Facultad de Ciencias

DESARROLLO DE UN MODO DE EJECUCIÓN RÁPIDA "GO-FAST" PARA EL DEPURADOR !UCDEBUG

(Development of a swift “Go-Fast” execution mode for the !UCDebug ARM debugger)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: David Herreros Cerro

Director: Pablo Fuentes Sáez

Co-Director: Cristóbal Camarero Coterillo

Septiembre - 2020

Resumen

Las asignaturas del área de Estructura y Organización de Computadores en la Universidad de Cantabria utilizan la arquitectura ARM de referencia para sus lecciones tanto teóricas como prácticas. Durante los laboratorios de algunas de estas asignaturas los alumnos utilizan el depurador !UCDebug para ejecutar y depurar su código. Parte de estas prácticas se realizan empleando periféricos, de los cuales algunos tienen requerimientos temporales estrictos para su correcto funcionamiento. El depurador no contaba con una forma de ejecutar códigos que utilizasen estos dispositivos debido a que sus modos de ejecución introducían un gran *overhead*, haciendo que no se pudiera interactuar con los dispositivos correctamente.

En este proyecto se busca implementar un nuevo modo de ejecución para !UCDebug que proporcione tiempos de ejecución similares a los de la ejecución en el sistema nativo. Esto permitiría poder utilizar una mayor variedad de periféricos en los laboratorios de las asignaturas.

Antes de comenzar con el desarrollo de este nuevo modo, se desarrolló una suite de tests por la falta de un mecanismo que verificase el funcionamiento del depurador. Esta suite de tests tiene como propósito el poder comprobar que las funcionalidades del depurador se ajusten a lo esperado.

El nuevo modo, denominado modo Go-Fast, emplea una estrategia basada en el desarrollo de un handler de las SWIs. Como desarrollo adicional, se buscó una forma de que el usuario pueda interactuar con el depurador mientras ejecuta en este modo, para lo que se optó por un mecanismo de interrupciones periódicas.

Con el desarrollo realizado en este proyecto el depurador ha demostrado que puede llegar a ejecutar correctamente los mismos códigos de ensamblador ARM que el sistema nativo, hasta el punto que se ha comprobado el depurador.

Palabras clave: Depurador, Docencia, Prácticas de laboratorio, ARM, Raspberry Pi, RISC OS

Abstract

Computer Structure and Organization courses at the University of Cantabria follow the ARM architecture as a reference for their lectures and practical sessions. In some of these courses, students use the !UCDebug ARM debugger at the computer lab to run and debug their codes. Part of these practical lessons are conducted using peripheral devices, some of which have strict time requirements for their correct functioning. The debugger was unable to properly execute code that interacted with such devices because its execution modes introduced a large overhead, making it impossible to meet their time requirements.

This project seeks to implement a new execution mode for !UCDebug that provides execution times similar to those of the execution in the native system. This would allow a greater variety of peripherals to be used in these courses.

Before starting the development of this new mode, a test suite was developed to address the lack of a mechanism that checks the debugger behavior. This test suite was developed in order to verify that the debugger's functionalities match a specific intended behavior.

The new mode, called Go-Fast mode, employs a strategy based on the development of a SWI handler. As a further development, a way for the user to interact with the debugger while running in this mode was sought, for which a periodic interrupt mechanism was chosen.

With the development conducted in this work, the debugger has shown that it can successfully execute the same ARM assembly codes as the native system, to the extent that the debugger has been tested.

Agradecimientos

Primero, me gustaría agradecer a mis directores Pablo y Cristóbal por su dedicación y paciencia durante todos estos meses, ya que sin ellos y su incansable asistencia no habría logrado terminar este proyecto. También quiero agradecer a mis amigos y familia, por estar siempre ahí cuando los he necesitado, sobre todo, en los tiempos más difíciles. Por último, quiero agradecer en especial a mis padres por todo su apoyo a largo de los años, por transmitirme los valores de esfuerzo y trabajo, y por brindarme la oportunidad de llegar hasta aquí.

Muchas gracias a todos.

Índice

Índice de figuras

Índice de tablas

1. Introducción

Este trabajo nace como respuesta a la necesidad de mejorar una herramienta de depuración de código en ensamblador ARM, el depurador !UCDebug. Este depurador se utiliza en varias asignaturas del área de Estructura y Organización de Computadores de los Grados en Ingeniería Informática y en Ingeniería de Tecnologías de Telecomunicación de la Universidad de Cantabria. Se trata de un programa de código libre que corre sobre el sistema operativo RISC OS empleando la plataforma Raspberry Pi. La herramienta permite a los alumnos de dichas asignaturas verificar el comportamiento y funcionalidad de sus códigos durante la realización de prácticas de laboratorio, ayudando a afianzar los conceptos aprendidos de forma teórica.

Sin embargo, al analizar el uso de dispositivos periféricos en la realización de dichas prácticas, se observa que el *overhead* (o sobrecarga de tiempo de ejecución) que involucra el depurador hace inviable el uso de algunos dispositivos con requerimientos de tiempo muy estrictos. Asimismo, el aumento significativo del tiempo de ejecución de los programas al emplear el depurador implica que las constantes de manejo de otros dispositivos tengan que ser distintas de las empleadas si el programa se ejecuta de forma autónoma.

Para paliar estas carencias, se hace necesario desarrollar e implementar una forma de ejecución alternativa que reduzca el overhead causado por la interacción con el depurador, manteniendo los beneficios que éste aporta. A esta nueva forma de ejecución del depurador, que se suma a las ya existentes previamente, se le ha denominado modo Go-Fast. Para garantizar que el desarrollo de este nuevo modo no interfiera con la funcionalidad implementada previamente, también se ha desarrollado una suite de tests que permiten verificar y garantizar el correcto funcionamiento de la herramienta.

1.1. Motivación

Las asignaturas del área de Estructura y Organización de Computadores se imparten con una arquitectura de referencia, la cual se emplea como modelo para la explicación de los diversos conceptos. En el caso de la Universidad de Cantabria, se empleó la arquitectura MIPS desde 2003 hasta 2018, combinando hardware real con un simulador en la realización de prácticas de laboratorio de estas asignaturas. Concretamente, se hacía uso de un puesto de laboratorio equipado con placas Mycable XXS1500, dotados de un microprocesador AMD Au1500 basado en la arquitectura MIPS, que se manejaban de forma indirecta a través de un PC convencional. Además, para algunas de las prácticas se empleaba el simulador Spim corriendo sobre el sistema operativo Windows.

La arquitectura MIPS presentaba diversas ventajas por su orientación docente y su simplicidad, pero se empezó a observar una alta tasa de abandono y una baja tasa de aprobados. Los comentarios recibidos en las encuestas de calidad atribuían estas estadísticas a una baja motivación en aprender una arquitectura con poco impacto durante los últimos años y a las condiciones dificultosas para trabajar en las prácticas de las asignaturas, puesto que había que realizar un proceso de aprendizaje para dos entornos de laboratorio distintos (simulador, y hardware real), duplicando el esfuerzo y tiempo invertido [1].

Además de esto, ambos entornos tenían carencias que hacían que trabajar en ellos afectara negativamente a los alumnos. El simulador se utilizaba como punto de primer contacto con la arquitectura, por lo que sólo se utilizaba para desarrollos de códigos sencillos. Al trabajar con el simulador, existía un distanciamiento con la ejecución sobre un hardware real que desmotivaba a los alumnos.

En la parte del desarrollo utilizando los equipos especializados sí que podían ejecutar directamente sobre un hardware existente, pero resultaba problemático para los alumnos por otros motivos: el alto coste de los equipos, que imposibilitaba el trabajo autónomo fuera de las horas de laboratorio, la falta de una herramienta de desarrollo que fuera interactiva y fácil de usar dificultaba el depurado de los códigos, y la monotonía del desarrollo de las prácticas en un entorno cerrado.

Para solventar estos problemas a partir del curso 2017-18 se decide cambiar de arquitectura de referencia. Este cambio de arquitectura conlleva un cambio en la materia impartida, en el repertorio de prácticas a desarrollar en los laboratorios y un cambio en el hardware. Con motivo de afrontar todos estos cambios se desarrollan una serie de proyectos de innovación docente encargados de ajustar las asignaturas a estos requisitos.

En estos proyectos se contemplan las múltiples arquitecturas disponibles, de las que se acaba optando por utilizar ARM. Se elige esta arquitectura por su gran impacto en el mercado, su alta relevancia en la actualidad, por la disponibilidad de hardware a precio reducido, por la alta variedad de periféricos disponible y por su uso ya establecido en varias universidades a nivel nacional [2].

Durante la elección del hardware a utilizar para las prácticas se tuvieron en cuenta los factores de coste, disponibilidad, si es ampliable con hardware externo, facilidad de uso, características del hardware, e independencia del uso de simuladores. Tras valorar los sistemas hardware disponibles se acabó eligiendo la computadora Raspberry Pi [3]. El uso de la Raspberry Pi presenta una serie de ventajas a la hora de trabajar en el laboratorio que la hacen superior al resto de alternativas:

- Precio reducido.
- Disponibilidad de periféricos a precios muy asequibles, permitiendo plantear prácticas más variadas.
- Capacidad para ejecutar un sistema operativo sobre el que el alumno pueda realizar las prácticas en un entorno familiar, haciendo además que el uso de un equipo externo para trabajar sea optativo.
- Comunidad de desarrollo grande y activa.
- Puerto Ethernet que nos permite conectarnos a internet directamente desde el dispositivo.
- Posible reutilización del hardware para otros proyectos de la universidad por su gran versatilidad.

Con ello se consigue que el alumno trabaje en un único sistema, el cual utilizará tanto en las sesiones de laboratorio como en las horas de trabajo autónomo. Esto aumenta la familiaridad y cercanía con el hardware y software empleado, y disminuye el tiempo empleado en aprender a trabajar en los entornos de desarrollo.

