciar el microporcesador.

³Imagen tomada de <https://www.digikey.com/>

³Imagen tomada de <https://www.digikey.com/>

TABLA 2.2. Resumen de periféricos.

Periférico	Funcionalidad
CAN	Bus de comunicación de grado industrial
PIO	Entradas y salidas digitales
SPI	Interfaz de comunicación sincrónica
UART	Puerto para dispositivos serie
Watchdog	Detección de errores y reinicio del integrado

2.4. Entornos de desarrollo

Para escribir el código que corre en el dispositivo bajo prueba se utilizó un entorno integrado de desarrollo (IDE). Este IDE es MPLAB y fue provisto por el fabricante del integrado. MPLAB está compuesto por una colección de programas que trabajan como un único sistema. Entre ellos se encuentran:

- Compilador para lenguaje C.
- Biblioteca CMSIS de ARM.
- Biblioteca HARMONY 3 de Microchip.
- Herramienta gráfica para la planificación de terminales.
- Herramienta gráfica para la configuración de periféricos.
- Herramienta gráfica para la configuración de reloj.
- Cliente GDB para sesiones de depuración.

Para realizar el código del inyector por consola de comandos se utilizó el lenguaje de programación Python 3. Es un lenguaje interpretado que permite escribir código portable. Además, el intérprete tiene la capacidad de crear ambientes virtuales. Un ambiente virtual es un espacio de trabajo donde las dependencias instaladas quedan encapsuladas. De esta manera, se puede simular el despliegue en un ambiente de producción. Finalmente, junto al intérprete se utilizó un gestor de paquetes llamado PIP. Esto facilitó la instalación automática del sistema.

Para escribir el código en Python 3, el *firmware* en C y esta memoria en *LATEX*, se utilizó el editor de texto Neovim. Este programa está basado en el editor Vi de los sistemas Unix. Su funcionamiento es modal, esto significa que el editor funciona en los siguientes modos:

- Modo normal:
 - Navegar el documento.
 - Ejecutar comandos de consola con la posibilidad de volcar el *standard output* en el documento.
 - Ejecutar *scripts* de Neovim que permiten, por ejemplo, ordenar alfabéticamente una lista.
 - Ejecutar búsquedas y reemplazos con comandos *sed*.
 - **Grabar y ejecutar macros.**
- **Modo inserción:** permite escribir en el documento.

En la tabla 2.2 se resume la funcionalidad de cada uno de ellos.

TABLA 2.2. Resumen de periféricos.

Periférico	Funcionalidad
CAN	Bus de comunicación de grado industrial
PIO	Entradas y salidas digitales
SPI	Interfaz de comunicación sincrónica
UART	Puerto para dispositivos serie
Watchdog	Detección de errores y reinicio del integrado

2.4. Entornos de desarrollo

Para escribir el código que corre en el dispositivo bajo prueba se utilizó un entorno integrado de desarrollo (IDE). Este IDE es MPLAB y fue provisto por el fabricante del integrado. MPLAB está compuesto por una colección de programas que trabajan como un único sistema. Entre ellos se encuentran:

- Compilador para lenguaje C.
- Biblioteca CMSIS de ARM.
- Biblioteca HARMONY 3 de Microchip.
- Herramienta gráfica para la planificación de terminales.
- Herramienta gráfica para la configuración de periféricos.
- Herramienta gráfica para la configuración de reloj.
- Cliente GDB para sesiones de depuración.

Para realizar el código del inyector por consola de comandos se utilizó el lenguaje de programación Python 3. Es un lenguaje interpretado que permite escribir código portable. Además, el intérprete tiene la capacidad de crear ambientes virtuales. Un ambiente virtual es un espacio de trabajo donde las dependencias instaladas quedan encapsuladas. De esta manera, se puede simular el despliegue en un ambiente de producción. Finalmente, junto al intérprete se utilizó un gestor de paquetes llamado PIP. Esto facilitó la instalación automática del sistema.

Para escribir el código en Python 3, el *firmware* en C y esta memoria en *LATEX*, se utilizó el editor de texto Neovim. Este programa está basado en el editor Vi de los sistemas Unix. Su funcionamiento es modal, esto significa que el editor funciona en los siguientes modos:

- Modo normal:
 - Navegar el documento.
 - Ejecutar comandos de consola con la posibilidad de volcar el *standard output* en el documento.
 - Ejecutar *scripts* de Neovim que permiten, por ejemplo, ordenar alfabéticamente una lista.
 - Ejecutar búsquedas y reemplazos con comandos *sed*.

2.4. Entornos de desarrollo

15

- Modo visual: permite seleccionar bloques del documento para aplicar comandos.
- Modo terminal: es un *buffer* que emula una terminal Unix.

Neovim tiene la capacidad de conectarse a un servidor de análisis sintáctico de un lenguaje en particular. Esto lo logra a través del *Language Server Protocol (LSP)*. El protocolo permite que un *demonio* realice el análisis de la sintaxis del *código* y envíe al editor información sobre errores y advertencias. De esta manera se separa al editor del análisis sintáctico del lenguaje. Además, Neovim tiene incorporado los diccionarios de la mayoría de los idiomas. Con solo ejecutar :set spelllang=es y :set spell, el editor resalta las palabras que no estén escritas en correcto castellano.

