



# MAESTRÍA EN INTERNET DE LAS COSAS

## MEMORIA DEL TRABAJO FINAL

### **Evaluador de microcontroladores para misiones espaciales**

#### **Autor:**

**Esp. Ing. Gonzalo Nahuel Vaca**

#### Director:

Ing. Roberto Cibils (INVAP S.E.)

#### Codirector:

Ing. Damian Rosetani (INVAP S.E.)

#### Jurados:

Mg. Ing. Iván Andrés León Vásquez (INVAP S.E.)

Mg. Ing. Rodrigo Cardenas (FIUBA)

Esp. Ing. Pablo Almada (FIUBA-UTN)

*Este trabajo fue realizado en la Ciudad Autónoma de Buenos Aires,  
entre mayo de 2021 y junio de 2022.*



## *Resumen*

Esta memoria explica el trabajo realizado para INVAP S.E. en el área de la tecnología aeroespacial. Se realizó una herramienta que simula los efectos de la radiación cósmica en un microcontrolador.

La herramienta sirve para abaratar el costo de la producción de satélites y aumentar su confiabilidad. Las simulaciones permiten evaluar técnicas de mitigación de errores y componentes no calificados para uso espacial. Para realizar este trabajo se valió de la teoría de arquitecturas de microcontroladores y sus protocolos de depuración.



# Índice general

<b>Resumen</b>	<b>I</b>
<b>1. Introducción general</b>	<b>1</b>
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
<b>2. Introducción específica</b>	<b>9</b>
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
<b>3. Diseño e implementación</b>	<b>17</b>
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
<b>4. Ensayos y resultados</b>	<b>31</b>
4.1. Laboratorio remoto . . . . .	31
4.2. Ensayos de inyector . . . . .	31
4.3. Validación con el cliente . . . . .	31
<b>5. Conclusiones</b>	<b>33</b>
5.1. Logros obtenidos . . . . .	33
5.2. Trabajo futuro . . . . .	34
<b>Bibliografía</b>	<b>35</b>



# Índice de figuras

1.1. Satélite SAOCOM <sup>1</sup> . . . . .	2
1.2. Capas magnéticas de la tierra y viento solar <sup>2</sup> . . . . .	3
1.3. Ejemplo simplificado de <i>bit flip</i> en un bloque <i>SDRAM</i> <sup>3</sup> . . . . .	3
1.4. Proyección de la constelación <i>Starlink</i> <sup>4</sup> . . . . .	5
1.5. Cámara de pruebas de iones pesados <sup>5</sup> . . . . .	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> <sup>6</sup> . . . . .	10
2.2. Diagrama del módulo <i>CoreSight</i> <sup>7</sup> . . . . .	11
2.3. Conexión de una sesión de depuración. . . . .	12
2.4. Sonda de depuración <i>Segger J-32</i> <sup>8</sup> . . . . .	13
2.5. Ejemplo del flujo de trabajo <i>Tmux-Neovim</i> . . . . .	15
3.1. Diagrama de configuración de las señales de reloj. . . . .	17
3.2. Diagrama de <i>loopback</i> del periférico <i>CAN</i> <sup>9</sup> . . . . .	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 <i>soft-errors</i> . . . . .	27
3.7. Flujo de tareas concurrentes. . . . .	28
4.1. Diagrama en bloques del laboratorio remoto. . . . .	31
4.2. Dispositivo alternativo <i>NUCLEO-F429ZI</i> . . . . .	32





# 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 . . . . .	11
2.2. Resumen de periféricos . . . . .	14
3.1. Estrategias de depuración . . . . .	18
3.2. Funcionalidades abstraídas . . . . .	23
4.2. Resumen de ensayos . . . . .	32
4.3. Resumen de la validación con el cliente . . . . .	32



*Dedicado a mi hija Helena*



# Capítulo 1

## Introducción general

Este capítulo presenta de forma breve el contexto del trabajo realizado. Se explica la problemática solucionada y los conceptos necesarios para la lectura de la memoria.

### 1.1. El espacio como recurso estratégico

La explotación del espacio exterior permite mejorar la producción de bienes y servicios. Además, provee datos de valor científico que incrementan la calidad de vida de las personas [1]. Finalmente, es una fuente de desafíos que impulsa el desarrollo tecnológico.

El marco jurídico que regula la explotación del recurso es el tratado sobre el espacio exterior de 1967. Del cual, la República Argentina es miembro desde el 27 de enero de ese año. Los países pueden reclamar cualquiera de las órbitas disponibles pero tienen la obligación de hacer uso dentro de un margen de tiempo determinado. Luego, si el país no ocupó la órbita pierde el derecho a usarla [2].

El mercado de la explotación espacial está compuesto de estados y empresas. Siendo las últimas quienes adquirieron el protagonismo en el siglo XXI [3]. El marco jurídico y la competencia en este rubro empujan a las compañías a innovar de forma permanente. En particular, con nuevas técnicas y tecnologías que permitan abaratar el costo de las misiones.

El cliente de este trabajo es INVAP S.E. y es una empresa de la provincia de Río Negro. La compañía es modelo en su tipo y realiza proyectos tecnológicos complejos en áreas como: reactores nucleares, satélites y radares. En la figura 1.1 se puede observar un satélite realizado por la empresa.

FIGURA 1.1. Satélite SAOCOM<sup>1</sup>.

## 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.

En la figura 1.3 se puede ver el efecto transitorio de la radiación sobre los transistores de un integrado. En particular, la inyección de un error denominado *bit*

<sup>1</sup>Imagen tomada de la página oficial de INVAP S.E. [4]

*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 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.

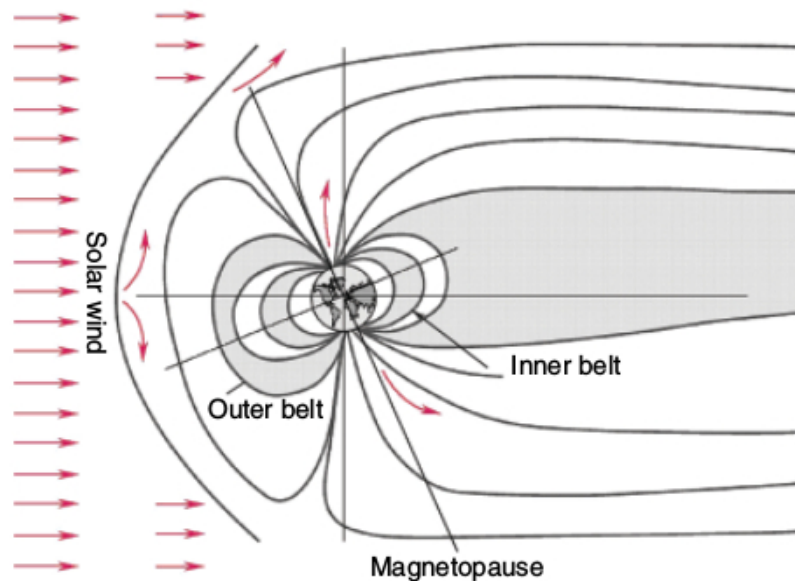


FIGURA 1.2. Capas magnéticas de la tierra y viento solar<sup>2</sup>.

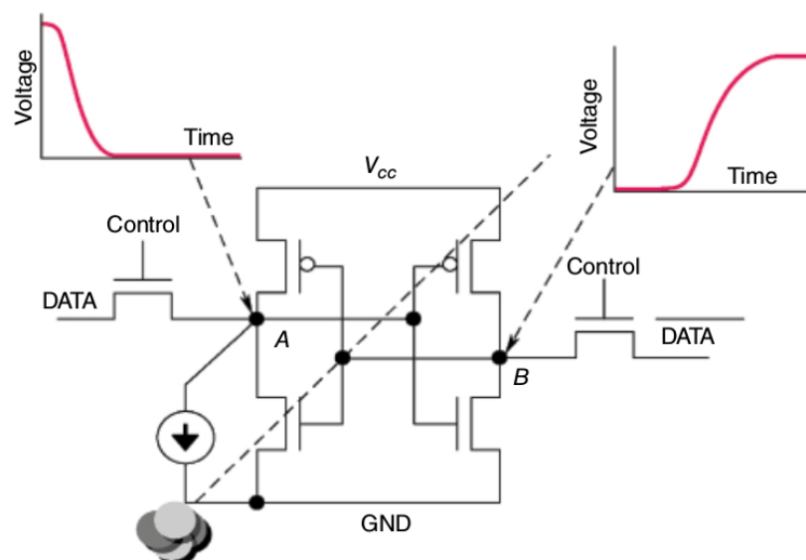


FIGURA 1.3. Ejemplo simplificado de *bit flip* en un bloque SDRAM<sup>3</sup>.

<sup>2</sup>Imagen tomada del artículo *Structure of space radiation* [5]

<sup>3</sup>Imagen tomada del artículo *Effects of space radiation on electronic devices* [7]

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 [8].

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.