De los múltiples modelos de la Raspberry Pi considerados se decidió utilizar el modelo 1 B+ [4] para los laboratorios de las asignaturas de Introducción a los Computadores, Estructura de Computadores y Microprocesadores. Se optó por este modelo en detrimento de otros más modernos debido a la falta de una documentación completa del procesador de los otros modelos y por errores de compatibilidad con el sistema operativo elegido, RISC OS [5]. Las características de este modelo se detallan en la Tabla 1.

Características Raspberry Pi modelo 1B+	
Tamaño	8.6 x 5.4 x 1.7 cm
Memoria RAM	512 MB
Procesador	Broadcom BCM2835
Velocidad de reloj	700 MHz
Multitarea	Sí
Tarjea gráfica	Dual Core VideoCore IV® Multimedia Co-Processor
Ethernet	10/100 Base Ethernet RJ45
Salida de vídeo	Puerto Full HDMI
Salida de audio	Conector de audio combinado de 3,5 mm y vídeo compuesto
Conector de Cámara	MIPI Camera Serial Interface (CSI-2)
Conector de Display	Display Serial Interface (DSI)
Voltaje de entrada	5V
Memoria Flash	Tarjeta SD (de 2 a 16 GB)
Puertos USB	Cuatro puertos USB 2.0
Conector de la GPIO	40 pines, 2 filas
Sistema Operativo	Distribuciones Linux
Entorno de desarrollo integrado (IDE)	Scratch, IDLE, cualquiera con soporte Linux
Precio	35€

Tabla 1 – Principales características de la Raspberry PI modelo 1 B+. Información obtenida de [2], [5] y [6]

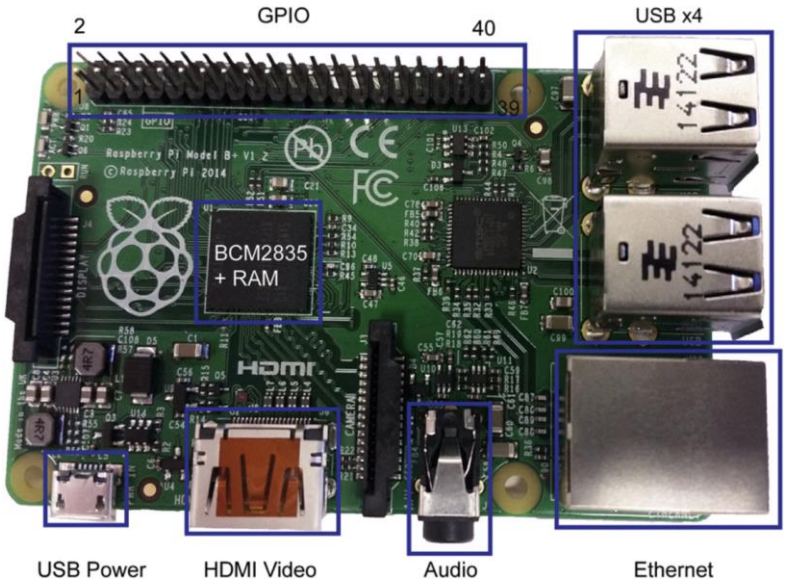

Figura 1 – Placa Raspberry Pi modelo 1B+ con los componentes desglosados. Imagen extraída de [7].

Como se ha mencionado antes, a la hora de elegir la Raspberry Pi como hardware de desarrollo se tuvo como factor la capacidad para ejecutar un sistema operativo, disponiendo de varias opciones, como Raspbian, Arch Linux o RISC OS entre otros [2]. De entre ellos se optó por RISC OS [8] por ser ligero y sencillo, por su interfaz gráfica en ventanas de carácter similar a los sistemas operativos con los que los alumnos están acostumbrados a trabajar, por su alta capacidad de manejar dispositivos de E/S a muy bajo nivel y por permitir trabajar con programas

que usen tanto el lenguaje C como ensamblador ARM [5]. Aunque se trate de un sistema ligero y sencillo cuenta con herramientas de trabajo muy útiles para el desarrollo de código, como son un editor de textos con resaltado de sintaxis [9] o la suite de compilación de GNU, GCC [10].

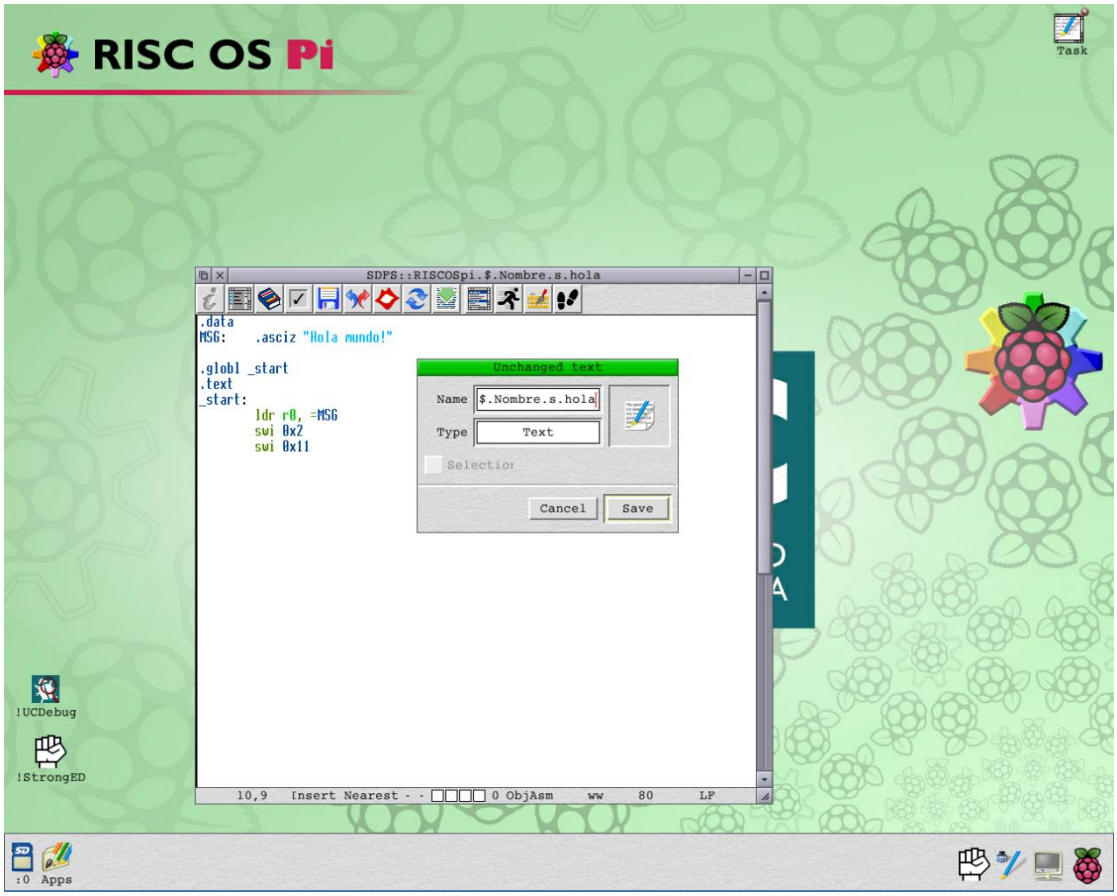

Figura 2 – Captura de pantalla del sistema operativo RISC OS, con una ventana abierta del editor de textos !StrongED.

La única limitación grave que se detecta en RISC OS es la falta de un depurador para el código ensamblador ARM que tenga gran facilidad de uso, una interfaz gráfica y un coste de licencia bajo o gratuito.

Para responder a esta necesidad se desarrolla el depurador !UCDebug en el marco de un proyecto de Innovación Docente de la IV Convocatoria del Vicerrectorado de Ordenación Académica y Profesorado de la Universidad de Cantabria. El depurador se diseña como una herramienta de depuración para los alumnos de las asignaturas de hardware, aunque cualquier otro interesado es libre de utilizarlo. El depurador se empieza a emplear en el curso 2018-19 una vez llegado al nivel de estabilidad y robustez necesario para su uso por los alumnos en el desarrollo de las prácticas.

Como parte de este proyecto de innovación docente se analiza el uso de periféricos de bajo coste para la Raspberry Pi. Esta iniciativa busca aportar a las prácticas un carácter más interactivo, fomentando así el interés de los alumnos. Los dispositivos analizados son la placa BerryClip [11], equipada con 6 leds, 1 buzzer y un interruptor; el sensor de temperatura y humedad DHT11 [12], un display LCD de 16 x 2 [13], el sensor de distancia por ultrasonidos HC-SR04 [14] y un lector de tarjetas RFID [15].

Figura 3 – Dispositivos periféricos contemplados para la realización de prácticas de laboratorio. En sentido contrario a las agujas del reloj, comenzando por arriba: sensor de distancia HC-SR04, sensor de temperatura DHT11, placa BerryClip y display LCD.

Durante el desarrollo de las pruebas de algunos de estos dispositivos se detectaron mediciones incorrectas y resultados anómalos. Esto es debido a que el modo de ejecución de código del depurador introduce un gran *overhead* a la ejecución de cada instrucción, provocando que los tiempos de ejecución totales aumenten en cuatro órdenes de magnitud. Esto produce que a la hora de leer y/o escribir en algunos dispositivos (como el sensor de temperatura y humedad) se produzcan errores al no poder cumplir los requisitos temporales. Por este motivo se determina la necesidad de añadir un modo de ejecución al depurador que proporcione mejores tiempos de ejecución.

El modo desarrollado se denominó “modo Go-Fast”, y busca ofrecer un mejor rendimiento a cambio de tener un menor control en la ejecución del código. Como parte del proyecto se exploró una implementación inicial de este modo, pero su inestabilidad y bajo rendimiento hizo que tuviera que descartarse.

1.2. Desarrollo realizado

El objetivo principal de este proyecto ha sido desarrollar una implementación del modo Go-Fast que se ajuste a los requisitos temporales de cualquier periférico que se quiera utilizar.