El último elemento del flujo de trabajo es el multiplexor de terminal Tmux. Este programa permite dividir la terminal, crear *buffers* y crear o conectarse a sesiones locales y remotas. Esto posibilita partir una terminal y trabajar en simultáneo en dos o más ordenadores. Finalmente, se trabajó de forma integrada con un ambiente de laboratorio remoto y sistemas de desarrollo locales. En la figura 2.5 se puede ver un ejemplo del flujo de trabajo.

```
[memoria] 1:editor 2:compiler 3:demo* "laptop" 17:12 15-may-22
  1 def main():
  2     print("Demostración de LSP")
  3
  4 if __name__ == "__main__":
  5     main();           # E703 statement ends with a semicolon
  6     # W391 blank line at end of file
  demo.py          1,5      Todo
  gonzalo@laptop:~/Documentos/FIUBA/taller/MIoT-Memoria$ > echo "Demostración Tmux"
  Demostración Tmux
  gonzalo@laptop:~/Documentos/FIUBA/taller/MIoT-Memoria$ >
```

FIGURA 2.5. Ejemplo del flujo de trabajo Tmux-Neovim.

2.4. Entornos de desarrollo

15

- Grabar y ejecutar macros.
- Modo inserción: permite escribir en el documento.
- Modo visual: permite seleccionar bloques del documento para aplicar comandos.
- Modo terminal: es un *buffer* que emula una terminal Unix.

Neovim tiene la capacidad de conectarse a un servidor de análisis sintáctico de un lenguaje en particular. Esto lo logra a través del *Language Server Protocol (LSP)*. El protocolo permite que un *daemon* realice el análisis de la sintaxis del *código* y envíe al editor información sobre errores y advertencias. De esta manera se separa al editor del análisis sintáctico del lenguaje. Además, Neovim tiene incorporado los diccionarios de la mayoría de los idiomas. Con solo ejecutar :set spelllang=es y :set spell, el editor resalta las palabras que no estén escritas en correcto castellano.

El último elemento del flujo de trabajo es el multiplexor de terminal Tmux. Este programa permite dividir la terminal, crear *buffers* y crear o conectarse a sesiones locales y remotas. Esto posibilita partir una terminal y trabajar en simultáneo en dos o más ordenadores. Finalmente, se trabajó de forma integrada con un ambiente de laboratorio remoto y sistemas de desarrollo locales.

En la figura 2.5 se puede ver un ejemplo del flujo de trabajo.

```
[memoria] 1:editor 2:compiler 3:demo* "laptop" 17:12 15-may-22
  1 def main():
  2     print("Demostración de LSP")
  3
  4 if __name__ == "__main__":
  5     main();           # E703 statement ends with a semicolon
  6     # W391 blank line at end of file
  demo.py          1,5      Todo
  gonzalo@laptop:~/Documentos/FIUBA/taller/MIoT-Memoria$ > echo "Demostración Tmux"
  Demostración Tmux
  gonzalo@laptop:~/Documentos/FIUBA/taller/MIoT-Memoria$ >
```

FIGURA 2.5. Ejemplo del flujo de trabajo Tmux-Neovim.

Capítulo 3

Diseño e implementación

Este capítulo detalla la generación de contenido original del trabajo. Se explica su diseño y producción.

3.1. Autoevaluación del dispositivo bajo prueba

La construcción del *firmware* de autoevaluación del dispositivo bajo prueba requirió superar las siguientes etapas:

- Configuración de las señales de reloj.
- Selección y configuración de los periféricos.
- Selección y configuración de los terminales externos.
- Implementación de las estrategias de validación de periféricos.
- Integración de una secuencia de validación y reporte.

Para configurar las frecuencias de reloj se buscó obtener 150 MHz para suministrar al *Master CAN Bus*. Con esta condición satisfecha, se pudo configurar las frecuencias de reloj del resto de los periféricos. En la figura 3.1 se puede observar la utilización del *Programmable Clock Controller* número cinco.

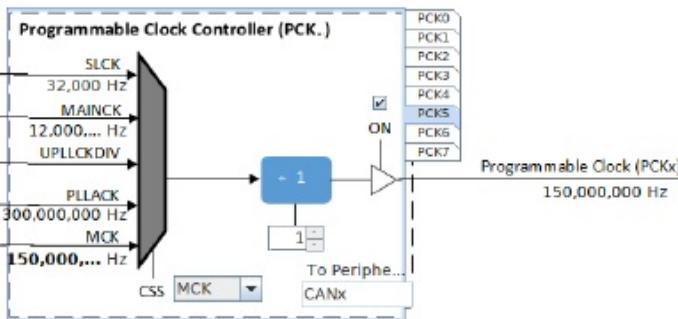


FIGURA 3.1. Diagrama de configuración de las señales de reloj.

El siguiente paso en la etapa de diseño fue la selección de las instancias de los periféricos del integrado. Es posible que dos periféricos comparten parte del circuito

Capítulo 3

Diseño e implementación

Este capítulo detalla la generación de contenido original del trabajo. Se explica su diseño y producción.