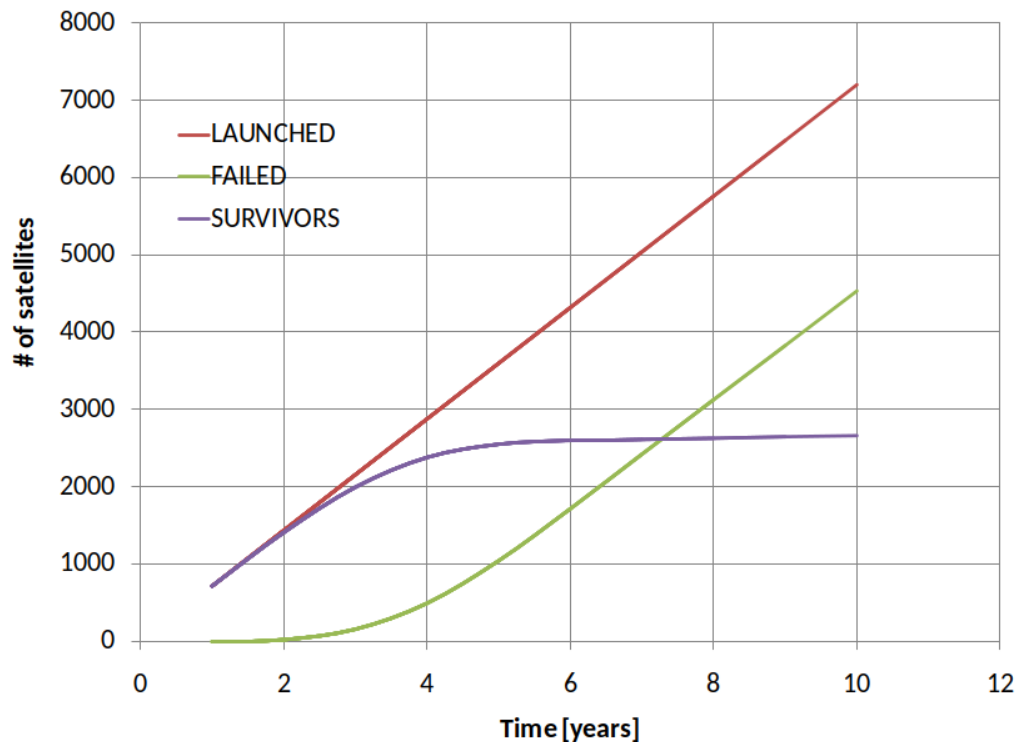
El caso de *Starlink* es un ejemplo de *new space* particular. Su volumen de satélites lanzados permite realizar conclusiones estadísticas significativas. En particular, su poca capacidad para cumplir sus objetivos si se mantiene la actual tasa de mortalidad de sus satélites [9]. En la figura 1.4 se 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* [9].

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



FIGURA 1.4. Proyección de la constelación *Starlink*<sup>4</sup>.

Las conclusiones del caso *Starlink* muestran la importancia de tener herramientas para simular el ambiente espacial. En particular, los efectos de la radiación cósmica para poder probar las técnicas de mitigación de errores seleccionadas. Finalmente, este trabajo agrega valor al cliente al incrementar la confiabilidad de los satélites y evitar los problemas de la competencia.

## 1.4. Estado del arte

La microelectrónica se encuentra en un proceso constante de cambio. Se incrementa la densidad de integración, la velocidad de los dispositivos y se reducen los niveles de tensión eléctrica. Este progreso hace que los circuitos sean más vulnerables a la radiación cósmica.

El uso de dispositivos sin calificación espacial hace que los riesgos frente a un error transitorio sean mayores. Además, la aplicación final de vuelo no suele estar disponible durante la fase de desarrollo de los proyectos. Esto dificulta aún más la mitigación de errores y aumenta la vulnerabilidad de los sistemas.

La manera tradicional de evaluar las técnicas de mitigación de errores es realizar un ensayo por radiación. En estos ensayos se utilizan instalaciones en tierra para pruebas de radiación. Sin embargo, estas instalaciones son costosas y la preparación y ejecución de los ensayos consumen mucho tiempo.

Para lograr un ensayo por radiación, las instalaciones deben producir un rayo de partículas cargadas. Estas partículas se pueden obtener de:

<sup>4</sup>Imagen tomada de la publicación de Roberto Cibils [9].

- Aceleradores de partículas: en esta categoría se incluyen ciclotrones y aceleradores lineales.
- Decaimiento por fisión: basados en el decaimiento por fisión espontánea de elementos como  $Cf^{252}$ .

En la figura 1.5 se puede observar una cámara de iones pesados. Durante el ensayo se ejecuta una metodología particular que define la actividad en el dispositivo bajo prueba. Además, se necesita de un sistema que controle y observe el dispositivo bajo prueba durante su exposición a la radiación. Finalmente, se requiere de personal calificado en el diseño, ejecución e interpretación de estos ensayos.

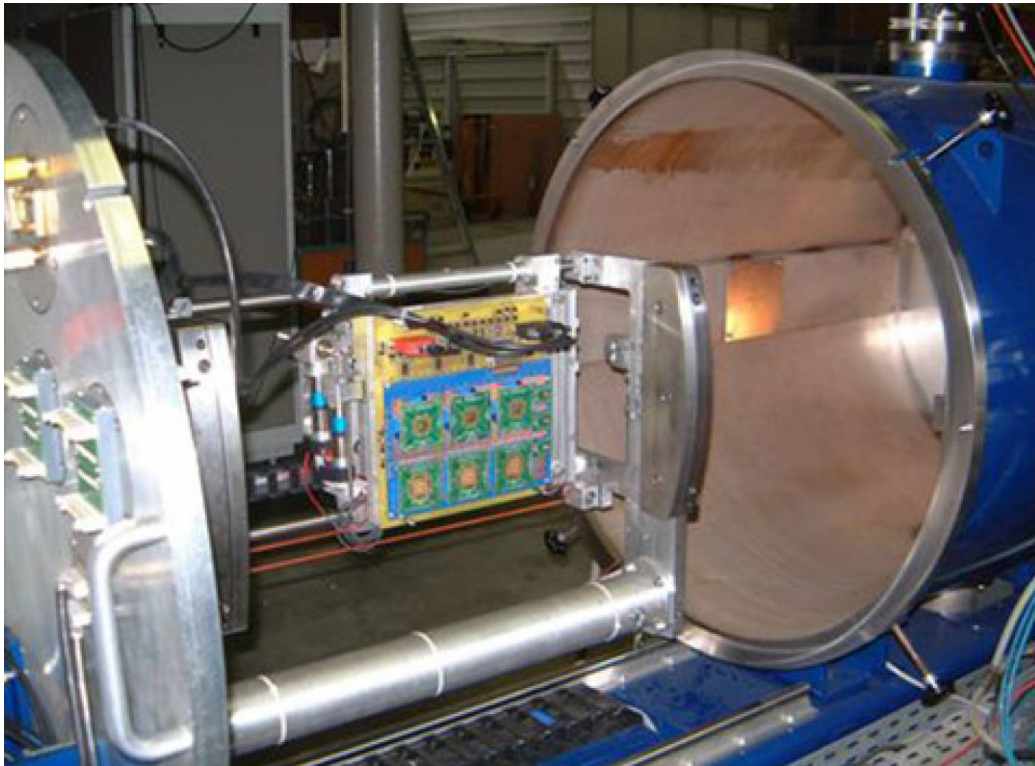


FIGURA 1.5. Cámara de pruebas de iones pesados<sup>5</sup>.

Las estrategias para la inyección de errores pueden ser estáticas o dinámicas. En las estáticas solo se observa si se producen cambios en valores determinados de memorias y registros. Las pruebas dinámicas se realizan mientras se activan secuencias de lectura y escritura de memorias. También incluyen la ejecución de programas para estimular el procesador.

El diseño de un ensayo dinámico necesita determinar la contribución de *SEU* sobre una sección transversal de memoria. La cual está relacionada con su tiempo de lectura y escritura (ciclo de trabajo). La sección transversal de un programa se puede definir entonces como:

$$\sigma_{(SEU)} = \sum d(R_i) \times \sigma_{R_i} \quad (1.1)$$

<sup>5</sup>Imagen tomada del sitio web ucl.ac.uk [10].

Donde:

- $d(R_i)$  es el ciclo de trabajo del elemento de memoria  $R$ .
- $\sigma_{R_i}$  es la sección transversal 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 que 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  $\tau_{inj}$  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
<i>Software</i>	Baja	Bajo	Ciclos de CPU
<i>Hardware</i>	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.

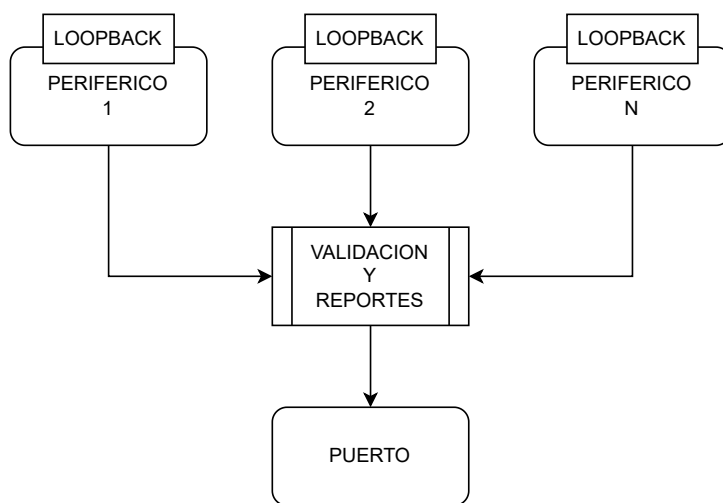


FIGURA 1.6. Diagrama simplificado del dispositivo bajo prueba.