La metodología de trabajo empleada ha sido la de desarrollo iterativo e incremental. Esta metodología resulta muy útil en desarrollos donde se busca añadir nueva funcionalidad a un software preexistente, ya que se hace el desarrollo de una parte del proyecto que aporta una funcionalidad nueva y se comprueba que su funcionamiento sea el esperado. Esto permite aprovechar el conocimiento adquirido en cada fase para el desarrollo de las subsiguientes, y tener nuevas funcionalidades que comprobar, recibiendo retroalimentación sobre el funcionamiento del sistema, que evita tener que realizar grandes cambios al final del desarrollo. Para ello, el proyecto se ha subdivido en partes más pequeñas e independientes entre sí: la suite de tests, el handler de las SWIs y las interrupciones periódicas del modo Go-Fast.

Para poder comenzar con la implementación del modo Go-Fast se contempla el desarrollo de un mecanismo para verificar que los cambios introducidos en el depurador no dañan las funcionalidades previamente implementadas. Con este fin se desarrolla una suite de tests que comprueba el correcto funcionamiento del depurador. Anteriormente, el depurador no contaba con un mecanismo de este tipo, dificultando el desarrollo del depurador y aumentando el número de errores encontrados durante el desarrollo de las prácticas en el laboratorio.

Gracias a estos tests se pudo verificar que el funcionamiento del depurador en el estado previo al desarrollo del modo Go-Fast fuera el esperado, detectándose durante las pruebas algunos errores que podrían dar problemas en el futuro. También proporcionan un mecanismo de desarrollo que minimiza el número de errores sin detectar en el código, aumentado considerablemente la calidad del código desarrollado, y simplifica la labor de depuración de estos errores.

El modo Go-Fast emplea una estrategia basada en la creación de un handler para las SWIs. Este handler permite una gestión eficiente y liviana de las SWIs del código del usuario, haciendo que en el modo Go-Fast el depurador no tenga que gestionar esa tarea, la cual sería el mayor limitante para conseguir los tiempos de ejecución requeridos.

Una vez implementado el modo Go-Fast, y comprobado que funciona gracias a la suite de tests, se decidió agregar una nueva funcionalidad que mejore la usabilidad de este nuevo modo. Esta funcionalidad busca solucionar el problema de que cuando se ejecuta un código en modo Go-Fast no se puede interactuar con el depurador hasta que se termine la ejecución del código. Para ello se desarrolla un sistema de interrupciones periódicas que permite al usuario poner comandos por la consola del depurador sin que se disminuya significativamente el rendimiento del modo Go-Fast.

El resto de este documento se organiza en cuatro capítulos, donde se detalla el trabajo realizado, además de los conceptos necesarios para entender la arquitectura y el depurador. En el capítulo dos se explican las características de la arquitectura ARM, sus principales mecanismos y cómo trabajar con el depurador, estableciendo los conceptos base del sistema con el que se desarrolla el proyecto. En el capítulo tres se explica el desarrollo del modo Go-Fast, contando con dos secciones principales en las que se detalla el handler de las SWIs y las interrupciones periódicas del modo Go-Fast. En el capítulo cuatro se detalla la suite de tests, explicando cómo funcionan los distintos tests desarrollados. Este documento queda cerrado con un capítulo de conclusiones y resultados obtenidos.

2. Características del sistema

El desarrollo realizado en este proyecto parte del depurador !UCDebug realizado en un Proyecto de Innovación Docente anterior. Para comprender el desarrollo realizado y las decisiones de diseño tomadas es necesario conocer en cierto detalle la arquitectura ARM y el funcionamiento y características del depurador. Este capítulo busca explicar los conceptos más relevantes de ambos apartados para facilitar la explicación del trabajo realizado.

2.1. Arquitectura ARM

ARM es una arquitectura hardware de tipo RISC (Reduced Instruction Set Computer) que actualmente se encuentra en una gran parte de los dispositivos del mercado, con más de 11000 millones de chips distribuidos hasta la primera mitad de 2018 [16]. Se trata de una arquitectura más sencilla que las que vemos en otras arquitecturas rivales, como x86. Esto se debe a que tiene un repertorio de instrucciones reducido, limitando el tipo de operaciones que se pueden realizar; por ejemplo, no permite operar directamente con valores en memoria, sino que han de estar necesariamente en el banco de registros. Estas características permiten crear dispositivos muy pequeños, con un bajo consumo y con bajas necesidades de refrigeración, permitiéndole una alta penetración en el mercado de dispositivos portátiles y embebidos [17].

En este trabajo nos vamos a centrar en la versión v6 de la arquitectura, por ser la empleada en la configuración del puesto de laboratorio de las asignaturas de Estructura y Organización de Computadores. No obstante, las características indicadas a continuación aplican a la mayoría de las versiones de 32 bits posteriores.

2.1.1. Banco de registros

La arquitectura cuenta con un banco de registros de 32 bits. Posee 18 registros: 16 registros numerados (R0 a R15) y dos registros de manejo especial, con instrucciones separadas, denominados Current Program Status Register (CPSR) y Saved Program Status Register (SPSR). De los 16 registros numerados, a tres de ellos se los refiere por otra nomenclatura habitualmente debido a que tienen una función específica: el registro R13 es el *Stack Pointer* (SP), el registro R14 es el *Link Register* (LR) y el registro R15 es el *Program Counter* (PC). Algunos de estos registros se encuentran desdoblados en varios registros físicos, para algunos de los modos del procesador. En la Tabla 2 vemos en detalle el banco de registros, qué registros son privados para cada modo, y cuáles son compartidos entre los modos del procesador.

La funcionalidad de los registros es la siguiente:

- **Registros R0 a R12**: se utilizan como registros de propósito general.
- **Registro SP**: guarda la dirección del final del *stack* del modo de procesador al que pertenece. Según sea necesario se puede disminuir o incrementar el valor del registro SP para reservar o liberar (respectivamente) espacio en el stack, que es un segmento de la memoria del procesador que el programador maneja de forma dinámica.
- **Registro LR**: almacena la dirección a la que retornar tras la ejecución de una rutina o función. Cuando se produce una instrucción de salto y enlazado *("Branch and Link, BL"),* o cuando se produce una excepción este registro se actualiza para que apunte a la instrucción siguiente a la instrucción que se estaba ejecutando.
- **Registro PC**: almacena la dirección de la instrucción a ejecutar. Con cada ciclo del procesador, este registro se incrementa automáticamente de cuatro en cuatro (cada instrucción ocupa cuatro *bytes*), pero una instrucción de salto, una instrucción

aritmético/lógica que modifique este registro o una excepción pueden provocar que apunte a cualquier punto de la memoria. Esto puede dar lugar a posibles errores de ejecución.

Modes						
	Privileged modes					
		Exception modes				
User	**System**	**Supervisor**	**Abort**	**Undefined**	**Interrupt**	**Fast interrupt**
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Tabla 2 – Organización del banco de registros en los distintos modos del procesador. Los registros señalizados son los que son privados de ese modo. Imagen extraída de [17].

- **Registro CPSR**: almacena el estado de ejecución y el modo actual del procesador. Este registro consta de flags condicionales y bits de configuración. Las flags condicionales pueden actualizarse tras la ejecución de instrucciones aritmético/lógicas para reflejar ciertas características del resultado; por ejemplo, si el resultado de una operación ha sido cero. El valor de estas flags puede utilizarse para condicionar la ejecución de otras instrucciones posteriores. Los bits de configuración permiten modificar diversas características de la configuración del procesador; por ejemplo, el modo del procesador o las interrupciones habilitadas. Este registro no puede manipularse de forma directa, sino que es necesario emplear instrucciones especiales (MSR/MRS). Además, algunas partes del registro sólo pueden modificarse si se ejecuta desde un modo privilegiado.
- **Registro SPSR**: cuando se produce una excepción se guarda el contenido del registro CPSR del modo previo en este registro. De este modo se permite restaurar el contexto del código cuando se retorna del handler de una excepción. Este registro solo puede ser manipulado con instrucciones especiales (las mismas que para el CPSR), y solo se puede acceder al

correspondiente al modo actual, si éste tuviera uno. Este registro sólo es utilizado durante la gestión de excepciones, por lo que sólo los modos relativos a excepciones cuentan con un registro SPSR.

2.1.2. Excepciones

Se denomina excepción a una interrupción del flujo de ejecución del código generada por una fuente interna o externa para que el procesador pase a atender un evento. Las excepciones pueden ser causadas por eventos en el hardware o por instrucciones, como es el caso de las SWIs, las instrucciones no definidas o las instrucciones de punto de ruptura o *breakpoint* (BKPT).

La arquitectura ARM soporta siete tipos de excepciones, detallados en la Tabla 2. Las excepciones tienen asignadas un nivel de prioridad según su tipo, que determina el orden en el que se atenderán en el caso de que se produzcan varias simultáneamente. Priorizar las excepciones ligadas a eventos con limitaciones temporales permite evitar que se produzcan fallos que pudiesen afectar al comportamiento del sistema.

Las excepciones de tipo instrucción no definida e interrupción software comparten la misma prioridad debido a que no pueden ocurrir a la vez. Ambas son excepciones generadas por instrucciones, las excepciones de instrucción no definida se producen cuando la instrucción a ejecutar no es reconocida por el sistema y las interrupciones software se producen con la instrucción SWI. Si la instrucción no fuera reconocida por el sistema implicaría que no es una SWI, por lo que una instrucción no puede producir ambas excepciones.

Tipo de excepción	Modo	Prioridad
Reset	Supervisor	1 (Más alta)
Instrucción no definida	Undefined	7 (Más baja)
Interrupción Software (SWI)	Supervisor	7 (Más baja)
Prefetch Abort	Abort	6
Data Abort	Abort	2
IRQ (Interrupción)	IRQ	4
FIQ (Interrupción rápida)	FIQ	3

Tabla 3 – Tipos de excepciones [17].

Dentro de estas excepciones se encuentra el subconjunto de las interrupciones, que permiten avisar de un evento que requiere atención. Las interrupciones pueden ser de tres tipos: IRQ (interrupción por un evento hardware), FIQ (interrupción por un evento hardware de alta prioridad) y SWI (interrupción software).