3.1. Autoevaluación del dispositivo bajo prueba

La construcción del *firmware* de autoevaluación del dispositivo bajo prueba requirió superar las siguientes etapas:

- Configuración de las señales de reloj.
- Selección y configuración de los periféricos.
- Selección y configuración de los terminales externos.
- Implementación de las estrategias de validación de periféricos.
- Integración de una secuencia de validación y reporte.

Para configurar las frecuencias de reloj se buscó obtener 150 MHz para suministrar al *Master CAN Bus*. Con esta condición satisfecha, se pudo configurar las frecuencias de reloj del resto de los periféricos.

En la figura 3.1 se puede observar la utilización del *Programmable Clock Controller* número cinco.

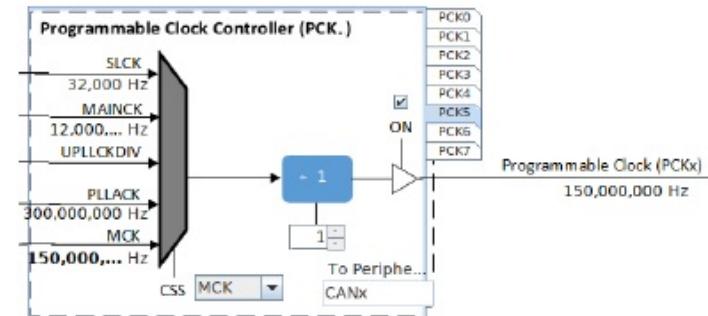


FIGURA 3.1. Diagrama de configuración de las señales de reloj.

interno o terminales del encapsulado. Esta situación puede generar una disminución en las funcionalidades o una total incompatibilidad. Finalmente, se seleccionaron instancias completamente disjuntas.

Luego de seleccionar las instancias de los periféricos, se configuraron para realizar un *loopback*. La configuración se realizó de la siguiente manera:

- *CAN*: se utilizó el MCAN1 con una configuración de *loopback* interna, como se puede ver en la figura 3.2.
- *PIO*: se configuraron dos terminales del dispositivo bajo prueba. El primero como salida sin *latch* y el segundo como entrada sin circuito anti rebote.
- *SPI*: la configuración elegida fue por defecto ya que el *loopback* se logró conectando TX y RX con un cable.
- *UART*: se configuró el periférico con una velocidad de 9600 baudios, 8 bits de datos y sin bits de paridad.
- *Watchdog*: el disparo se configuró con un contador en 4095 cuentas. Este valor se estimó entre dos y cinco ejecuciones del *loop* principal.

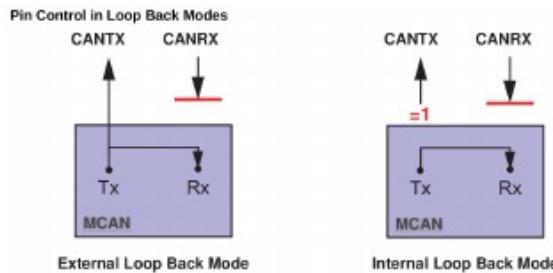


FIGURA 3.2. Diagrama de *loopback* del periférico CAN¹.

Como se puede ver en la figura 3.3, se priorizaron los *loopbacks* físicos externos. Cuando esta estrategia no fue posible, se optó por internos provistos por el fabricante. Finalmente, en los casos que las dos primeras opciones fueron imposibles, se utilizó una estrategia de *software*. En la tabla 3.1 se puede ver un resumen de las estrategias aplicadas.

TABLA 3.1. Comparación entre estrategias de depuración.

Periférico	Validación	Detección en un ciclo
CAN	Loopback interno	Sí
PIO	Loopback externo	No
SPI	Loopback externo	Sí
UART	Lógica en firmware	No
Watchdog	Lógica en inyector	No

¹Imagen tomada de la hoja de datos del dispositivo bajo prueba [12].

El siguiente paso en la etapa de diseño fue la selección de las instancias de los periféricos del integrado. Es posible que dos periféricos comparten parte del circuito interno o terminales del encapsulado. Esta situación puede generar una disminución en las funcionalidades o una total incompatibilidad. Finalmente, se seleccionaron instancias completamente disjuntas.

Luego de seleccionar las instancias de los periféricos, se configuraron para realizar un *loopback*. La configuración se realizó de la siguiente manera:

- *CAN*: se utilizó el MCAN1 con una configuración de *loopback* interna, como se puede ver en la figura 3.2.
- *PIO*: se configuraron dos terminales del dispositivo bajo prueba. El primero como salida sin *latch* y el segundo como entrada sin circuito anti rebote.
- *SPI*: la configuración elegida fue por defecto ya que el *loopback* se logró conectando TX y RX con un cable.
- *UART*: se configuró el periférico con una velocidad de 9600 baudios, 8 bits de datos y sin bits de paridad.
- *Watchdog*: el disparo se configuró con un contador en 4095 cuentas. Este valor se estimó entre dos y cinco ejecuciones del *loop* principal.