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.

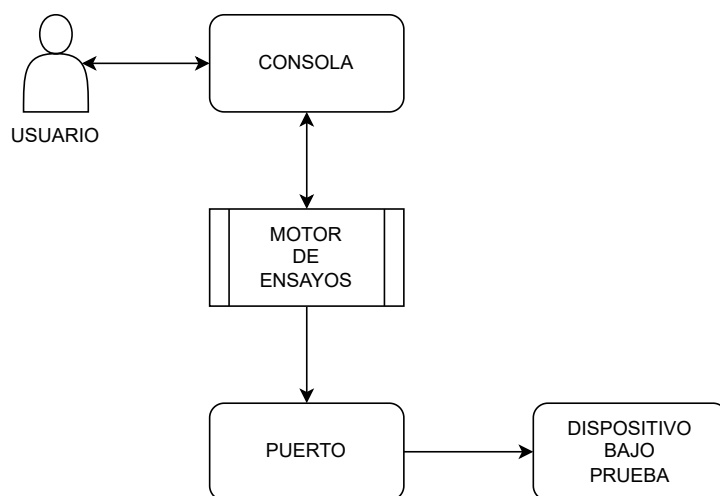


FIGURA 1.7. Diagrama simplificado del sistema de inyección de errores.

## 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 [11]. 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) [12]. 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  $60,0 \text{ MeV.cm}^2/\text{mg}$ . 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*.

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*.

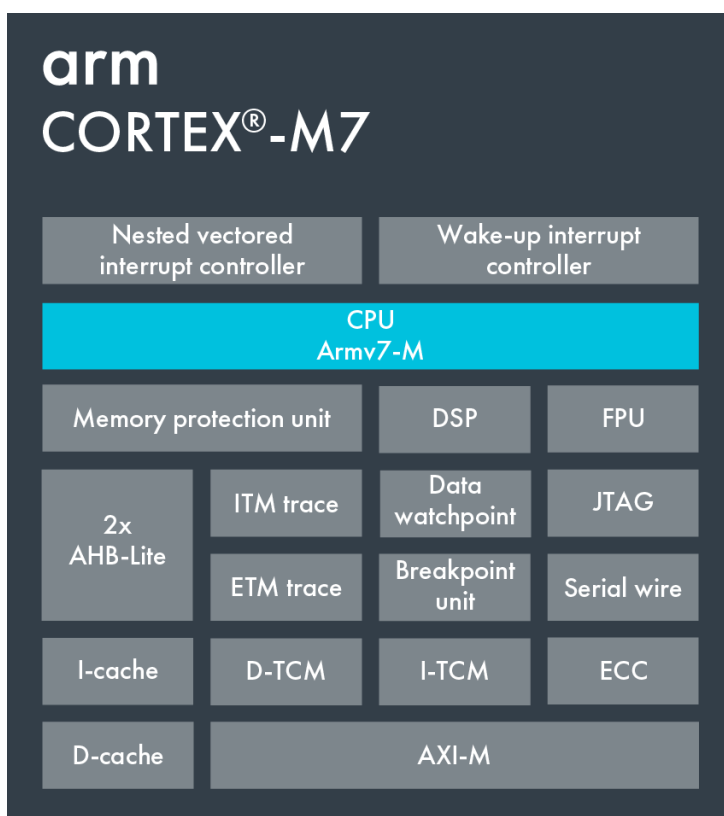


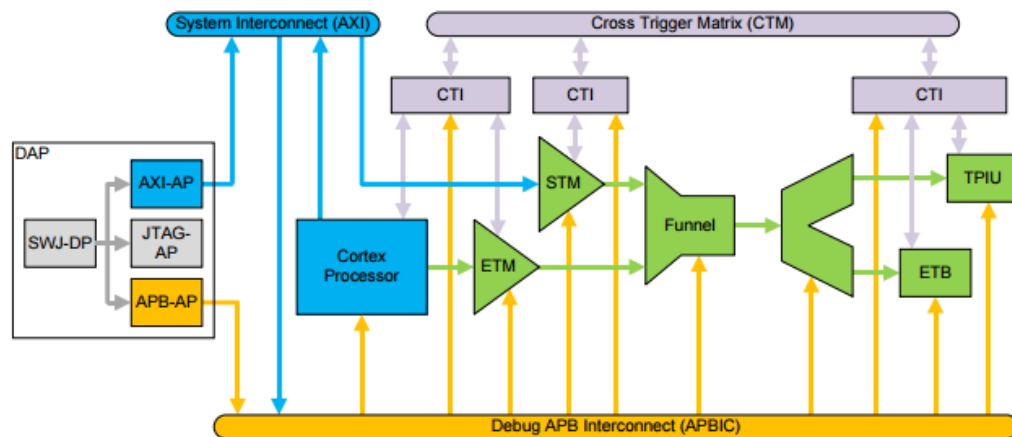
FIGURA 2.1. Diagrama de la arquitectura *Cortex M7*<sup>1</sup>.

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*.
- *Embedded Trace Macrocells*: permite extraer información y controlar el núcleo del dispositivo.

<sup>1</sup>Imagen tomada de la página oficial de *ARM Developers*. [13]

- *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 CoreSight<sup>2</sup>.

## 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 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) [15]. En la tabla 2.1 se puede observar un resumen de los servidores evaluados en el trabajo.

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

<sup>2</sup>Imagen tomada del artículo *How to debug: CoreSight basis*. [14]

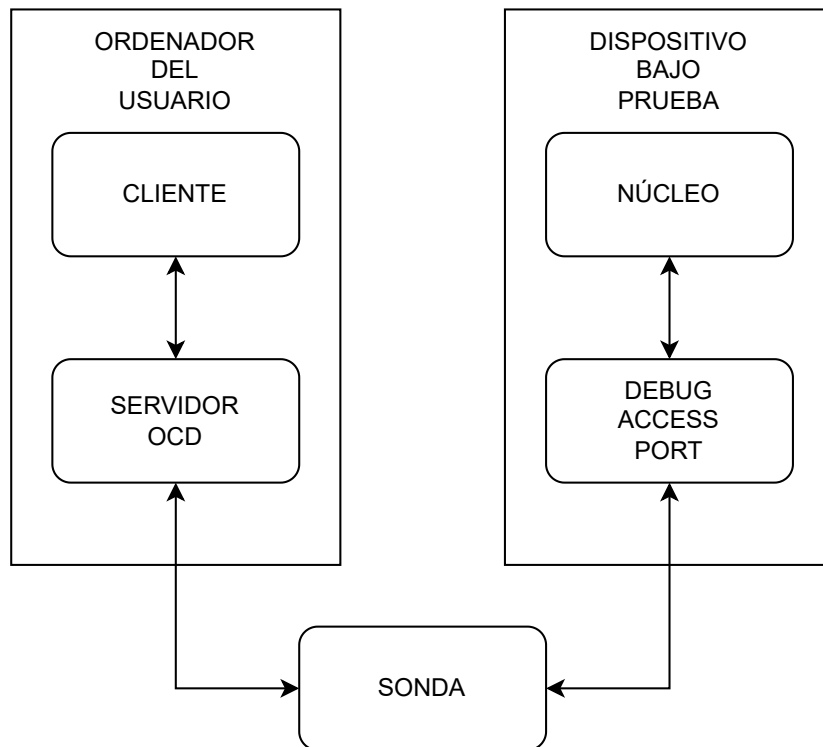


FIGURA 2.3. Conexión de una sesión de depuración.

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 [16]. 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
  - 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-32<sup>3</sup>.

## 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 microprocesador.

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

---

<sup>3</sup>Imagen 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.

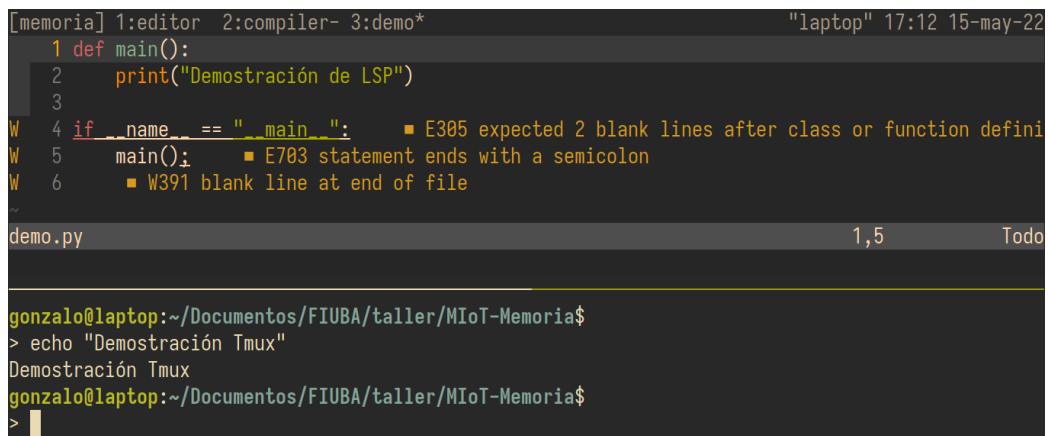
Para escribir el código en Python 3, el *firmware* en C y esta memoria en  $\text{\LaTeX}$ , se utilizó el editor de texto Neovim. Este programa está basado en el editor Vi de los sistemas Unix. Su funcionamiento es modal, esto significa que el editor funciona en los siguientes modos:

- 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.