De las siete excepciones solo tratamos con seis de ellas, ya que la excepción reset no se puede producir dentro del depurador. Estas seis excepciones se generan de las siguientes formas:

- **Instrucción no definida**: Se producen cuando se intenta ejecutar una instrucción con un formato no reconocido por el procesador. También se emplea en ocasiones de forma deliberada para extender o emular instrucciones de coprocesadores no disponibles en el dispositivo, pero dicho caso no se ha considerado en el diseño del depurador ni en este trabajo.
- **Interrupción software**: Se producen utilizando la instrucción SWI, para solicitar servicios del sistema operativo.

- **Prefetch abort**: Se producen cuando se intenta ejecutar una instrucción que no se encuentra en la cache. Adicionalmente se pueden producir utilizando la instrucción BKPT (Breakpoint) con fines de depuración.
- **Data abort**: Se producen cuando se intenta acceder a una dirección de memoria no válida o que no se encuentra en la cache.
- **IRQ**: Se producen por un dispositivo (externo o interno). Tiene menor prioridad que las FIQ. Están pensados para su uso con dispositivos de carácter lento (como un ratón o un teclado).
- **FIQ**: Se producen por un dispositivo (externo o interno). Tienen mayor prioridad que las IRQ y están pensadas para usarse en transferencias de datos o canales de procesado. Gracias a los registros privados de este modo no suele ser necesario salvar el estado de los registros, agilizando aún más el proceso.

Tanto las interrupciones IRQ como las FIQ están diseñadas para que pueden acontecer varias excepciones del mismo tipo a la vez. En caso de que eso ocurra no se dispone de ninguna política de prioridad que permita asignar mayor importancia a una interrupción que a otra (se hace encuesta al registro pending para saber cuál atender). Por ello todas las rutinas que requieran de su atención prioritaria han de programarse para que sean del tipo FIQ, que sí nos asegura mayor prioridad sobre las IRQ [17].

Modos de ejecución del procesador

Para la correcta gestión de las excepciones y los privilegios, el procesador cuenta con siete modos de ejecución.

De estos siete modos, cinco están destinados a la gestión de excepciones (los modos Supervisor, Abort, Undefined, IRQ y FIQ) como vemos detallado en la Tabla 3. Estos modos son privilegiados y cuentan con registros privados para poder ejecutar sus handlers sin afectar al contexto del modo llamador.

El modo User es el único modo no privilegiado, y es el modo en el que se debe ejecutar el código de usuario.

Finalmente tenemos el modo System, el único modo privilegiado no asociado a ninguna excepción. Comparte todos los registros con el modo User. Uno de sus usos es utiliza para que si se produce una excepción mientras se está atendiendo otra que utiliza el mismo modo del procesador se pueda preservar el contexto de la anterior excepción [18].

Tratamiento de las excepciones

Cuando se produce una excepción se para el flujo de ejecución del código y se salta a unas posiciones de memoria fijas, denominadas vectores de excepciones, que permiten el salto a la rutina correspondiente para cada excepción. Estas rutinas, a la que se denomina comúnmente *handler,* se encargan de determinar la causa de la excepción y de atenderla. Cuando se termina la ejecución del handler se reanuda el flujo de ejecución del código previo. Cuando se atiende una excepción el sistema cambia automáticamente de modo para poder preservar el estado del código interrumpido.

Como se ha indicado previamente, para provocar una excepción de tipo software interrupt se hace uso de las instrucciones SWI. Esta instrucción se utiliza como interfaz para acceder a uno de los servicios proporcionados por el sistema operativo, determinado por el identificador que la instrucción recibe como parámetro.

Al provocar esta excepción se pasa a modo supervisor y se salta al handler de las SWIs. Este handler identifica qué instrucción ha provocado la excepción (la dirección de la SWI es la instrucción anterior a la apuntada por el registro LR), lee el identificador de la rutina demandada y ejecuta el código correspondiente al servicio solicitado (por ejemplo, imprimir un carácter por pantalla o terminar la ejecución del código).

Otra instrucción que genera una excepción es la instrucción BKPT. La excepción producida es de tipo prefetch abort, que puede emplearse para detener la ejecución y poder comprobar el estado del programa, ayudando en la depuración del código. En caso de que el procesador incluya hardware de depuración, el breakpoint puede ser gestionado directamente por dicho hardware. Esta instrucción está diseñada para utilizarse en el proceso de depuración del código [17].

2.2. Depurador !UCDebug

El depurador !UCDebug es una herramienta de depuración de código ensamblador ARM enfocada al uso docente. Este software se distribuye como código libre bajo una licencia GPLv3, disponible públicamente en un repositorio de GitHub [19].

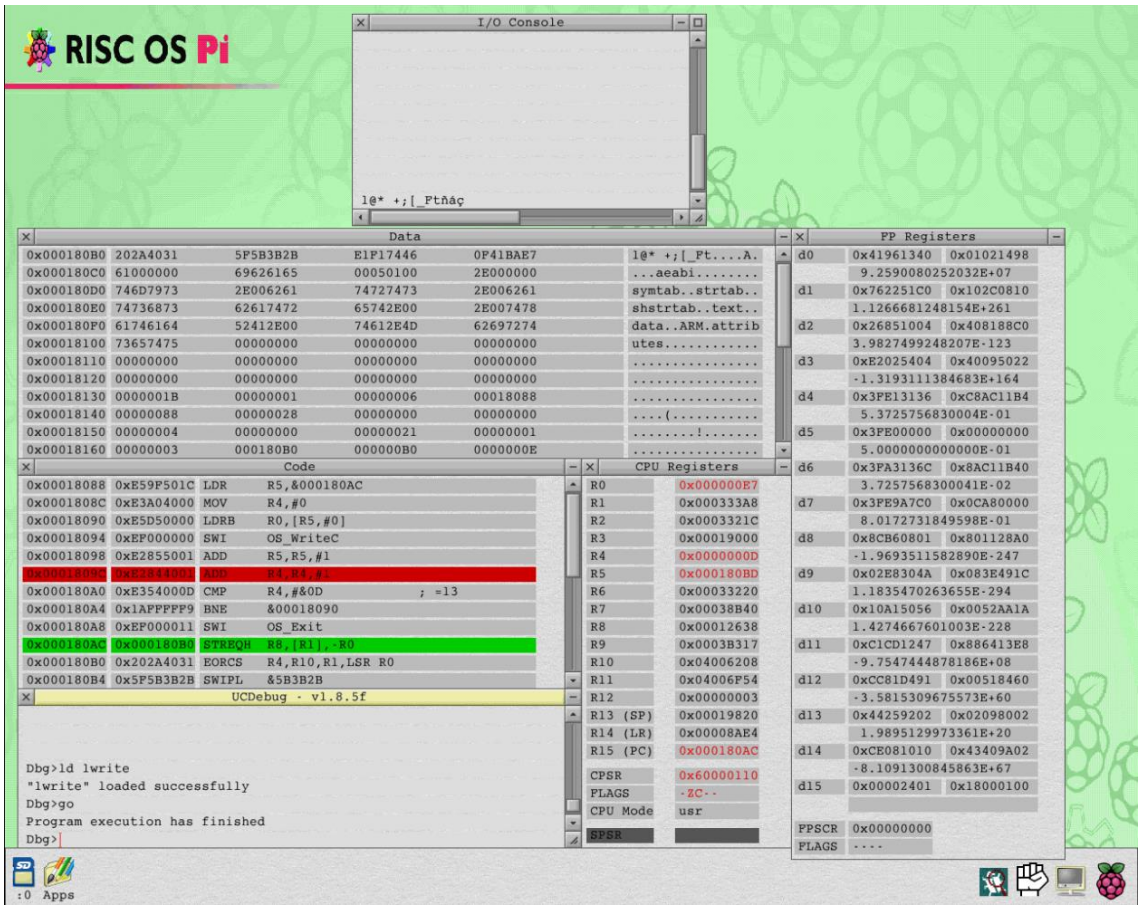

Figura 4 – Captura de la interfaz del depurador. Se muestran las seis ventanas detalladas. Abajo a la derecha se muestra el icono del depurador.

2.2.1. Apariencia

El depurador cuenta con una interfaz gráfica basada en ventanas. Se compone de cuatro ventanas principales, las cuales adoptan una configuración anidada por defecto, pero cada ventana puede ser reposicionada y minimizada (*iconizada*, en el argot de RISC OS) de forma independiente. Además, cuenta con dos ventanas adicionales que se muestran bajo demanda. En la Figura 4 se puede observar la interfaz del depurador con las seis ventanas desplegadas.

Las cuatro ventanas principales son las siguientes:

- **Ventana de comandos**: esta ventana se utiliza como consola de entrada de comandos para interactuar con el depurador. Además, en esta ventana recibiremos los mensajes relativos al estado de ejecución del depurador. Cualquier otro tipo de operación de entrada/salida con el programa en ejecución se realizará en la ventana de entrada/salida.
- **Ventana de datos**: muestra el contenido de la memoria. Está dividida en tres secciones. Empezando por la izquierda se muestran las direcciones de memoria que estamos viendo, indicando la dirección inicial de cada línea. La sección central nos muestra el contenido de la memoria en hexadecimal organizado en cuatro palabras por línea. La sección de más a la derecha nos vuelve a mostrar el contenido de la memoria, pero esta vez en formato ASCII. Al cargar un código, la ventana se reposiciona por defecto al comienzo del área de datos de dicho programa.
- **Ventana de código**: muestra el contenido de la memoria, interpretado como instrucciones. Cuando se carga un programa, el depurador automáticamente muestra al comienzo de la ventana la primera instrucción a ejecutar en el código cargado. las instrucciones del código cargado en el depurador. Se indica con un resaltado verde cuál es la siguiente instrucción que se va a ejecutar (la que esté alojada en la dirección señalada por el PC) y en un resaltado rojo las instrucciones en las que el usuario ha puesto un breakpoint. Esta ventana está dividida en tres columnas. Empezando por la izquierda, tenemos la dirección del código en la que se encuentra la instrucción. En la columna central se indica el código en hexadecimal de la instrucción en lenguaje máquina. En la columna de más a la derecha se muestra la instrucción en lenguaje ensamblador ARM, junto con sus operandos y el comentario asociado, si tuviera alguno.
- **Ventana de registros de la CPU**: muestra el contenido de los registros de propósito general del modo de ejecución actual, incluyendo el CPSR así como un desglose de algunos campos de este: el estado de las flags y el modo de ejecución de la CPU. Si el modo actual no tiene registro SPSR asociado, la fila correspondiente se muestra resaltada en gris oscuro. El contenido de los elementos se muestra en color rojo si tras la última ejecución de código su contenido ha sido modificado. La información de esta ventana se organiza en filas con formato nombre del elemento, información del elemento.