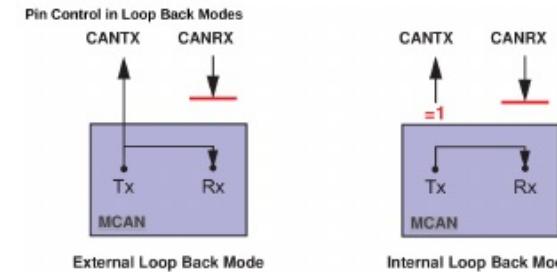


FIGURA 3.2. Diagrama de *loopback* del periférico CAN¹.

Como se puede ver en la figura 3.3, se priorizaron los *loopbacks* físicos externos. Cuando esta estrategia no fue posible, se optó por internos provistos por el fabricante. Finalmente, en los casos que las dos primeras opciones fueron imposibles, se utilizó una estrategia de *software*.

En la tabla 3.1 se puede ver un resumen de las estrategias aplicadas.

¹Imagen tomada de la hoja de datos del dispositivo bajo prueba [12].

Una vez configurados los componentes de *hardware* del dispositivo bajo prueba; se procedió a diseñar el *firmware*. Se comenzó con la estructura que define los reportes de estado del dispositivo bajo prueba. Los reportes están formados por 2 bytes, el primero es el carácter “F” y marca el inicio del reporte mientras que el segundo byte lleva la información del estado de los periféricos. En el código 3.1 se puede ver la implementación del segundo byte del reporte.

```

1 #define BIT 1
2
3 struct status_bitfield_t
4 {
5     uint8_t CAN:BIT;
6     uint8_t SPI:BIT;
7     uint8_t PIO:BIT;
8     uint8_t WATCHDOG:BIT;
9 } __attribute__((packed));
10
11 typedef union
12 {
13     struct status_bitfield_t status_of;
14     uint8_t packed;
15 } report_t;
```

CÓDIGO 3.1. Definición de la estructura de reportes.

En el código 3.2 se puede observar la implementación del lazo principal. Es importante notar que en la línea 10 se utilizó la *union* para transformar el reporte en caracteres legibles para una persona. En la figura 3.4 se puede observar el flujo completo del programa.

```

1 while ( true )
2 {
3     SYS_Tasks ( );
4     report.status_of.CAN = validate_CAN();
5     report.status_of.PIO = validate_PIO();
6     report.status_of.SPI = validate_SPI();
7     report.status_of.WATCHDOG = NORMAL;
8
9     buffer[FRAME_START] = 'F';
10    buffer[FLAGS_INDEX] = report.packed + 'A';
11    USART1_Write(&buffer[0], FRAME_SIZE);
12
13    WDT_Clear();
14 }
```

CÓDIGO 3.2. Lazo principal del *firmware* de autoevaluación.

TABLA 3.1. Comparación entre estrategias de depuración.

Periférico	Validación	Detección en un ciclo
CAN	Loopback interno	Sí
PIO	Loopback externo	No
SPI	Loopback externo	Sí
UART	Lógica en firmware	No
Watchdog	Lógica en inyector	No

Una vez configurados los componentes de *hardware* del dispositivo bajo prueba; se procedió a diseñar el *firmware*. Se comenzó con la estructura que define los reportes de estado del dispositivo bajo prueba. Los reportes están formados por 2 bytes, el primero es el carácter “F” y marca el inicio del reporte mientras que el segundo byte lleva la información del estado de los periféricos. En el código 3.1 se puede ver la implementación del segundo byte del reporte.

```

1 #define BIT 1
2
3 struct status_bitfield_t
4 {
5     uint8_t CAN:BIT;
6     uint8_t SPI:BIT;
7     uint8_t PIO:BIT;
8     uint8_t WATCHDOG:BIT;
9 } __attribute__((packed));
10
11 typedef union
12 {
13     struct status_bitfield_t status_of;
14     uint8_t packed;
15 } report_t;
```

CÓDIGO 3.1. Definición de la estructura de reportes.

En el código 3.2 se puede observar la implementación del lazo principal. Es importante notar que en la línea 10 se utilizó la *union* para transformar el reporte en caracteres legibles para una persona. En la figura 3.4 se puede observar el flujo completo del programa.

```

1 while ( true )
2 {
3     SYS_Tasks ( );
4     report.status_of.CAN = validate_CAN();
5     report.status_of.PIO = validate_PIO();
6     report.status_of.SPI = validate_SPI();
7     report.status_of.WATCHDOG = NORMAL;
8
9     buffer[FRAME_START] = 'F';
10    buffer[FLAGS_INDEX] = report.packed + 'A';
11    USART1_Write(&buffer[0], FRAME_SIZE);
12
13    WDT_Clear();
14 }
```

CÓDIGO 3.2. Lazo principal del *firmware* de autoevaluación.