- 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
W 4 if __name__ == "__main__":      ■ E305 expected 2 blank lines after class or function defini
W 5     main():                      ■ E703 statement ends with a semicolon
W 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.

## 2.5. Requerimientos del cliente

Se realizaron una serie de reuniones con el cliente y se pudo definir los requerimientos del trabajo. A continuación se enumeran los principales:

1. Referentes al inyector por consola de comandos:
  - a) Generará de una interfaz de usuario.
  - b) Permitirá configurar el ensayo a realizar.
  - c) Observará la salida del dispositivo bajo prueba.
  - d) Inyectará *soft-errors* en el dispositivo bajo prueba.
  - e) Persistirá las operaciones, entradas y salidas.
  - f) Generará informes del ensayo realizado.
2. Referentes al proceso del dispositivo bajo prueba:
  - a) Verificará el estado de los periféricos del dispositivo bajo prueba.
  - b) Detectará si el dispositivo bajo prueba perdió su secuencia.
  - c) Generará reportes de estado de periféricos y secuencia.
  - d) Permitirá que el inyector por consola de comandos configure el alcance de la secuencia.
  - e) Permitirá que el inyector por consola de comandos maneje el flujo de su secuencia.

El cliente definió algunas restricciones para el desarrollo del sistema. Estas se enumeran a continuación:

- Utilización de un repositorio con control de versiones *Gitlab*.
- Documentación del código con *Doxygen*.
- Utilización exclusiva del lenguaje de programación *Python 3*.

## 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 del dispositivo bajo prueba.
- Selección y configuración de los periféricos del dispositivo bajo prueba.
- Selección y configuración de los terminales externos del dispositivo bajo prueba.
- 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.

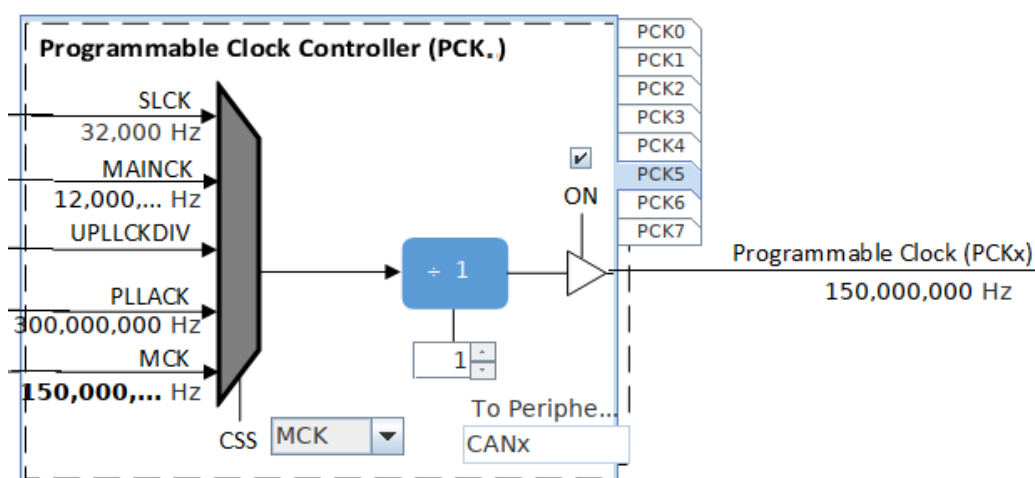


FIGURA 3.1. Diagrama de configuración de las señales de reloj.

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

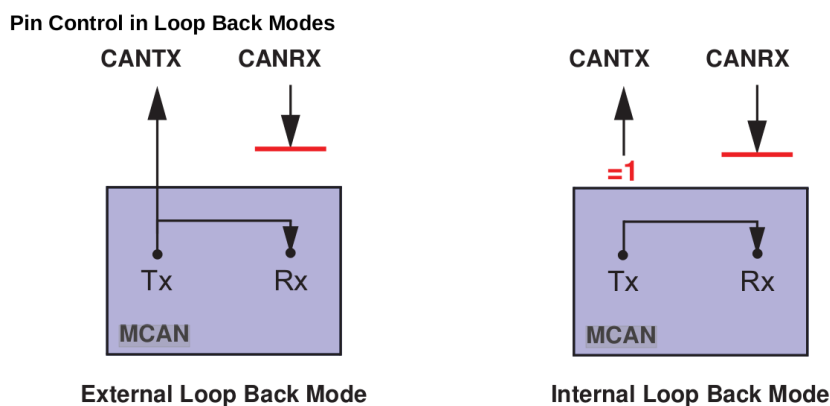


FIGURA 3.2. Diagrama de *loopback* del periférico CAN<sup>1</sup>.

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	Si
PIO	Loopback externo	No
SPI	Loopback externo	Si
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

<sup>1</sup>Imagen tomada de la hoja de datos del dispositivo bajo prueba [11].

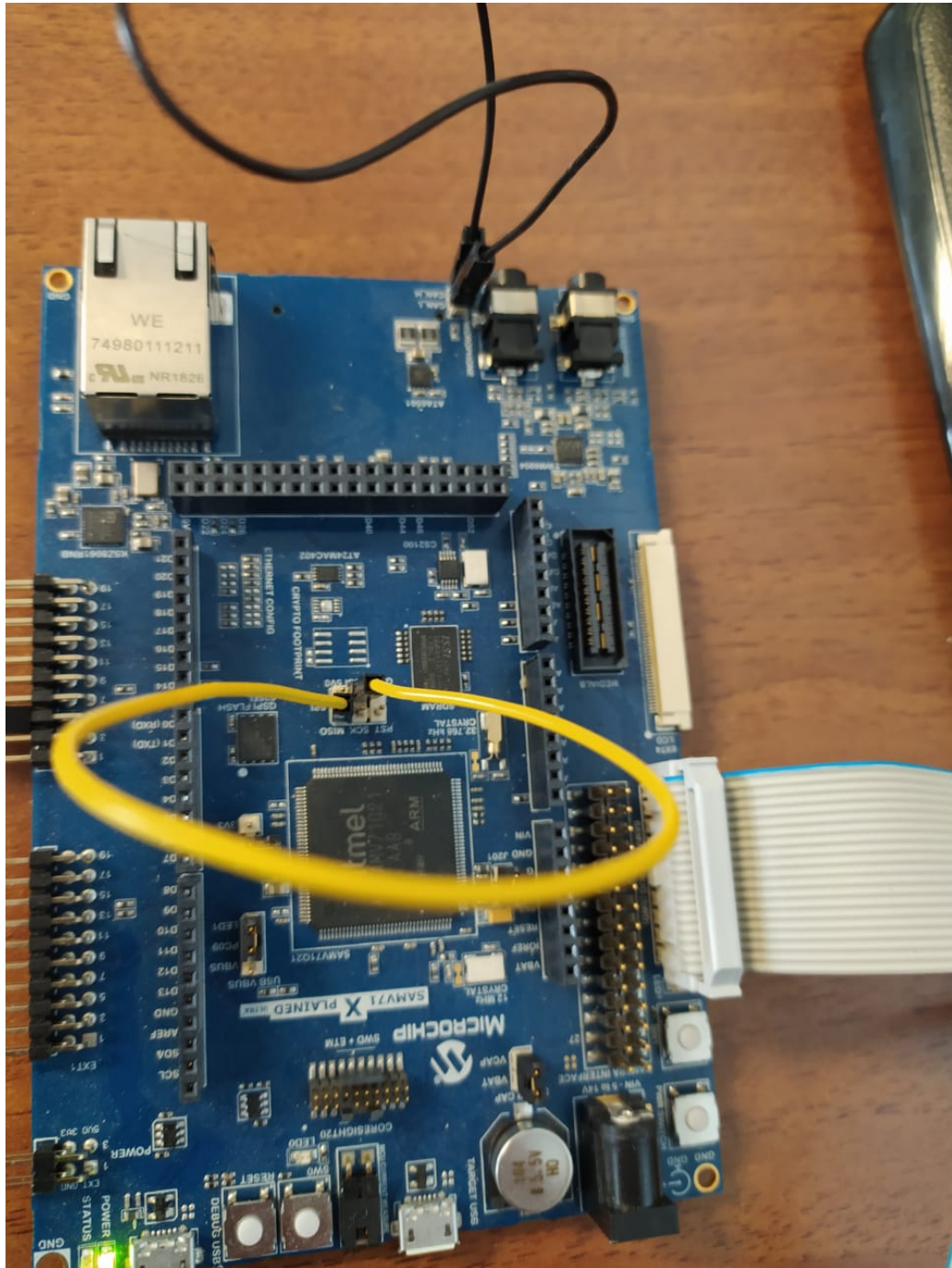


FIGURA 3.3. Fotografía del dispositivo bajo prueba.