Las ventanas dos ventanas que se muestran bajo demanda son las siguientes:

- **Ventana de registros de la unidad de punto flotante (FPU)**: muestra el contenido de los registros de la unidad de punto flotante, incluyendo el FPSCR (registro de propósito especial similar al CPSR), y el estado de las flags. La información se organiza de la misma manera que en la ventana de registros de la CPU. Está ventana por defecto no se muestra al usuario, pero se puede abrir mediante el menú del depurador, teniendo que elegir si se quieren ver los registros de simple precisión o los de doble precisión (no se pueden visualizar ambos a la vez).

- **Ventana de entrada/salida**: esta ventana gestiona todas las operaciones de entrada/salida de caracteres producidas por el código del usuario. Por defecto esta ventana no se muestra, pero se abre automáticamente cuando se produce una operación de entrada/salida en el código.

Uso del depurador

El depurador puede interactuar con el depurador mediante diversos comandos introducidos por la ventana de consola, y mediante los botones del ratón sobre las distintas ventanas y el icono de la aplicación.

Las ventanas de código y memoria se pueden navegar mediante el scroll con la rueda de ratón combina con el clic izquierdo y derecho, que permite mover hacía arriba o abajo la sección de código/memoria mostrada en la ventana.

En la ventana de código al hacer clic con la rueda de ratón cambiará cómo se agrupa la información del contenido en hexadecimal (por palabras, medias palabras o bytes).

El depurador proporciona múltiples funciones adicionales al interactuar con el icono del depurador en la barra de aplicaciones. Haciendo clic con la rueda del ratón se mostrará el menú del depurador, que nos permite reiniciar la posición de las ventanas del depurador, mostrar la ventana de registros de punto flotante, mostrar información del depurador o salir. Haciendo clic izquierdo en el icono provoca que se ponga en foco la ventana de comandos del depurador. Haciendo clic derecho dos veces se provoca que la ejecución del código se detenga.

El depurador se maneja principalmente a través de los siguientes comandos de texto:

- **about**: muestra un mensaje con información sobre el depurador junto a una lista de los autores y contribuidores.
- **help**: muestra una lista con los comandos disponibles y su sintaxis.
- **ld nombre_de_fichero**: nos permite cargar un programa en el depurador dado el nombre del ejecutable. Reinicia el estado del depurador y se coloca como siguiente instrucción a ejecutar el punto de entrada del código.
- **stop:** termina la ejecución del programa actual.
- **clr**: elimina todos los breakpoints de usuario.
- **br [dirección]**: nos permite poner un breakpoint de usuario en la dirección que se le pasa como parámetro. Si no se pasa dirección muestra una lista con todos los breakpoints de usuario.
- **set [s] dirección =valor**: reemplaza *s* bits de contenido de la memoria en la dirección recibida, por el valor recibido. Si no se pasa un parámetro *s*, se reemplaza una palabra.
- **reg nombre=valor**: este comando recibe por parámetro el nombre de un registro y un valor. La función de este comando es reemplazar el contenido de ese registro por el del valor recibido.
- **mem [dirección]:** reinicia la ventana de datos para que empiece en la dirección pasada como parámetro. Si no recibe ningún parámetro, se reinicia a la dirección del comienzo de la sección de código.
- **code [dirección]**: reinicia la ventana de código para que empiece en la dirección pasada como parámetro. Si no recibe ningún parámetro, se reinicia a la dirección del punto de entrada del código.
- **Comandos de ejecución del código**: para la ejecución del código se utiliza una serie de comandos muy similares. Hay cinco comandos, asociados a cada modo de ejecución del

depurador: **go** (modo Go), **tr** (modo Traza), **gd** (modo Go-Direct), **gt** (modo Go-To) y **gf** (modo Go-Fast). Los comandos go, tr, gd y gf pueden recibir una dirección como parámetro opcionalmente. Su comportamiento es la de empezar a ejecutar el código en su modo correspondiente desde la instrucción actual, o desde la dirección recibida como parámetro (si hubiera). La instrucción gt recibe una dirección como parámetro obligatorio y empieza a ejecutar en modo GT desde la instrucción actual.

Para el apartado de depuración, el usuario puede introducir breakpoints mediante la interfaz gráfica. Una vez cargado el código a depurar haciendo clic con la rueda del ratón en una instrucción de la ventana de código introducirá un nuevo breakpoint, o lo quitará si ya estaba puesto (se resaltan en rojo las instrucciones con un breakpoint de usuario).

La información que devuelve al usuario hay veces que está limitada u ofuscada deliberadamente, mostrando un mensaje menos completo de lo que sería posible con las comprobaciones que realiza el propio depurador. Esto se debe a que el depurador tiene un enfoque docente, y como parte del aprendizaje de los lenguajes ensamblador se busca que el alumno sea capaz de encontrar la causa de los fallos por sí mismo. Por ejemplo, ante un fallo de acceso a memoria (excepción de tipo Data Abort), sólo se indica el tipo de fallo y la dirección de la instrucción que lo ha causado, omitiendo si se debe a un problema de alineamiento, a una dirección no existente, o a una falta de privilegios de acceso.

El depurador ofrece cinco modos de ejecución del código del usuario:

- **Modo Go**

El modo Go del depurador busca ser el modo de ejecución principal a la hora de desarrollar código, ya que ofrece una mayor seguridad y control a la hora de ejecutar el código del usuario. Su funcionalidad consiste en hacer una ejecución completa del código del usuario hasta llegar a un breakpoint puesto por el usuario, o hasta que se finalice la ejecución.

Para asegurar un alto nivel de seguridad y robustez, cada instrucción del código del usuario se sustituye por una instrucción BKPT 1, guardando previamente la información relativa a la instrucción sustituida. Esto provoca que antes de ejecutar cualquier instrucción se tenga que pasar por el núcleo del depurador. Al volver al núcleo del depurador se restauran los handlers nativos de RISC OS y se habilitan todas las interrupciones, lo cual permite al depurador gestionar errores que nuestros handlers no contemplan y ejecutar código en condiciones más similares a las nativa del sistema.

- **Modo traza**

En este modo se ejecuta una única instrucción y se devuelve el control al usuario. Cualquier excepción que se produzca a razón de la instrucción, o que estuviera pendiente de atender se atenderá antes de devolverle el control al usuario. Este modo está pensado para poder depurar de forma muy fina el funcionamiento del código.

- **Modo Go-Direct**

Este modo tiene el mismo funcionamiento que el modo Go, pero no para la ejecución cuando llega a un breakpoint del usuario, sino que continua con la ejecución del código hasta llegar al final del programa o producirse un error o interacción con el usuario.

- **Modo Go-To**

Este modo tiene el mismo funcionamiento que el modo Go-Direct, pero se detiene si la ejecución llega a la instrucción cuya dirección se pasa como parámetro al comando (gt addr).

- **Modo Go-Fast**

Este modo, de forma similar al modo Go, nos permite ejecutar un código sin que se detenga hasta que se llegue a un breakpoint de usuario o termine el programa. En este modo se prima la velocidad de ejecución en detrimento del control sobre la ejecución.

2.2.2. Estructura

El código del depurador está organizado en dos partes muy diferentes. La primera, relativa a la interfaz de ventanas, está escrita en lenguaje C, y maneja únicamente aspectos visuales del depurador. La segunda parte es el núcleo central del depurador, escrito en lenguaje ensamblador ARM, que maneja el control de la ejecución del código del usuario en el procesador, e interactúa con la interfaz de ventanas.

En el depurador se pueden dar tres tipos de breakpoints: los colocados por el usuario a través del depurador, las instrucciones BKPT de control introducidos por el propio depurador para controlar la ejecución del código y las instrucciones BKPT que pueda haber originalmente en el código. Cabe destacar que este último caso producirá que la ejecución se termine por una excepción prefetch abort, la cual se notificará como un error al usuario.

Una vez el depurador comienza a ejecutar el código del usuario, la única forma de volver a retomar el control en la ejecución es cuando se produce una excepción. Esto ocurre porque RISC OS es un sistema operativo con multitarea cooperativa, que requiere que los programas devuelvan voluntariamente el control de la ejecución al sistema, sin que puedan ser expulsados de forma forzosa. El depurador puede requerir volver al núcleo del depurador en puntos arbitrarios del código, y además realizarlo de forma transparente al usuario, sin notificárselo. Para ello se decide utilizar la instrucción BKPT 1, que produce una excepción de tipo prefetch abort. El depurador sustituye las instrucciones del código del usuario por esta instrucción siempre que necesita volver al núcleo del depurador, preservando antes la información de la instrucción sustituida para su futura restauración. Para que el usuario no sea consciente de este proceso, el depurador se encarga de tratarlas antes de que lleguen a ejecutarse. No se contempla que el código del usuario contenga instrucciones BKPT, ya que para detener la ejecución dispone de los breakpoints de usuario del propio depurador.