3.1. Autoevaluación del dispositivo bajo prueba

21

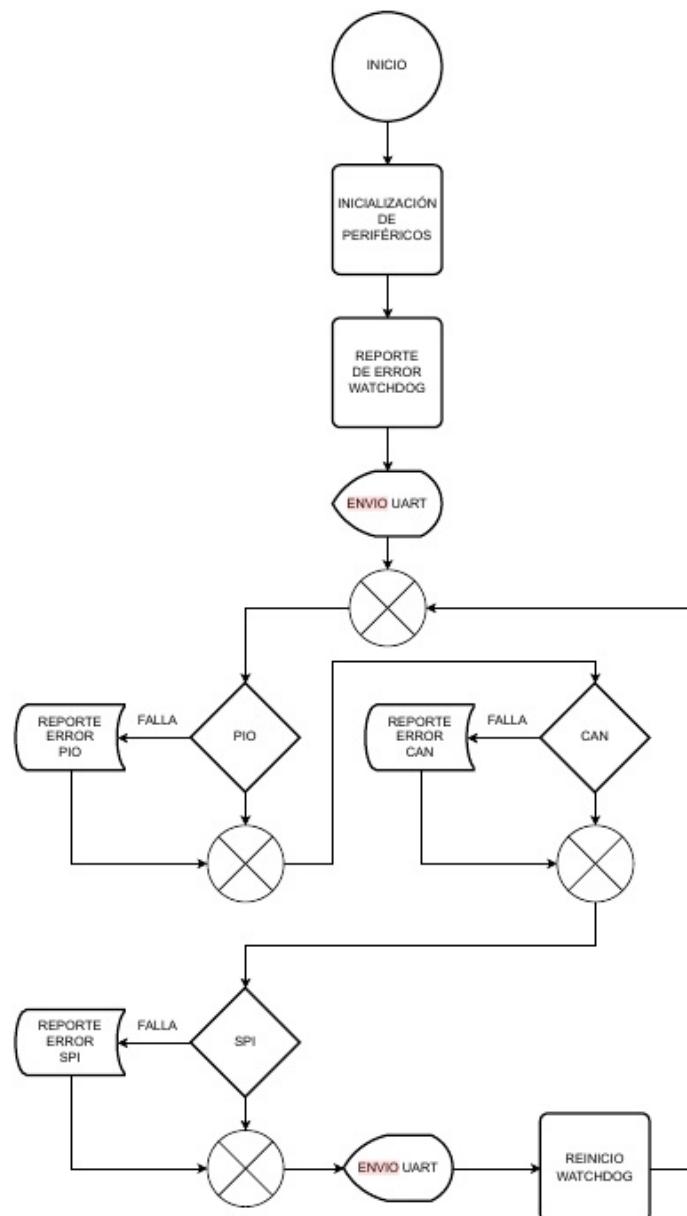


FIGURA 3.4. Flujo del firmware de autoevaluación.

3.1. Autoevaluación del dispositivo bajo prueba

21

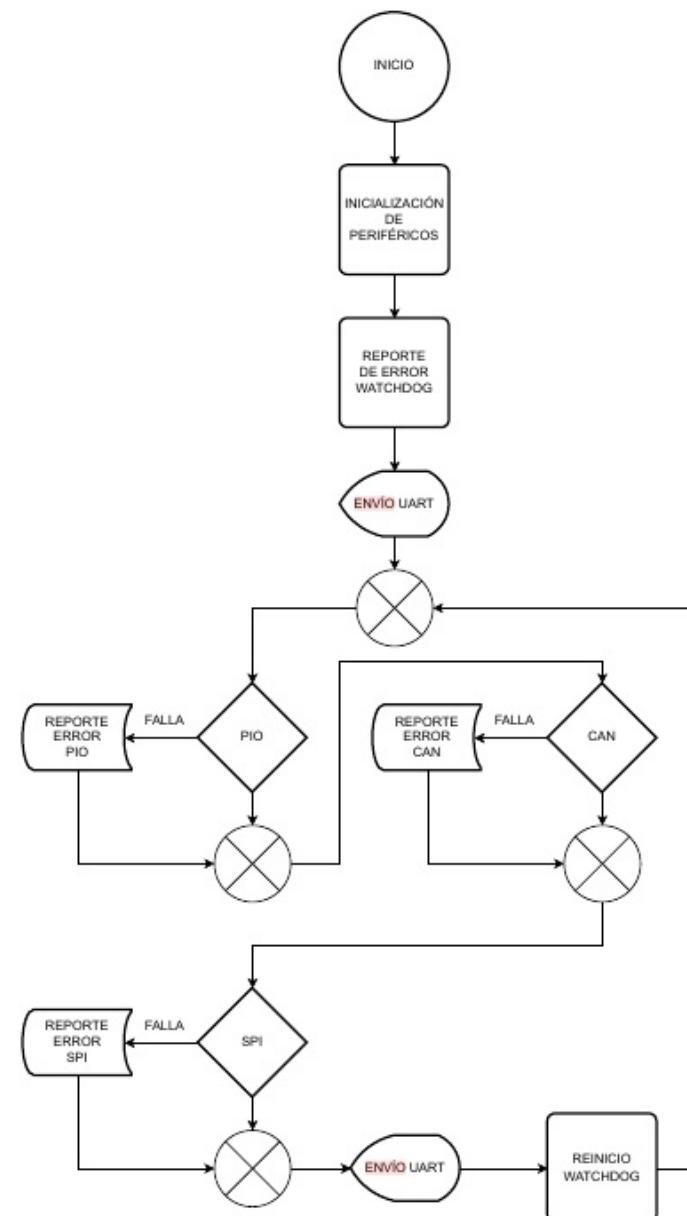


FIGURA 3.4. Flujo del firmware de autoevaluación.