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.

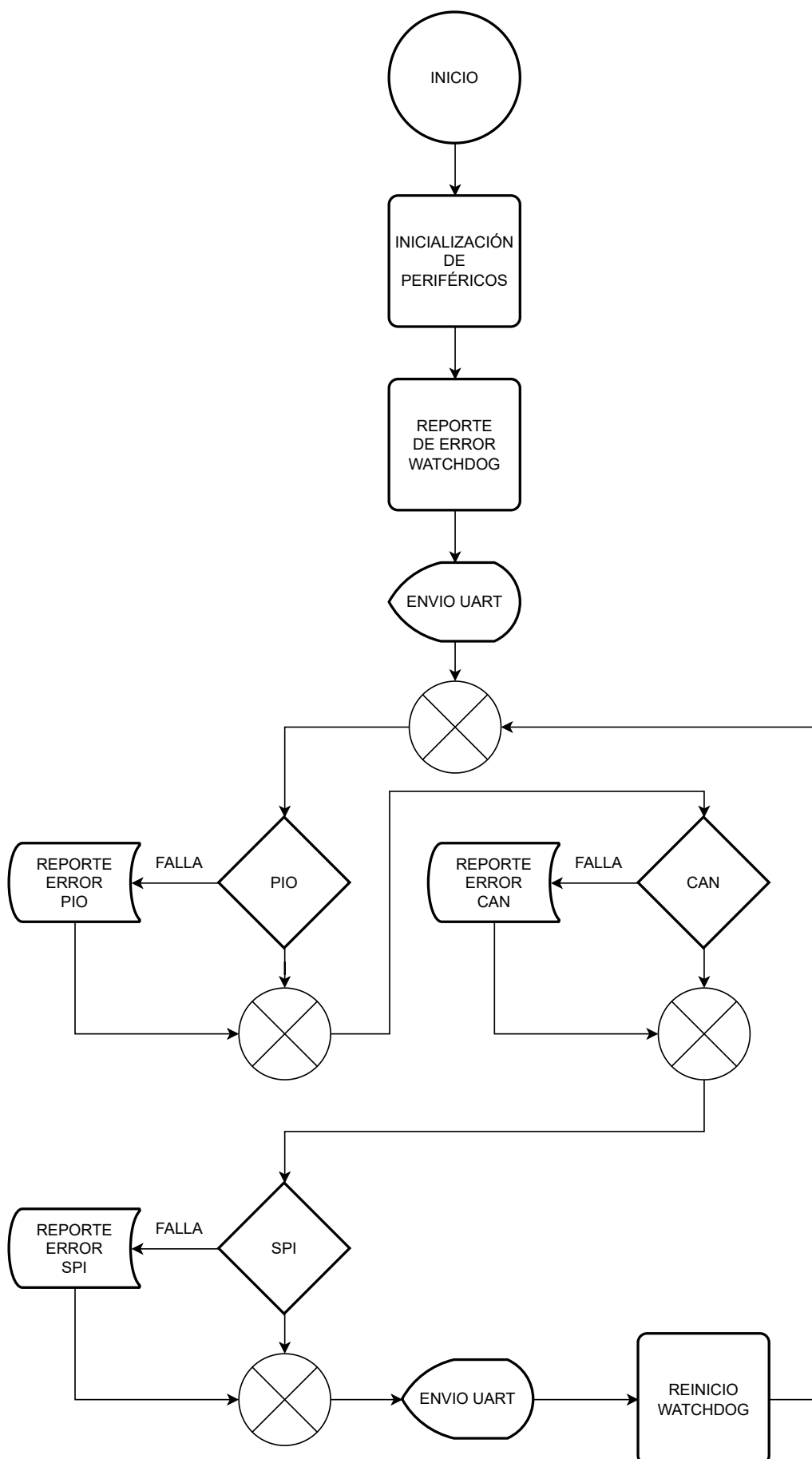
```

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.



FIGURA 3.4. Flujo del *firmware* de autoevaluación.

### 3.2. Interfaz de programación de aplicaciones

La interfaz de programación de aplicaciones tiene la función de abstraer al inyector de errores del servidor *OCD*. Esto se logró con los siguientes patrones de diseño:

- Programación orientada a objetos (*OOP*): este patrón de diseño se basa en agrupar en una unidad lógica las funcionalidades y estados que tengan un alto grado de acoplamiento. Esto significa que las funciones que tienen efectos colaterales junto a los datos mutados se encapsulan dentro de una construcción denominada objeto. Entonces, el principal objetivo de un objeto es contener dentro suyo los efectos colaterales. Además, el lenguaje de programación *Python 3* permite, a través de una *class*, modelar un tipo de dato para instanciar a un objeto. Finalmente, este patrón se utilizó para contener las funciones de lectura y escritura de registros y memorias.
- *Resource Acquisition Is Initialization (RAII)*: este patrón de diseño consiste en modelar el ciclo de vida de un recurso con la implementación de un objeto. Un recurso es todo aquello que requiera mantenimiento luego de su uso, como por ejemplo: liberar memoria, cerrar una conexión o unir dos hilos de un programa. Esto se logra al adquirir un recurso cuando se invoca el constructor de una *class*, por ejemplo, la conexión con la sonda de depuración se realiza durante la instanciación de un objeto llamado conexión. Luego, cuando se desea cerrar la conexión se invoca al destructor del objeto. Dentro de esta función se encuentra el código para cerrar de forma ordenada la conexión con la sonda y el dispositivo bajo prueba. Finalmente, este patrón de diseño hace que el programa maneje de forma robusta los recursos ya que está garantizada la ejecución de los destructores.

En el código 3.3 se puede observar un ejemplo de *RAII* donde se maneja como recurso la detención del núcleo del dispositivo bajo prueba. Esto permite que frente a una excepción del proceso que esté utilizando la interfaz de programación de aplicaciones, el núcleo pueda continuar operando. Finalmente, se posibilita recuperar el proceso sin tener que reiniciar el dispositivo bajo prueba.

```

1 class Halted():
2     def __init__(self, target):
3         self.target = target
4
5     def __enter__(self):
6         self.target.halt()
7
8     def __exit__(self, exc_type, exc_val, traceback):
9         self.target.resume()

```

CÓDIGO 3.3. Ejemplo de *Resource Acquisition Is Initialization (RAII)*.

Los patrones de diseño utilizados permiten escribir funciones expresivas y robustas. Como se puede ver en el código 3.4, es fácil comprender lo que sucede. En la línea 2 se detiene el núcleo y en la línea 3 se lee una posición de memoria. Luego, en la línea 4 el núcleo reanuda su funcionamiento y finalmente, se retorna el valor leído.