2.2.2.1. Handlers

El depurador cuenta con handlers propios para gestionar parte de las excepciones del sistema. Esto se hace en una primera instancia para poder manejar las excepciones prefetch abort provocadas por la instrucción BKPT 1. Como se ha detallado en esta sección, estas instrucciones se introducen en el código del usuario por el depurador para que se retorne al núcleo del depurador en los puntos necesarios, para lo que necesitamos que las excepciones prefetch abort producidas por estas instrucción no se notifiquen al usuario y se continúe con la ejecución. Para ello, se desarrolló un handler para las excepciones prefetch abort encargado de esa gestión.

Más adelante se decidió implementar el resto de los handlers para que el depurador pueda gestionar los errores de forma distinta a como se hace en nativo. Esto se debe a que si se produce un error con los handlers nativos de RISC OS el sistema asociaría el error con la aplicación del propio depurador, por lo que éste se cerraría. Además, nos permite mostrar mensajes de error

personalizado según el error. Si los errores se gestionasen con los propios handlers de RISC OS el sistema nos mostraría una ventana de error como vemos en la Figura 5:

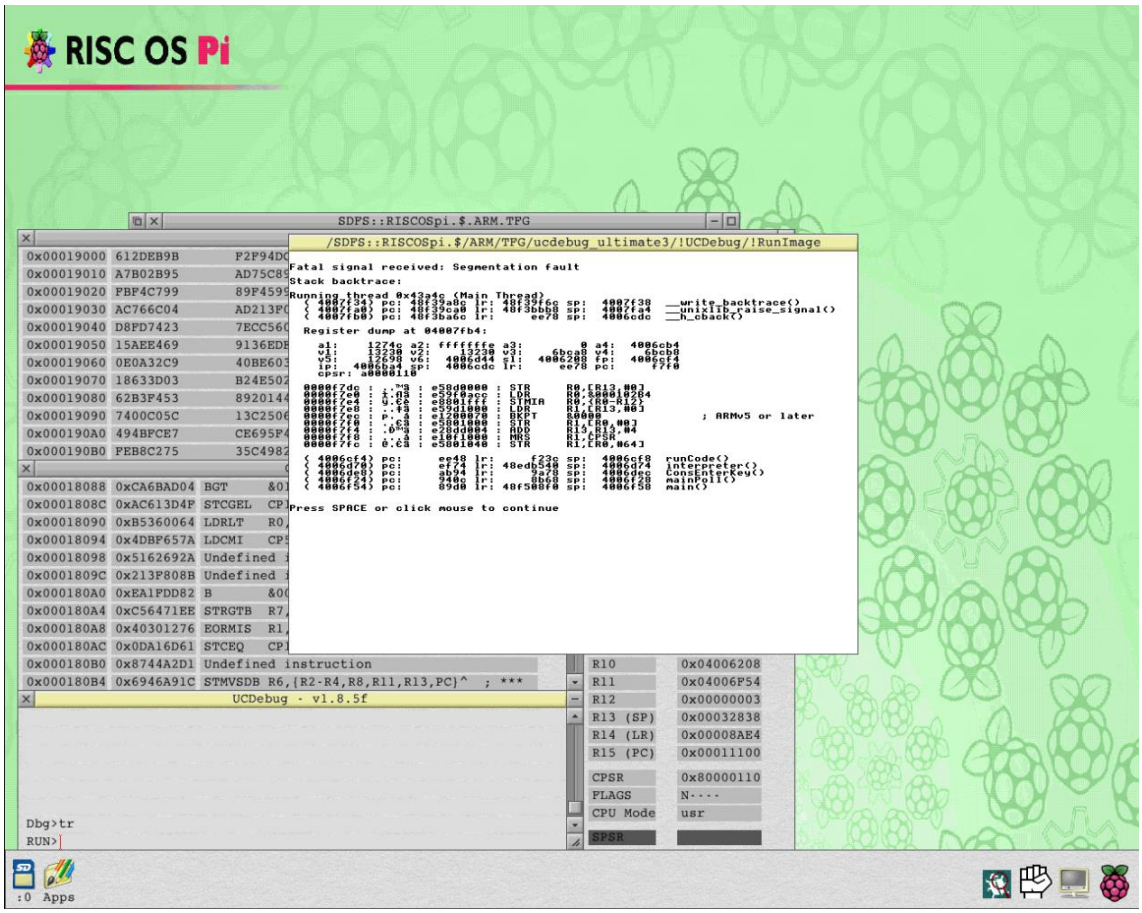

Figura 5 – Captura del depurador en la que se muestra la ventana de error tras producirse una excepción con los handlers nativos de RISC OS.

El mensaje de error le presenta al usuario demasiada información, mostrada de forma poco clara y de la cual gran parte no le es útil, ya que es información relativa a la implementación del depurador. Otro problema es que una vez que aparezca la ventana de error la siguiente interacción del usuario con el sistema provocará el cierre de la ventana de error y del depurador. Esto implica que el usuario no podrá utilizar las funciones del depurador, dificultando en gran medida depurar el error. Además, la ventana de error siempre aparece en el centro de la pantalla, por lo que es muy probable que opaque información sobre el estado del depurador.

En cambio, tratar el error mediante los handlers del depurador evita estos problemas. En vez de cerrarse, el depurador detiene la ejecución del código del usuario, permitiendo así analizar el estado en que se produjo el error y utilizar las funciones del depurador. Como vemos en la Figura 6, el mensaje de error se muestra a través de la ventana de comandos de forma concisa y con la información realmente útil para el usuario.

Para proporcionar al usuario estas funcionalidades, el depurador cuenta con cinco handlers propios que sustituyen los handlers nativos de RISC OS durante la ejecución del código del usuario. Cada vez que se retorna al núcleo del depurador se restauran los handlers nativos de RISC OS, y cuando se va a saltar al código del usuario se vuelven a poner los del depurador.

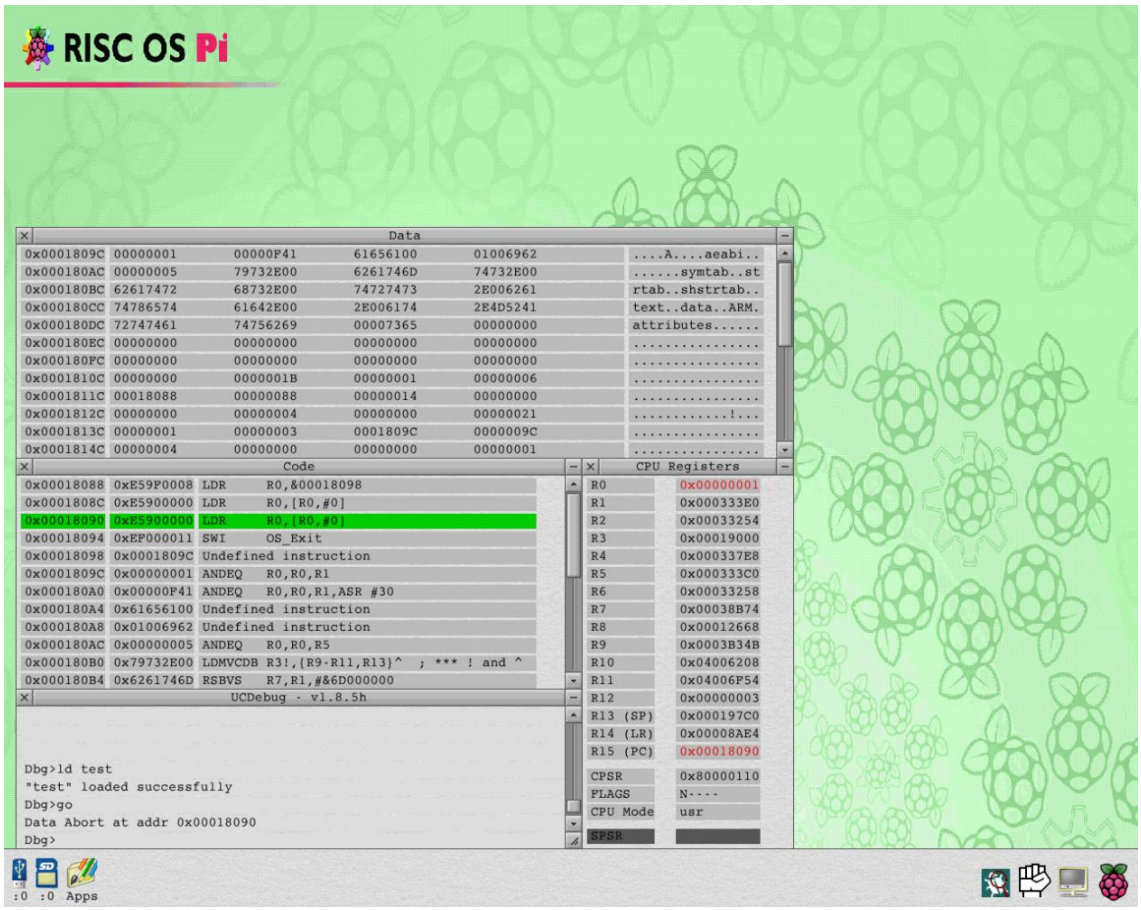

Figura 6 – Captura del depurador tras haberse producido una excepción de tipo Data Abort con los handlers del depurador.

Antes del desarrollo de este proyecto el depurador contaba con cuatro handlers, a los que se ha añadido un handler nuevo para atender las excepciones por SWI (interrupción software). El funcionamiento de los cuatro handlers originales es el siguiente:

- **Handler de excepciones por prefetch abort**: Una excepción prefetch abort se puede producir debido a una instrucción breakpoint o porque la instrucción que se iba a ejecutar no está en cache.

  Lo primero que hace el handler es identificar qué instrucción ha producido el prefetch abort. Si la instrucción es un breakpoint vuelve directamente al núcleo del depurador, donde determinará si proseguir con otra instrucción o retornar a la interfaz gráfica, dependiendo del modo de ejecución del depurador que se esté usando.

  Si no se trata de una instrucción BKPT introducida por el propio depurador, entonces el error se produjo por un error de prefetch en la dirección de la instrucción. Este error se gestiona retornando al núcleo del depurador con un error de prefetch abort, lo cual producirá que se detenga la ejecución y se notifique al usuario.