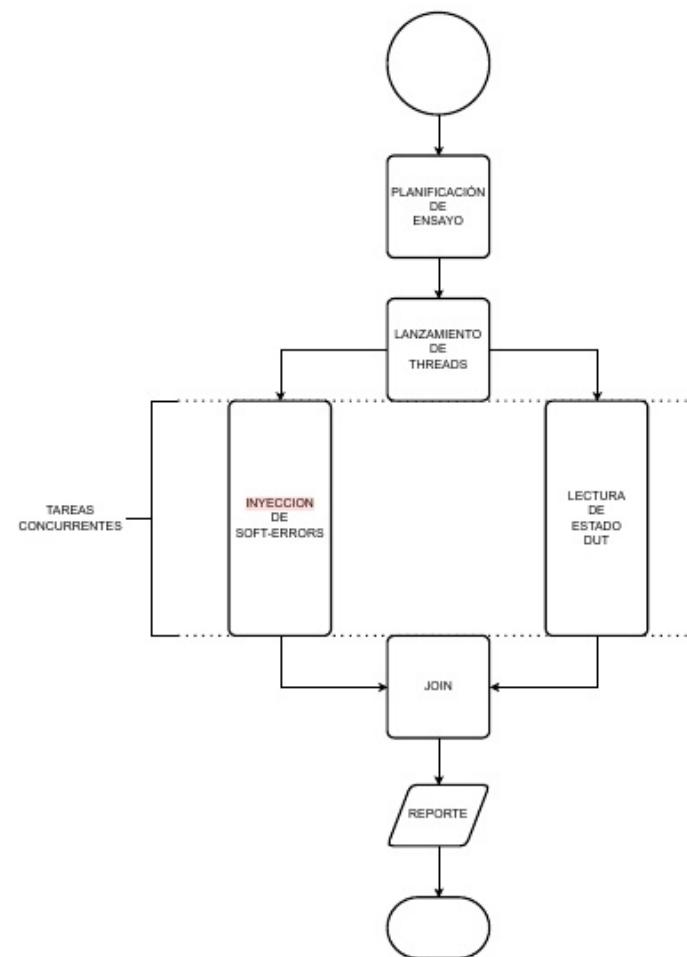


FIGURA 3.7. Flujo de tareas concurrentes.

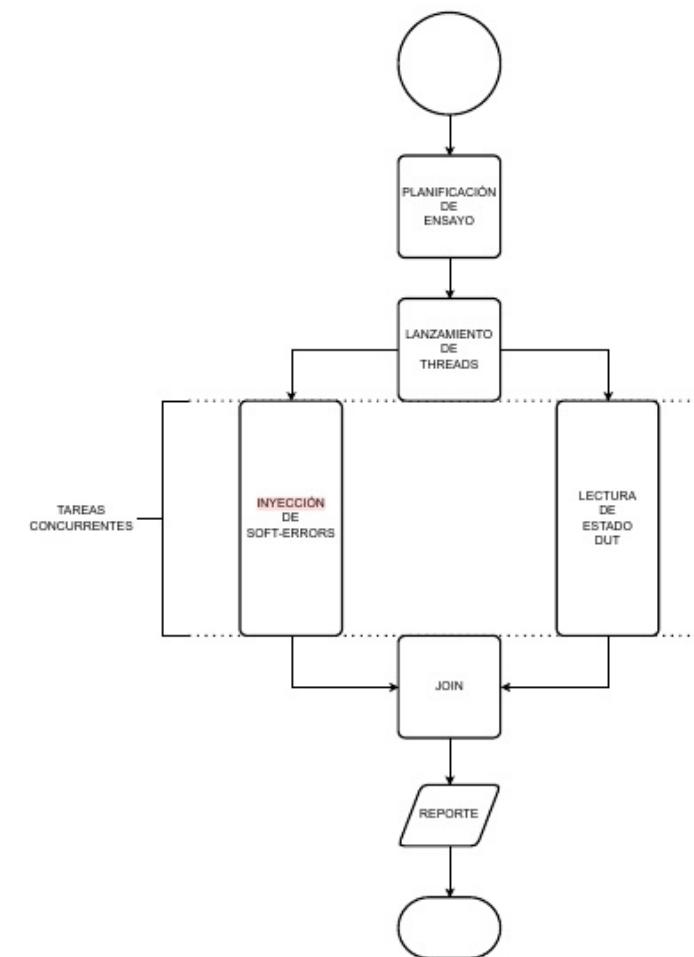


FIGURA 3.7. Flujo de tareas concurrentes.

Capítulo 4

Ensayos y resultados

En este capítulo se describe la estrategia de pruebas adoptada para determinar que el sistema se comporta de forma esperada.