```

1 def readMemory(self, addr: int):
2     with Halted(self.target):
3         val = self.target.read_memory(addr)
4     return val

```

CÓDIGO 3.4. Ejemplo de uso de *RAII*.

En la tabla 4.1 se puede observar un resumen de las funcionalidades y sus estrategias de abstracción.

TABLA 3.2. Funcionalidades abstraídas

Funcionalidad	Patrón de diseño	Acceso
Conexión al integrado	RAII	Público
Detener el núcleo	RAII	Privado
Registros CORE: read/write	OOP	Público
Memoria SDRAM: read/write	OOP	Público

Finalmente, se logró abstraer el ciclo de vida de la sesión de depuración que se muestra en la figura 3.5.

### 3.3. Sistema de inyección de *soft-errors*

Una vez lograda la interfaz de programación de aplicaciones explicada en la sección 3.2, se pudo construir el inyector por consola de comandos. Para lograr que el usuario utilice el programa se creó un sistema de archivo de configuración y un cliente de terminal en texto plano. El archivo de configuración se genera en formato *YAML* y se encarga de:

- Definir si el controlador de ensayos debe recoger los reportes del dispositivo bajo prueba.
- Configurar el puerto serie del ordenador.
- Definir la simulación:
  - El tipo de distribución para usar en el planificador de ensayos.
  - La tasa de inyección de errores.
  - La duración del tiempo de exposición en un registro o posición de memoria.
  - La secuencia de exposición de los registros.

En el código 3.5 se puede observar un archivo de configuración. En particular, la configuración que el sistema ofrece como ejemplo al usuario. Es importante notar que no se hace referencia a la sesión de depuración. Finalmente, el ensayo se expresa en términos de parámetros de radiación.

Para facilitar la tarea del usuario, una vez instalado el sistema no es necesario trabajar en una carpeta en particular. El inyector se puede invocar desde cualquier sitio donde el intérprete de Python 3 tenga permisos de ejecución. El operador puede entonces crear una carpeta del ensayo. Luego, crear un archivo de configuración y lanzar desde allí mismo la secuencia de inyecciones. Finalmente, el sistema genera el reporte correspondiente en esa ruta.

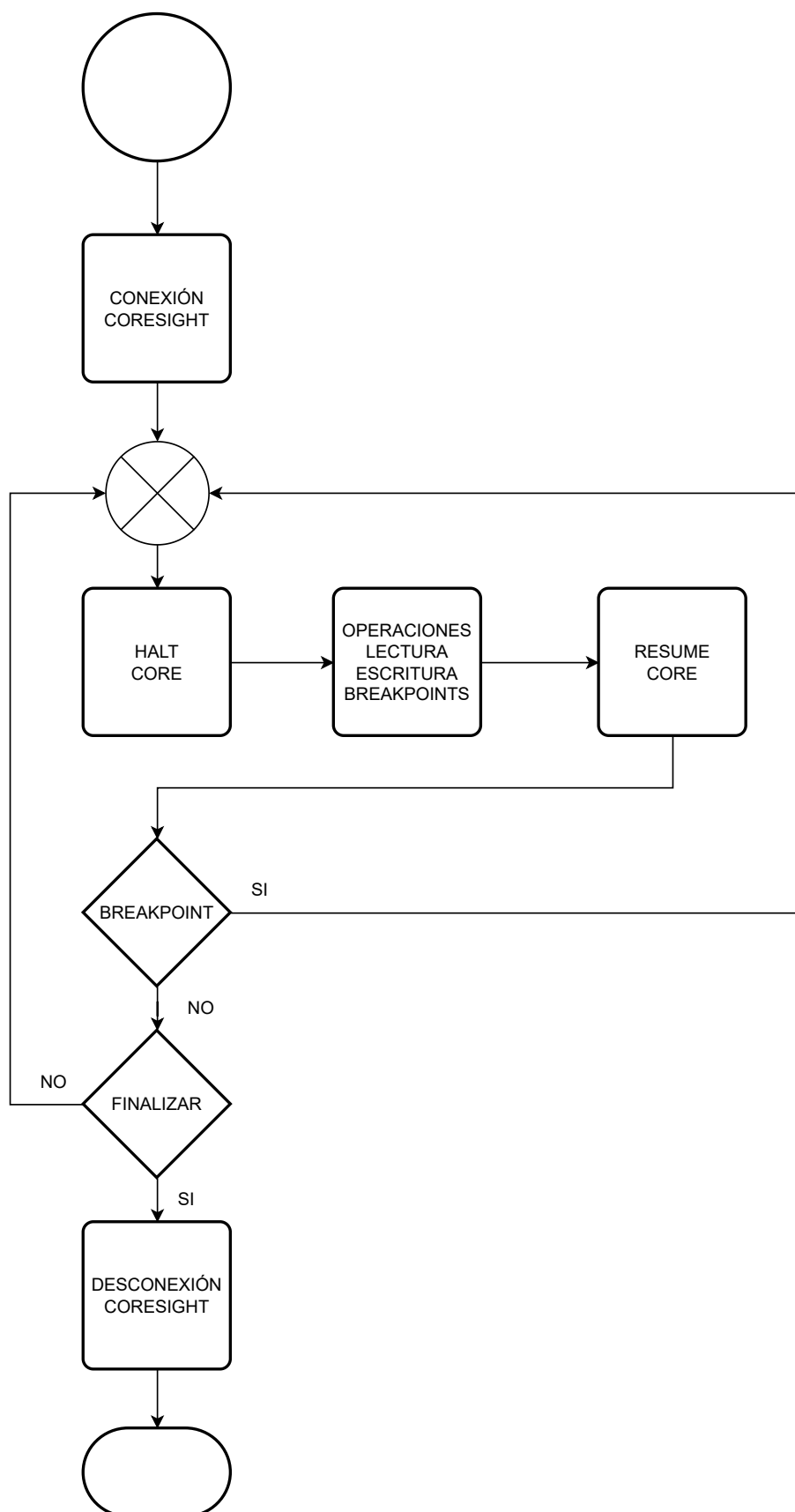


FIGURA 3.5. Flujo de una sesión de depuración.

```
1 # Perfil determina el tipo de ensayo a realizar
2 # 'simulador' solo genera inyecciones mientras
3 # que 'evaluador' recoge y procesa los
4 # reportes del DUT.
5 Perfil: 'simulador'
6
7 # Interfaz define el puerto serie donde se
8 # conecto el DUT.
9 Interfaz:
10     puerto: '/dev/ttyACM0'
11     baudios: 9600
12
13 # Simulacion define la forma de inyectar errores
14 Simulacion:
15     distribucion: 'Poisson'
16     tasa: 0.5
17     duracion: 5
18     registros: 'secuencial'
```

CÓDIGO 3.5. Ejemplo de configuración de ensayo.

La interfaz por consola se logró al utilizar el *standar input* y el *standar output* del proceso padre. En el caso de una sesión iniciada por un humano, el emulador de terminal del sistema. En una primera iteración de la consola se realizó una interfaz ASCII con la biblioteca *curses*. Sin embargo, el cliente prefirió una interfaz por texto plano para facilitar el *parsing* de un sistema de integración continua. En particular, el módulo de *CI/CD* de *Gitlab*.

El usuario invoca el programa al ejecutar el comando `sise`. Luego la consola responde con la leyenda *Ingrese el archivo de configuración >*. Se escribe la ruta y nombre del archivo a utilizar y el sistema lanza el ensayo. El reporte generado se persiste en la carpeta donde se inició el programa.

Con la información suministrada por el archivo de configuración y la consola, se puede iniciar el planificador de ensayos. Este módulo tiene la función de generar una variable aleatoria que genera tiempos entre inyecciones de errores. La variable aleatoria se construye con los datos de distribución y tasa de error. El proceso aleatorio se repite hasta que la sumatoria de los tiempos generados sea igual o mayor al tiempo deseado del ensayo. Luego, se asigna cada tiempo a un registro del núcleo en particular. Esta asignación se realiza según la configuración del usuario. Finalmente, se entrega la planificación al controlador de ensayos.

El controlador de ensayos es un hilo del programa que tiene la misión de ejecutar una planificación de ensayos. Su ejecución se basa en la interfaz de abstracción de aplicaciones explicada en la sección 3.2. Luego, tiene la responsabilidad de observar el tiempo de ejecución del dispositivo bajo prueba. Cuando el momento es adecuado, el controlador invoca una función de *bit flip*. El bit a invertir se determina en el momento de inyección con una variable aleatoria uniforme. Finalmente, cuando el controlador consume la totalidad de la planificación, envía una señal para solicitar el *join* con el hilo principal del programa.

El generador de reportes es un hilo que escucha el puerto serie del ordenador. Mientras dura el ensayo, almacena todos los mensajes entrantes y los acumula junto a un *timestamp*. De la misma manera, persiste las inyecciones realizadas como tuplas. Estas tuplas tienen todos los datos relevantes de la inyección junto a un *timestamp*. Cuando el ensayo finaliza, recibe una señal desde el hilo principal del programa que le ordena hacer un *join*. Luego, el generador de reportes deja de

escuchar el puerto serie y al controlador de ensayos. Seguidamente, se procede a construir un reporte en formato de tabla de *MS Excel* y *CSV*. Para lograrlo, se genera una relación de causalidad entre errores inyectados y respuestas del dispositivo bajo prueba. Finalmente, se generan los archivos en la carpeta donde el comando `sise` fue lanzado.

La mayor dificultad del inyector fue el manejo concurrente de los hilos del controlador de ensayo y el generador de reportes. En particular, porque ambos hilos comparten el uso del puerto serie. Esta situación genera una condición de carrera que se tuvo que manejar. Luego, se decidió evitar candados y semáforos al explorar la topología de *bus* en árbol del protocolo *USB*. De esta manera, la conexión con la *UART* y el *DAP* se trató como si estuviesen conectados en puertos físicos diferentes. Esta decisión de diseño tiene la ventaja de no incrementar el error de las mediciones de tiempo. Un error en la medición generaría relaciones de causalidad incorrectas y por lo tanto los informes no serían confiables. Sin embargo, se introdujo un punto de falla al depender de la capacidad de la sonda de depuración y su gestión de su árbol de dispositivos. Finalmente, en la figura 3.7 se puede observar un diagrama con los hilos del programa.

### 3.4. Biblioteca para el desarrollo de ensayos

Para que el usuario pueda realizar sus propios ensayos, se creó una biblioteca que le provee la funcionalidad necesaria. Esta biblioteca se divide en dos partes:

- Funcionalidades del núcleo.
- Funcionalidades de la memoria.

Para facilitar la interacción con el núcleo, se implementó una lista con los nombres de los registros. De esta manera, se facilita el uso de las funciones. En el código 3.6 se puede observar la colección que se le ofrece al diseñador.

```
1 CORE_REGISTERS = [
2     'lr', 'pc', 'sp', 'xpsr', 'r0', 'r1', 'r2', 'r3',
3     'r4', 'r5', 'r6', 'r7', 'r8', 'r9', 'r10', 'r11', 'r12'
4 ]
```

CÓDIGO 3.6. Lista de registros accesibles por el usuario.

Las funcionalidades ofrecidas para manipular el núcleo del integrado se muestran en el código 3.7. Los detalles son los siguientes:

- En la línea 6 se muestra el uso de la función de lectura de registros. Se invoca a partir del objeto de conexión y su argumento es el nombre del registro a leer.
- En la línea 10 se puede observar el uso del método de escritura de registros. Necesita como argumento el nombre del registro y el valor a escribir. Luego, la función retorna una tupla con los valores previos y posterior a la escritura.
- En la línea 15 se ve una llamada a la función de *bit flip* del registro del núcleo. Se debe indicar el nombre del registro y la posición del bit a invertir. Finalmente, se retorna el valor previo y posterior al llamado del método.