  Que se produzca un error de prefetching sin una instrucción breakpoint no es el comportamiento esperado que ocurra cuando se accede a este handler. Para evitar que estos errores sucedan se hace un prefetch de cada instrucción que se va a ejecutar, y hay una comprobación para que la ejecución no se salga del bloque de memoria asignado al

usuario. Este mecanismo no aplica a llamadas al sistema u otras funcionalidades previstas del depurador, y se mantiene únicamente como elemento de salvaguarda en caso de que algún error provoque esta excepción.

- **Handler de excepciones por instrucción no definida**: Una excepción por instrucción no definida se produce cuando se intenta ejecutar una instrucción no reconocida por el sistema. Esto ocurre habitualmente cuando se intenta ejecutar una dirección que no es parte del segmento de código del usuario, por ejemplo, si se intenta ejecutar parte del área de datos.

  El comportamiento del handler únicamente retorna al núcleo del depurador notificando de un error por instrucción no definida, provocando que se muestre al usuario un mensaje de aviso a través de la ventana de consola.

- **Handler de excepciones por data abort**: Identifica si la excepción ha sido provocada por un fallo de página o por otro motivo, como un acceso a una dirección de memoria ilegal. Si fue por un fallo de página se comprueba si la dirección a la que se quería acceder es válida.

  Si es válida, se retorna al núcleo del depurador y se hace prefetch de la dirección que produjo el fallo de página y se vuelve a ejecutar la instrucción de lectura/escritura que produjo la excepción. Esto se hace para emular el comportamiento del handler de Data Abort nativo de RISC OS, el cual trae a la cache los bloques de memoria que detecta que no están en la misma. Aquí lo que hacemos es detectar dicho caso en el handler del depurador, pero le requerimos al handler nativo que haga la operación correspondiente de traer el bloque de memoria una vez regresamos al núcleo del depurador (cuando volvemos al núcleo del depurador siempre se restauran los handlers nativos de RISC OS).

  Si la dirección no es válida, o si la excepción no fue provocada por un fallo de página, se retorna al núcleo del depurador notificando un error de data abort.
- **Handler de excepciones de IRQ**: Es llamado cuando se produce una de las interrupciones programadas por el código de usuario. El handler se encarga de identificar qué dispositivo ha producido la interrupción, saltar a la correspondiente rutina de interrupción dentro del código del usuario, con el procesador en modo IRQ y de volver de la rutina de atención recuperando el estado del usuario previo a la interrupción.

El comportamiento del depurador es el mostrado en la Figura 7. A través de la GUI, el usuario puede cargar código del usuario e iniciar la ejecución del mismo. Al lanzar la ejecución, el depurador guarda el estado del sistema y comprueba la instrucción a ejecutar, en caso de que se trate de una SWI cuya ejecución deba capturar el depurador; este comportamiento se ha modificado en este trabajo, como se detallará posteriormente en la Sección 3.2. Posteriormente se identifica la instrucción que sucederá a la que toca ejecutar, y se reemplaza en el código por la instrucción BKPT, guardando previamente, guardando previamente la instrucción original para luego restaurarla. Se hace un prefetch de la dirección de memoria en que está la instrucción a ejecutar para evitar fallos de prefetching. Se sustituyen los handlers nativos de RISC OS por los propios del depurador, y se carga en los registros del procesador el estado del sistema cuando se ejecutó por última vez el código del usuario. De ahí se salta a la instrucción en cuestión, para poder ejecutarla. En ese punto se debe producir alguna de las excepciones explicadas previamente, devolviendo el control de la ejecución al núcleo del depurador al saltar a uno de sus handlers.

Tras regresar del handler y restaurar los handlers originales de RISC OS, se verifica si la excepción se debió a un error, en cuyo caso se regresa a la interfaz gráfica para mostrar un mensaje de aviso. En caso de ejecución sin fallos, se comprueba el modo de ejecución del depurador empleado, y se regresa a la interfaz o se mantiene en el núcleo del depurador, para seguir ejecutando código del usuario.

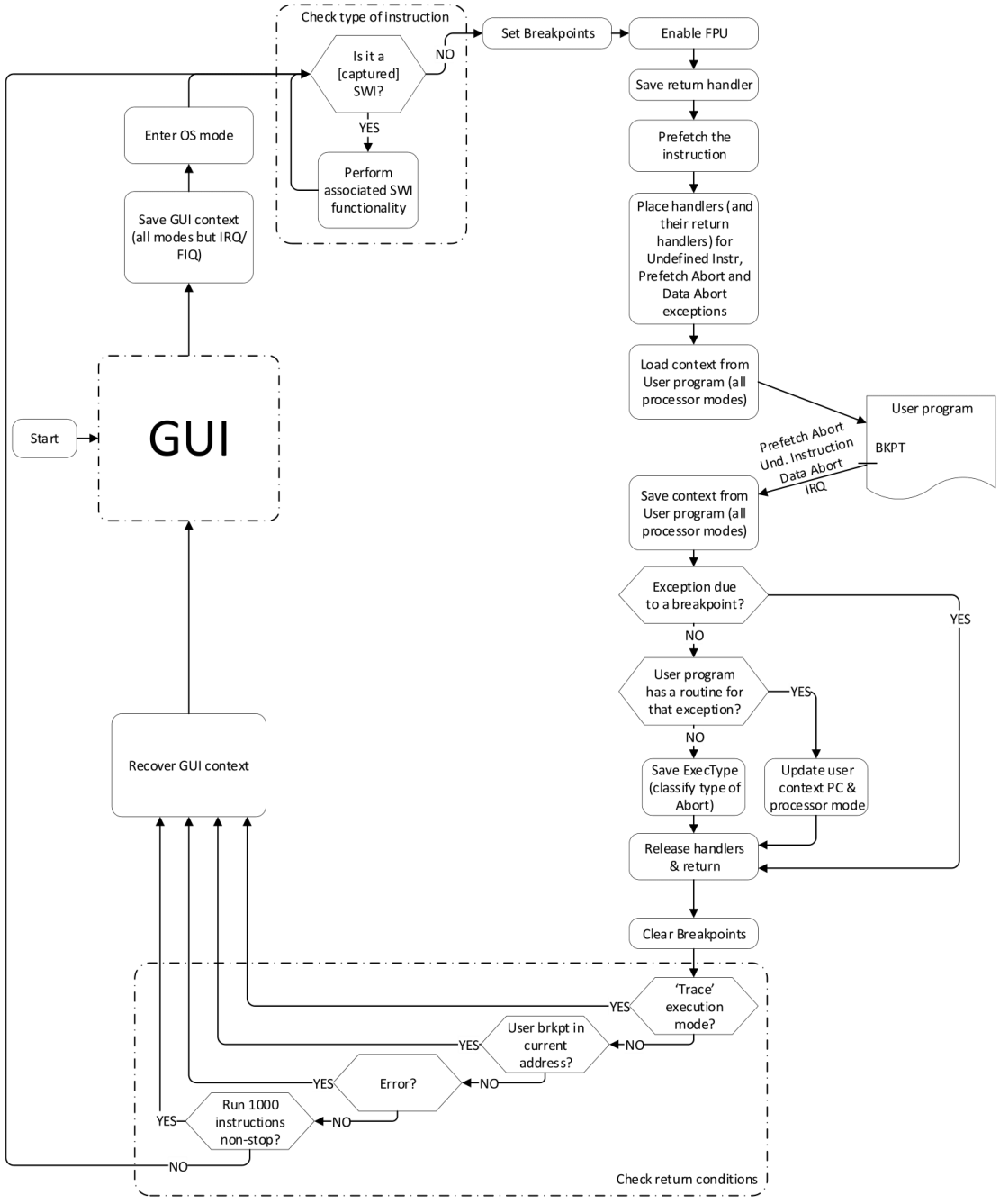

Figura 7 – Esquema de comportamiento del depurador !UCDebug.

2.2.2.2. SWIs

Del grueso de SWIs que RISC OS proporciona hay un conjunto de estás que no podemos utilizar directamente a través del handler de las SWIs nativo de RISC OS. Esto es debido a que necesitamos poder controlar su funcionamiento para que se muestre correctamente su

información en el depurador (como es el caso de las SWIs que imprimen caracteres por consola), para poder suministrarle el input necesario al usuario (como es el caso de las SWIs que requieren que el usuario teclee caracteres) o porque necesitamos adaptar su funcionamiento completamente (por ejemplo, la SWI OS_Exit terminaría la ejecución del depurador, pero solo nos interesa que termine la ejecución del código del usuario).

Aunque pueden existir otras SWIs que necesitarían ser capturadas para que funcionen correctamente en el depurador, durante el desarrollo inicial del depurador se decidió capturar únicamente las más necesarias para otorgar al usuario todas las funciones que vaya a necesitar para desarrollar códigos del tipo planteado en las prácticas de las asignaturas. Así, por ejemplo, de todas las SWIs que imprimen por pantalla hemos decidido capturar la que nos permite imprimir un carácter alojado en un registro y la que nos permite imprimir una cadena de caracteres alojada en memoria, ya que con esas dos se cubren los casos de escritura por pantalla más habituales.