4.1. Laboratorio remoto

Durante las primeras etapas del desarrollo no se disponía en Buenos Aires del dispositivo bajo prueba. Por esta razón, se montó un laboratorio remoto en San Carlos de Bariloche. Se dispuso una placa de evaluación *SAM V71 Xplained Ultra* conectada a un ordenador dentro de la red de INVAP S.E. La conexión entre la placa y el ordenador se logró a través de una sonda de depuración *Segger J-32*.

Para poder acceder al laboratorio remoto que se muestra en la figura 4.1 se necesitó:

- Credenciales de acceso y conexión a la VPN de INVAP S.E.
- Crear un túnel SSH con el ordenador remoto.

El túnel SSH se generó con *X11 forwarding* habilitado. De esta manera, se pudo generar ventanas gráficas en el ambiente local. Además, las operaciones de consola se integraron al ordenador personal con una sesión de *Tmux*. Finalmente, se logró implementar una interfaz de control del laboratorio remoto con una apariencia idéntica al ambiente local.

Para poder instalar las dependencias y los servidores OCD evaluados, se habilitaron los puertos necesarios que permitieron al ordenador remoto conectarse a los recursos en la Internet. Sin embargo, algunos recursos debieron ser compilados en el *host* y la transferencia de los *tarballs* se realizó por medio de *Secure Copy Files (scp)*.

Con el laboratorio remoto montado, se procedió a realizar las siguientes pruebas:

- Pruebas de configuración de sondas de depuración y compatibilidad con servidores OCD.
- Pruebas de acceso al dispositivo bajo prueba.

Las pruebas referidas a la sonda de depuración arrojaron como resultado lo siguiente:

- El modo de *boot* de la sonda determina el nivel de acceso al dispositivo bajo prueba.

Capítulo 4

Ensayos y resultados

En este capítulo se describe la estrategia de pruebas adoptada para determinar que el sistema se comporta de forma esperada.

4.1. Laboratorio remoto

Durante las primeras etapas del desarrollo no se disponía en Buenos Aires del dispositivo bajo prueba. Por esta razón, se montó un laboratorio remoto en San Carlos de Bariloche. Se dispuso una placa de evaluación *SAM V71 Xplained Ultra* conectada a un ordenador dentro de la red de INVAP S.E. La conexión entre la placa y el ordenador se logró a través de una sonda de depuración *Segger J-32*.

Para poder acceder al laboratorio remoto que se muestra en la figura 4.1 se necesitó:

- Credenciales de acceso y conexión a la VPN de INVAP S.E.
- Crear un túnel SSH con el ordenador remoto.

El túnel SSH se generó con *X11 forwarding* habilitado. De esta manera, se pudo generar ventanas gráficas en el ambiente local. Además, las operaciones de consola se integraron al ordenador personal con una sesión de *Tmux*. Finalmente, se logró implementar una interfaz de control del laboratorio remoto con una apariencia idéntica al ambiente local.

Para poder instalar las dependencias y los servidores OCD evaluados, se habilitaron los puertos necesarios que permitieron al ordenador remoto conectarse a los recursos en la Internet. Sin embargo, algunos recursos debieron ser compilados en el *host* y la transferencia de los *tarballs* se realizó por medio de *Secure Copy Files (SCP)*.

Con el laboratorio remoto montado, se procedió a realizar las siguientes pruebas:

- Pruebas de configuración de sondas de depuración y compatibilidad con servidores OCD.
- Pruebas de acceso al dispositivo bajo prueba.

Las pruebas referidas a la sonda de depuración arrojaron como resultado lo siguiente:

- El modo de *boot* de la sonda determina el nivel de acceso al dispositivo bajo prueba.

Capítulo 5

Conclusiones

Este capítulo explica de forma breve el cierre del trabajo realizado, sus logros y futuro.

5.1. Logros obtenidos

El primer logro a destacar es que se pudo mejorar el estado del arte actual. Su mejora es la capacidad de describir una simulación en términos de la radiación cósmica. Esto se logró al construir una capa de abstracción de aplicaciones. Esta, esconde la gestión de la sesión de depuración. De esta manera, se superaron las técnicas utilizadas en la actualidad. Luego, el desarrollador solo debe describir los errores a introducir en el dispositivo bajo prueba. Finalmente, se agrega valor al mejorar el tiempo, costo y claridad de los ensayos.

Un segundo logro a destacar es la recepción que obtuvo este trabajo. La cual fue muy positiva y con un impacto mayor al esperado. Ya que en la actualidad, INVAP S.E. se encuentra en proceso de integrar la herramienta en su ambiente de desarrollo de satélites. Además, el sistema realizado se utiliza dentro del marco de un proyecto final en la especialización en sistemas embebidos.

Como tercer logro a remarcar es que el trabajo pudo cumplir las expectativas y requerimientos del cliente. Además, se cumplieron sin desvíos respecto a la planificación de tiempo y recursos. Esto fue posible gracias los contenidos de gestión de proyectos que se impartieron durante el desarrollo de esta maestría. Finalmente, estos son los objetivos más destacados:

- Creación de un sistema de inyección de errores transitorios que permita evaluar técnicas de mitigación de errores.
- Acceso a los componentes de interés del dispositivo bajo prueba.
- Biblioteca para el diseño de ensayos de radiación cósmica en lenguaje Python 3.
- *Firmware* de autoevaluación del dispositivo bajo prueba que verifique los periféricos de interés.