```
1 import sise.library as sise
2
```

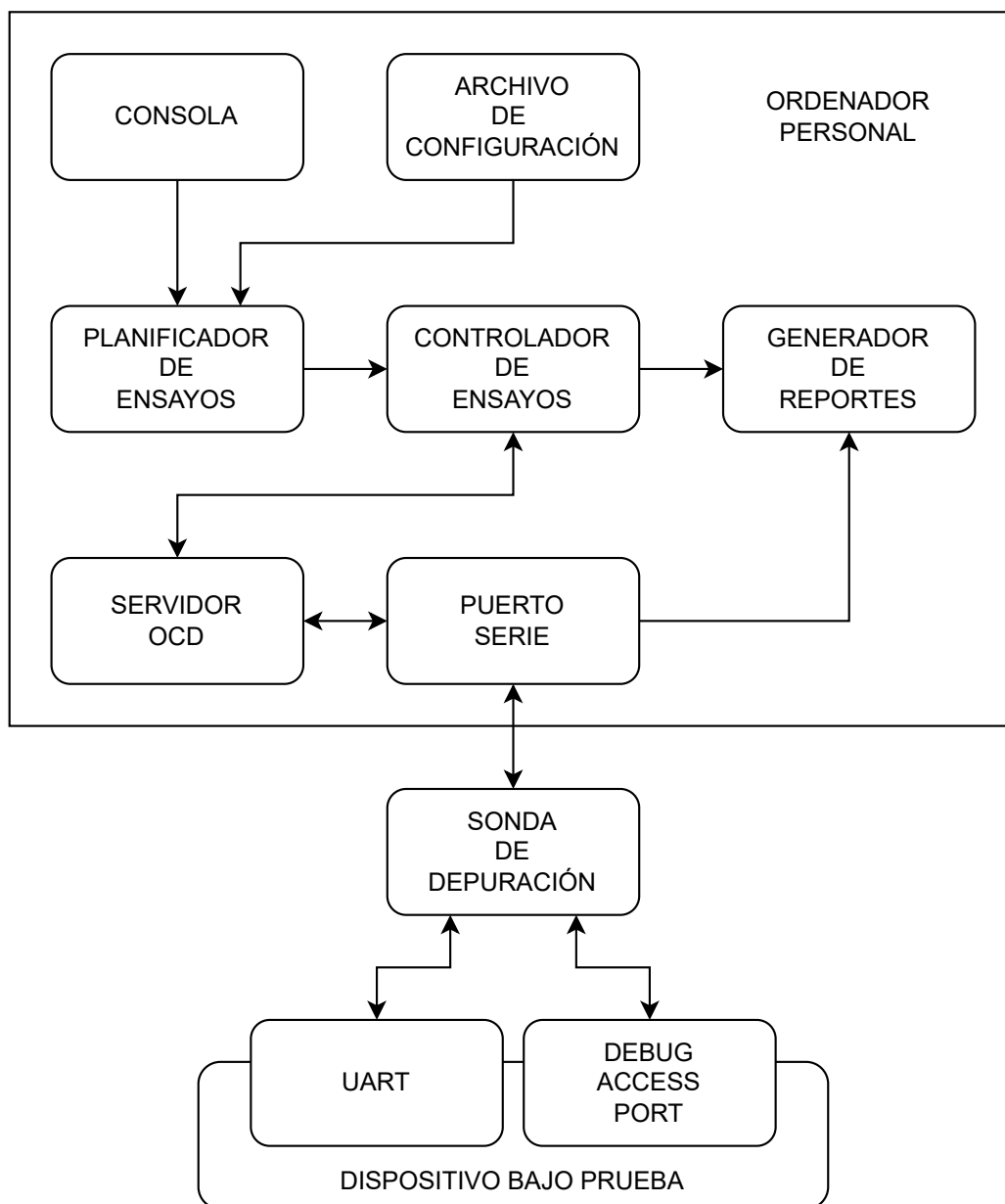


FIGURA 3.6. Diagrama en bloques del sistema de inyección de soft-errors.

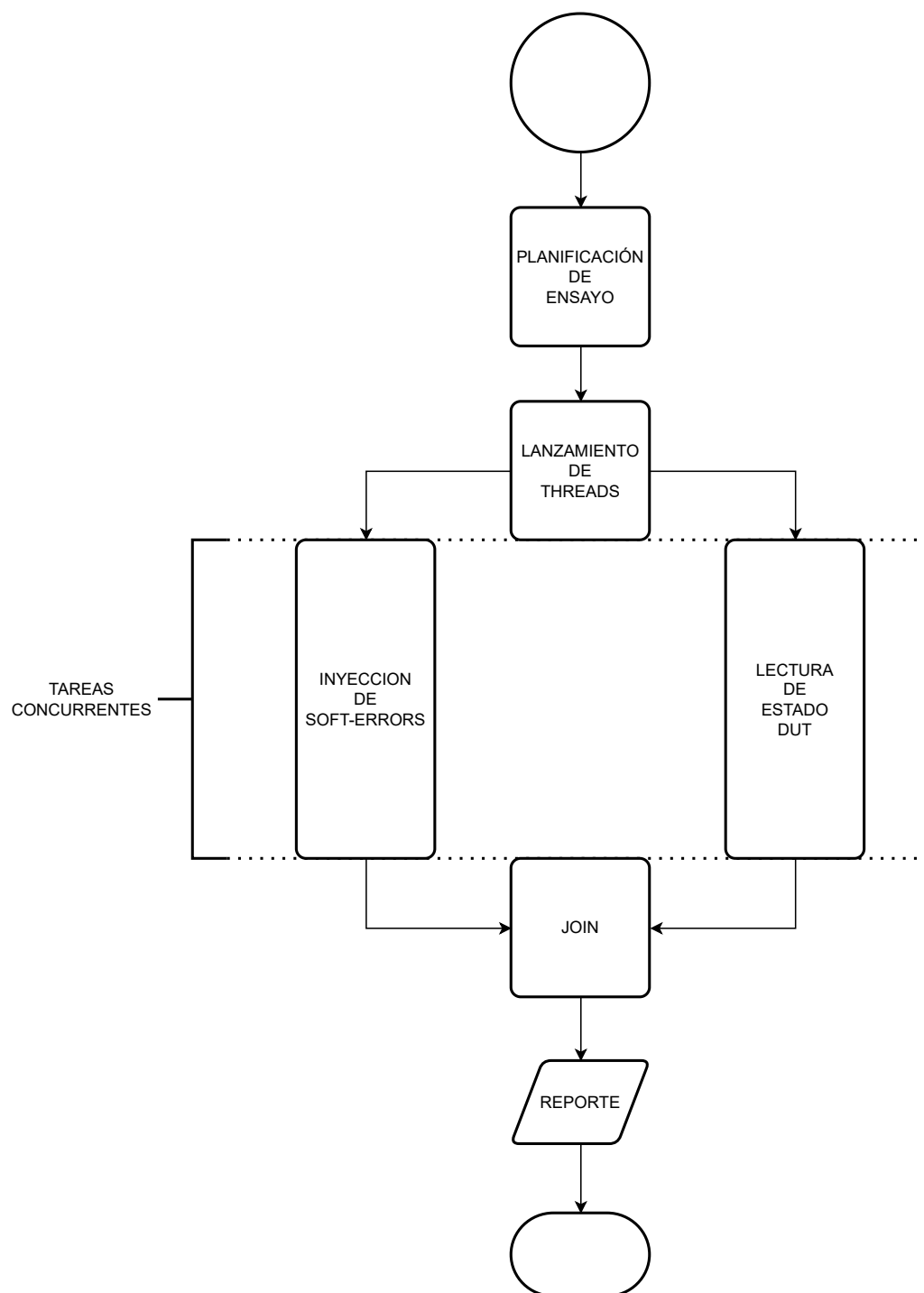


FIGURA 3.7. Flujo de tareas concurrentes.



```

3 dut = sise.Connection()
4
5 # Lectura del registro del CORE
6 rreg = dut.readRegister('pc')
7 print("PC:", rreg)
8
9 # Escritura del registro del CORE
10 wreg = writeRegister('r0', 0xffffffff)
11 print("(old, new):", wreg)
12
13 # Bit-flip en registro del CORE
14 bit = 2
15 bfreg = bitFlipRegister('r1', bit)
16 print("(old, new):", bfreg)
17
18 del(dut)

```

CÓDIGO 3.7. Ejemplo de uso en registros del núcleo.

Para interactuar con la memoria se dispone de las funciones demostradas en el código 3.8. Los métodos se detallan a continuación:

- Línea 7: el método permite realizar una lectura del dato en una posición de memoria. El argumento es una dirección alineada de la memoria y el retorno es el valor de una palabra de 32 bits.
- Línea 12: esta función se utiliza para escribir una posición alineada de memoria. Se necesita pasarle una dirección y un valor a escribir. Finalmente, retorna una tupla con el dato previo y posterior a la ejecución del método.
- Línea 18: la subrutina posibilita hacer un *bit flip* en una posición de memoria. El método toma como argumento una posición alineada de memoria y el bit a invertir. Luego de su ejecución, se retorna el valor previo y posterior a la inversión.