Capítulo 5

Conclusiones

Este capítulo explica de forma breve el cierre del trabajo realizado, sus logros y trabajo a futuro.

5.1. Logros obtenidos

El primer logro a destacar es que se pudo mejorar el estado del arte actual. Su mejora es la capacidad de describir una simulación en términos de la radiación cósmica. Esto se logró al construir una capa de abstracción de aplicaciones. Esta, esconde la gestión de la sesión de depuración. De esta manera, se superaron las técnicas utilizadas en la actualidad. Luego, el desarrollador solo debe describir los errores a introducir en el dispositivo bajo prueba. Finalmente, se agrega valor al mejorar el tiempo, costo y claridad de los ensayos.

Un segundo logro a destacar es la recepción que obtuvo este trabajo. Esta fue muy positiva y con un impacto mayor al esperado. Ya que en la actualidad, INVAP S.E. se encuentra en proceso de integrar la herramienta en su ambiente de desarrollo de satélites. Además, el sistema realizado se utiliza dentro del marco de un proyecto final en la Carrera de Especialización en Sistemas Embebidos.

Como tercer logro a remarcar es que el trabajo pudo cumplir las expectativas y requerimientos del cliente. Además, se cumplieron sin desvíos respecto a la planificación de tiempo y recursos. Esto fue posible gracias los contenidos de gestión de proyectos que se impartieron durante el desarrollo de esta maestría. Finalmente, estos son los objetivos más destacados:

- Creación de un sistema de inyección de errores transitorios que permita evaluar técnicas de mitigación de errores.
- Acceso a los componentes de interés del dispositivo bajo prueba.
- Biblioteca para el diseño de ensayos de radiación cósmica en lenguaje Python 3.
- *Firmware* de autoevaluación del dispositivo bajo prueba que verifique los periféricos de interés.

5.2. Trabajo futuro

La investigación realizada durante la producción del trabajo sugiere que es posible agregar las siguientes funcionalidades:

- Conexión entre el código fuente del dispositivo bajo prueba y el inyector de errores transitorios. Esto se lograría a través del uso de los símbolos de depuración generados en el binario. Con estos símbolos es posible unir un valor del registro *program counter* con una línea en el código fuente. Esta funcionalidad permitiría analizar la vulnerabilidad de un segmento de código escrito en lenguaje C/C++.
- Creación de instrucciones específicas para la inyección de *single event functional interrupt*. Estas instrucciones lograrían alterar los registros que definen la configuración de un periférico. De esta manera, se podrían realizar pruebas sobre las técnicas de mitigación de este tipo de errores. La dificultad a superar es que los registros de los periféricos son propios del dispositivo en particular. Para lograr una abstracción genérica se requiere definir el comportamiento de las funciones en tiempo de ejecución.

Si se lograran introducir estas funcionalidades al trabajo realizado, el método de simulación de errores transitorio por *software* quedaría obsoleto. Esto tendría el efecto de modificar el estado del arte; ya que se evitaría el problema que tiene el método al intentar introducir un error cuando el integrado no se encuentra en modo privilegiado.

Como se mencionó en la sección 5.1, este trabajo forma parte de un proyecto final de la **especialización en sistemas embebidos**. Probablemente se **obtengan sugerencias** y nuevas ideas durante su desarrollo. Finalmente, se seguirá con interés la evolución del trabajo en curso.

5.2. Trabajo futuro

La investigación realizada durante la producción del trabajo sugiere que es posible agregar las siguientes funcionalidades:

- Conexión entre el código fuente del dispositivo bajo prueba y el inyector de errores transitorios. Esto se lograría a través del uso de los símbolos de depuración generados en el binario. Con estos símbolos es posible unir un valor del registro *program counter* con una línea en el código fuente. Esta funcionalidad permitiría analizar la vulnerabilidad de un segmento de código escrito en lenguaje C/C++.
- Creación de instrucciones específicas para la inyección de *single event functional interrupt*. Estas instrucciones lograrían alterar los registros que definen la configuración de un periférico. De esta manera, se podrían realizar pruebas sobre las técnicas de mitigación de este tipo de errores. La dificultad a superar es que los registros de los periféricos son propios del dispositivo en particular. Para lograr una abstracción genérica se requiere definir el comportamiento de las funciones en tiempo de ejecución.

Si se lograran introducir estas funcionalidades al trabajo realizado, el método de simulación de errores transitorio por *software* quedaría obsoleto. Esto tendría el efecto de modificar el estado del arte; ya que se evitaría el problema que tiene el método al intentar introducir un error cuando el integrado no se encuentra en modo privilegiado.

Como se mencionó en la sección 5.1, este trabajo forma parte de un proyecto final de la **Carrera de Especialización en Sistemas Embebidos**. Probablemente se **obtengan sugerencias** y nuevas ideas durante su desarrollo. Finalmente, se seguirá con interés la evolución del trabajo en curso.