```

1 import sise.library as sise
2
3 dut = sise.Connection()
4
5 # Lectura de memoria
6 addr = 0x20400004
7 rmen = readMemory(addr)
8 print("men:", rmen)
9
10 # Escritura de memoria
11 addr = 0x20400008
12 wmen = writeMemory(addr, 0xfafafafa):
13 print("(old, new):", wmen)
14
15 # Bit-flip en memoria
16 addr = 0x20400000
17 bit = 0
18 bfmen = dut.bitFlipMemory(addr, bit)
19 print("(old, new):", bfmen)
20
21 del(dut)

```

CÓDIGO 3.8. Ejemplo de uso en memoria.



## Capítulo 4

# Ensayos y resultados

### 4.1. Laboratorio remoto

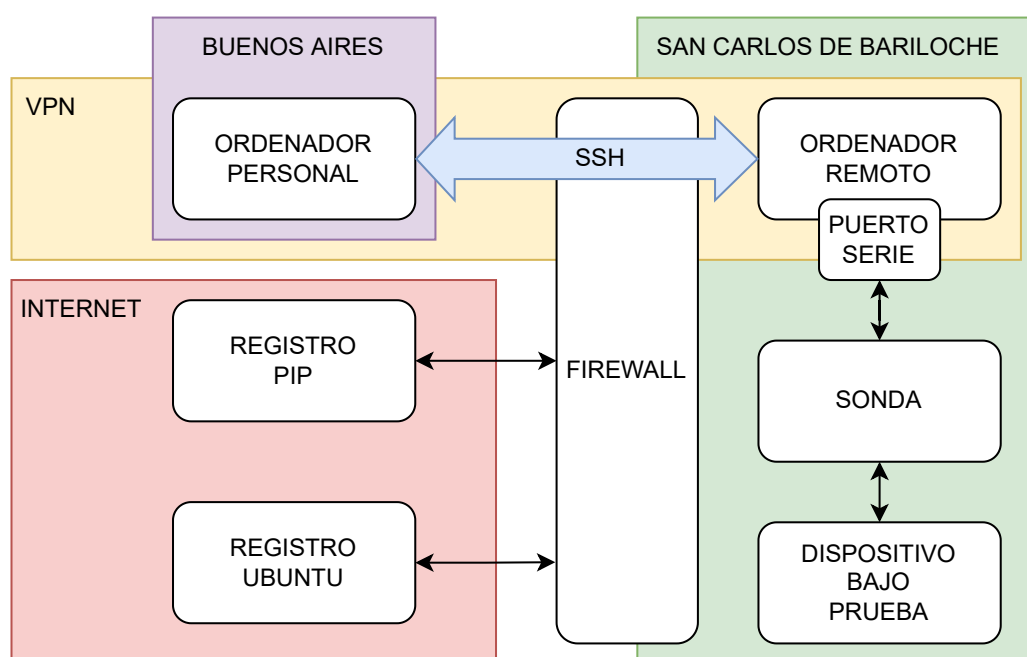


FIGURA 4.1. Diagrama en bloques del laboratorio remoto.

TABLA 4.1

Funcionalidad	Nivel de servicio
Carga de binarios en DUT	++
Comunicación con registro PIP	+++
Comunicación con registro Ubuntu	+++
Comunicación con debug access port	+++
Comunicación con UART	+

### 4.2. Ensayos de inyector

### 4.3. Validación con el cliente

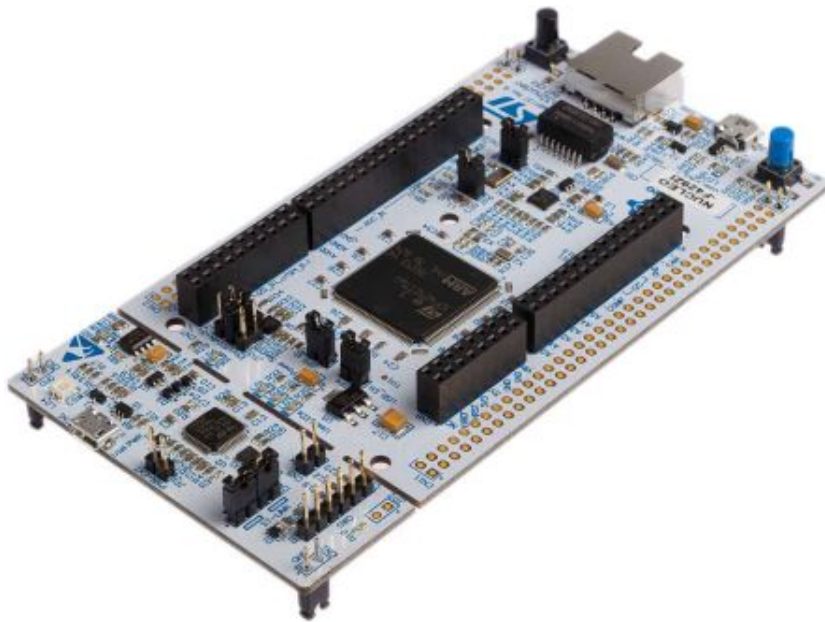
FIGURA 4.2. Dispositivo alternativo *NUCLEO-F429ZI*.

TABLA 4.2. Resumen de ensayos

Ensayo	Lab. local	Lab. remoto	DUT alerno
Escritura SDRAM	+++	+++	+++
Escritura registros CORE	+++	+++	+++
Funcionalidades extras	++	++	+++
Halt CORE	+++	+++	+++
Lectura SDRAM	+++	+++	+++
Lectura registros CORE	+++	+++	+++
Uso concurrente de puerto serie	+++	+	+++
Resume CORE	+++	+++	+++

TABLA 4.3. Resumen de la validación con el cliente

Expectativas	Cumplimiento
Acceso a memoria	+++
Acceso al CORE	+++
Biblioteca de ensayos	+++
Capacidad de bit-flip	+++
Configuración del sistema	+++
Distribución de errores	+++
Validación de periféricos	+++
Generación de reportes	+++

## 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.

## 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.

# Bibliografía

- [1] Steven J. Dick. *Remembering the Space Age*. National Aeronautics y Space Administration, 2008.
- [2] J. Hickman y E. Dolman. *Resurrecting the Space Age: A State-Centered Commentary on the Outer Space Regime*. National Aeronautics y Space Administration, 2002.
- [3] National Aeronautics y Space Administration. «Human Space Flight Transition Plan». En: *NASA Archive* (2008).
- [4] INVAP. INVAP SE. <https://www.invap.com.ar/>. Mayo de 2022. (Visitado 01-05-2022).
- [5] spaceradiation.eu. *Structure of space radiation*. <https://spaceradiation.eu/structure-of-space-radiation/>. Mayo de 2022. (Visitado 01-05-2022).
- [6] Raoul Velazco. «Inyección de fallos para el análisis de la sensibilidad a los errores transitorios, “soft errors”, provocados por las radiaciones en circuitos integrados». En: *Architectures Robustes of Integrated circuit and systems, Grenoble - France* (2014).
- [7] spaceradiation.eu. *Effects of space radiation on electronic devices*. <https://spaceradiation.eu/effects-of-space-radiation-on-electronic-devices/>. Mayo de 2022. (Visitado 01-05-2022).
- [8] Michael D. Griffin. «NASA and the Business of Space». En: *American Astronautical Society 52 Annual Conference* (2005).
- [9] Roberto Cibils. «Don't look up. Starlink project: bold venture or economic bubble?». En: *Mission Project Workshop 24/25 feb 2022* (2022).
- [10] ucl.ac.uk. *Heavy ion latchup tests in louvain la neuve*. [https://www.ucl.ac.uk/mssl/sites/mssl/files/styles/owl\\_carousel/public/heavy\\_ion\\_latchup\\_tests\\_in\\_louvain\\_la\\_neuve.jpg?itok=1dDe-TEo](https://www.ucl.ac.uk/mssl/sites/mssl/files/styles/owl_carousel/public/heavy_ion_latchup_tests_in_louvain_la_neuve.jpg?itok=1dDe-TEo). Mayo de 2022. (Visitado 01-05-2022).
- [11] ATMEL. «Atmel 44003 32 bit Cortex M7 Microcontroller SAMV71». En: *Atmel Smart ARM-based Flash MCU* (2016).
- [12] Microchip. «SAMV71RT Radiation Test Report». En: *Radiation Test Report 4.4* (2019).
- [13] developer.arm.com. *Cortex M4*. <https://developer.arm.com/>. Mayo de 2022. (Visitado 01-05-2022).
- [14] How to debug: CoreSight basis (Part 3). *Cortex M4*. <https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/how-to-debug-coresight-basics-part-3>. Mayo de 2022. (Visitado 01-05-2022).
- [15] Stand Shebs Richard Stallman Roland Pesch. *Debugging with GDB: The GNU Source-Level Debugger*. GNU Press, 2002.
- [16] PyOCD. *PyOCD*. <https://github.com/pyocd/pyOCD>. Mayo de 2022. (Visitado 15-05-2022).