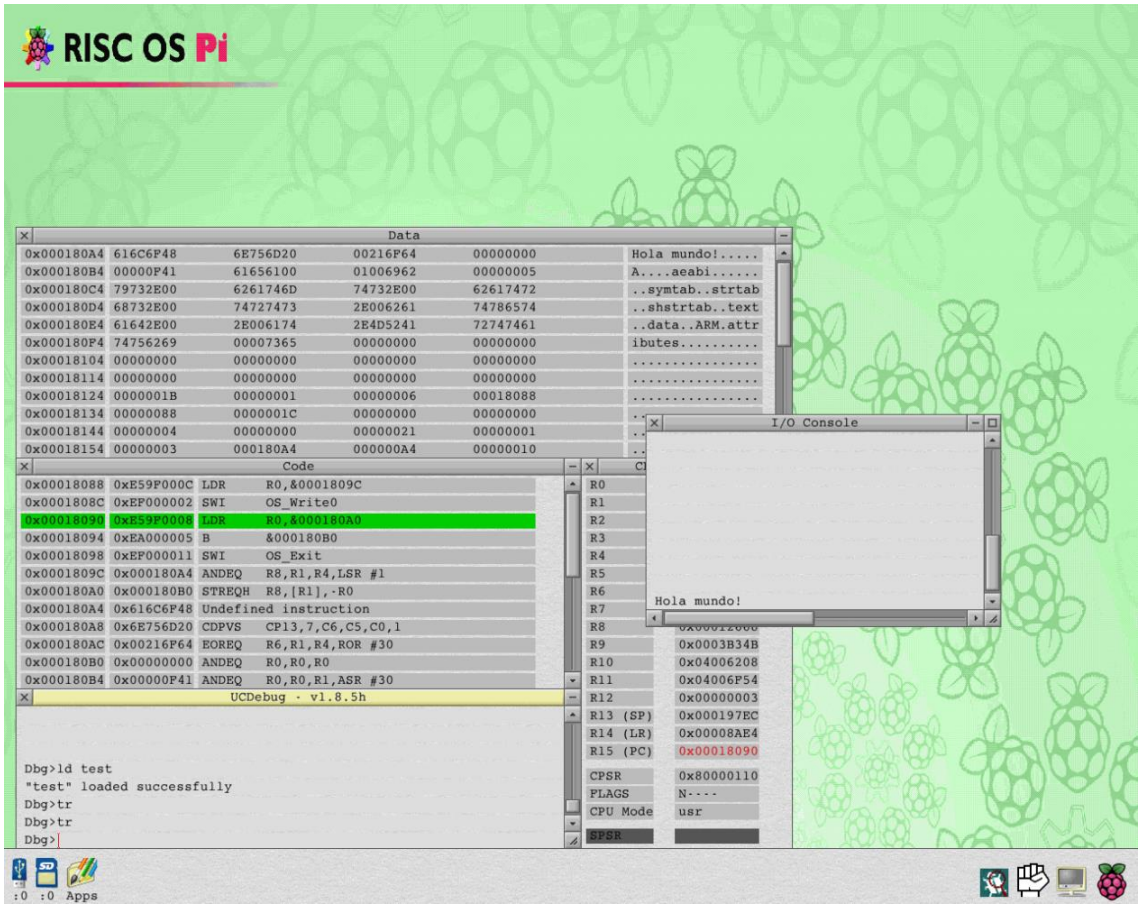

Figura 8 – Captura del depurador tras ejecutar un código que imprime por consola. La ventana de entrada/salida se muestra en la parte derecha bajo el nombre *I/O Console.*

Las SWIs que hacen lectura o escritura de caracteres deben ser capturadas por el depurador ya que el depurador utiliza una ventana propia para estas operaciones. Esta ventana del depurador funciona como una consola donde se producen las operaciones de entrada/salida relativas al código del usuario en ejecución. Esta ventana requiere tareas extra de gestión que no le podemos encargar a RISC OS, como que si ejecutamos una SWI que la utilice y la ventana no está

abierta actualmente, se debe abrir. Además, se requiere indicarle a RISC OS cuáles son los streams de entrada y salida que vamos a usar durante la ejecución de estas SWIs. Esta ventana podemos observarla en la Figura 8.

Las SWIs de gestión de rutinas de atención han de ser capturadas ya que necesitamos un control preciso de las interrupciones para que no se produzcan mientras estamos en el núcleo del depurador. Durante la ejecución del código se pueden producir interrupciones externas al depurador, la cuales debemos ignorar, pero también hay interrupciones que debemos atender cada cierto tiempo para que el sistema funciones. Además, nos interesa saltar a la rutina de atención de forma controlada, ya que nos permite depurar la propia rutina, que de otra forma se trataría como una caja negra.

El conjunto completo de SWIS capturadas por el depurador son las siguientes:

- **OS_Exit**: Retorna al núcleo del depurador indicándole que se ha terminado la ejecución del programa. El depurador se encargará de notificárselo al usuario y parar la ejecución del programa. Actualmente no recibe parámetros, pero en futuro podría actualizarse para que soporte la salida con errores.

  Esta SWI es capturada ya que si la ejecutáramos con los handlers nativos lo que haría sería terminar la ejecución del depurador, cerrándolo en el proceso, cuando sólo queremos que se pare la ejecución del código del usuario.

- **OS_WriteC**: Retorna al núcleo del depurador indicándole que se ha producido una SWI OS_WriteC. El depurador se encargará de imprimir por pantalla el carácter almacenado en el registro R0.
- **OS_Write0**: Retorna al núcleo del depurador indicándole que se ha producido una SWI OS_Write0. El depurador se encargará de imprimir por pantalla la cadena de caracteres a la que apunta R0. Además debe de actualizar el puntero en R0 para que apunte al carácter siguiente al carácter nulo que indica el fin de la cadena de caracteres impresa.
- **OS_ReadC**: Retorna al núcleo del depurador indicándole que se ha producido una SWI OS_ReadC. El depurador se encargará de leer el carácter del usuario y almacenarlo en R0.
- **OS_ClaimDeviceVector**: Asocia una rutina de atención a una de las fuentes de interrupción de los dispositivos (IRQ).

  El sistema cuenta con una tabla de vector de interrupciones. Cada uno de estos vectores almacena la información correspondiente a las rutinas de atención, a qué dispositivo atienden y qué valor se le debe pasar en R12. Cuando utilizamos esta SWI añadimos una nueva entrada a esa tabla, siempre que los parámetros utilizados correspondan a un número de dispositivo y rutina de atención válidos. Además, el depurador debe comprobar que la fuente de interrupciones no se encuentre enmascarada en el controlador de interrupciones.

  Para que el depurador pueda habilitar las interrupciones del usuario al saltar a su código, esta SWI marca qué fuentes de interrupciones está usando el usuario.

  Si no consigue asignar la rutina de atención, retorna al núcleo del depurador con un error indicando que el dispositivo seleccionado no está disponible.

  Esta SWI recibe en el R0 el número de dispositivo, en R1 la dirección de la rutina de atención y en R2 el valor que se pasa al registro R12 cuando se llama a la rutina.

- **OS_ReleaseDeviceVector**: Elimina una rutina de atención a una de las fuentes de interrupción de los dispositivos (IRQ) y reinstala la anterior si existiera. Si no existiera se desactivan las interrupciones de esa IRQ. Los parámetros recibidos en los registros R0 a R2 deben ser los mismo que se utilizaron en la SWI OS_ClaimDeviceVector, ya que sirven de identificador para el vector de interrupciones que queremos eliminar.

3. Modo Go-Fast

En este capítulo se explica el replanteamiento del modo Go-Fast, partiendo de cómo era al comienza del proyecto, hasta llegar a la versión actual. Se detalla el enfoque tomado y el funcionamiento de las dos piezas principales desarrolladas para la implementación del modo. Para concluir, hay una sección de evaluación en la que se comentan los resultados obtenidos y se comparan con las otras formas de ejecutar código disponibles en el equipo.

3.1. Introducción

Este desarrollo partía de un prototipo del modo Go-Fast el cual no era funcional. Este prototipo presentaba algunos elementos útiles para el desarrollo de la nueva implementación.

El planteamiento original era interrumpir la ejecución del código del usuario únicamente en los puntos en que es absolutamente necesario volver a la interfaz del depurador: las instrucciones SWI y los breakpoints establecidos por el usuario. Para asegurarse de que se vuelva al núcleo del depurador (y de ahí a la interfaz) se sustituían esas instrucciones en el código del usuario por instrucciones BKPT.

Esas instrucciones sustituidas deben conservarse para su ejecución tras volver al núcleo del depurador por el breakpoint. Para ello se utilizaba una estructura de bloques que contienen la información necesaria para restaurar las instrucciones sustituidas (la dirección de memoria de la instrucción, y el código máquina de la instrucción original del código). Como el número de SWIs que el usuario puede introducir en su código es indeterminado, el espacio de memoria reservado para esta estructura debía poder almacenar muchas instrucciones. Esto supone un problema, ya que la memoria disponible nos limita en este aspecto.

Además, tener que tratar las SWIs supone ralentizar el modo Go-Fast. En primer lugar, el depurador debe hacer un análisis completo del código cada vez que queramos ejecutarlo, para detectar las SWIs en el código y sustituirlas. Por otra parte, cada vez que volvamos al depurador debemos comprobar si la siguiente instrucción a ejecutar en el código del usuario es una de las SWIs capturadas por el depurador.

En el nuevo planteamiento del modo Go-Fast se decidió eliminar la sustitución de SWIs por instrucciones BKPT en favor del desarrollo de un nuevo handler para las excepciones por SWIs (los breakpoints introducidos por el usuario seguimos gestionándolos como en la implementación original). Esto reduce la duración del procesamiento inicial, ya que no necesitamos analizar el código para localizar las SWIs, y el problema de tener que guardar información de un número de instrucciones no especificado, ya que el número de breakpoints de usuario es limitado (actualmente el máximo son 256 breakpoints). Los únicos puntos que ralentizan la ejecución son cuando se ejecuta una SWI, que al gestionarse con el nuevo handler es mucho más eficiente que la anterior implementación con breakpoints, y en los propios breakpoints del usuario, donde el impacto sobre el tiempo de ejecución del código no es crítico porque se va a detener el flujo de ejecución. Además, esta implementación del handler de las SWIs beneficia al resto de modos, ya que evita que el núcleo del depurador tenga que comprobar si la siguiente instrucción a ejecutar es una de las SWIs capturadas.

Un problema adicional del modo Go-Fast es que el usuario, una vez inicia la ejecución del código, no puede interactuar con el sistema hasta que se llegue a un breakpoint de usuario, el programa termine, o se produzca una SWI que requiera input del usuario. Para corregir esta limitación se introduce un mecanismo de interrupciones que periódicamente hace que el núcleo del depurador retome el control de la ejecución, retornando desde ahí a la interfaz.

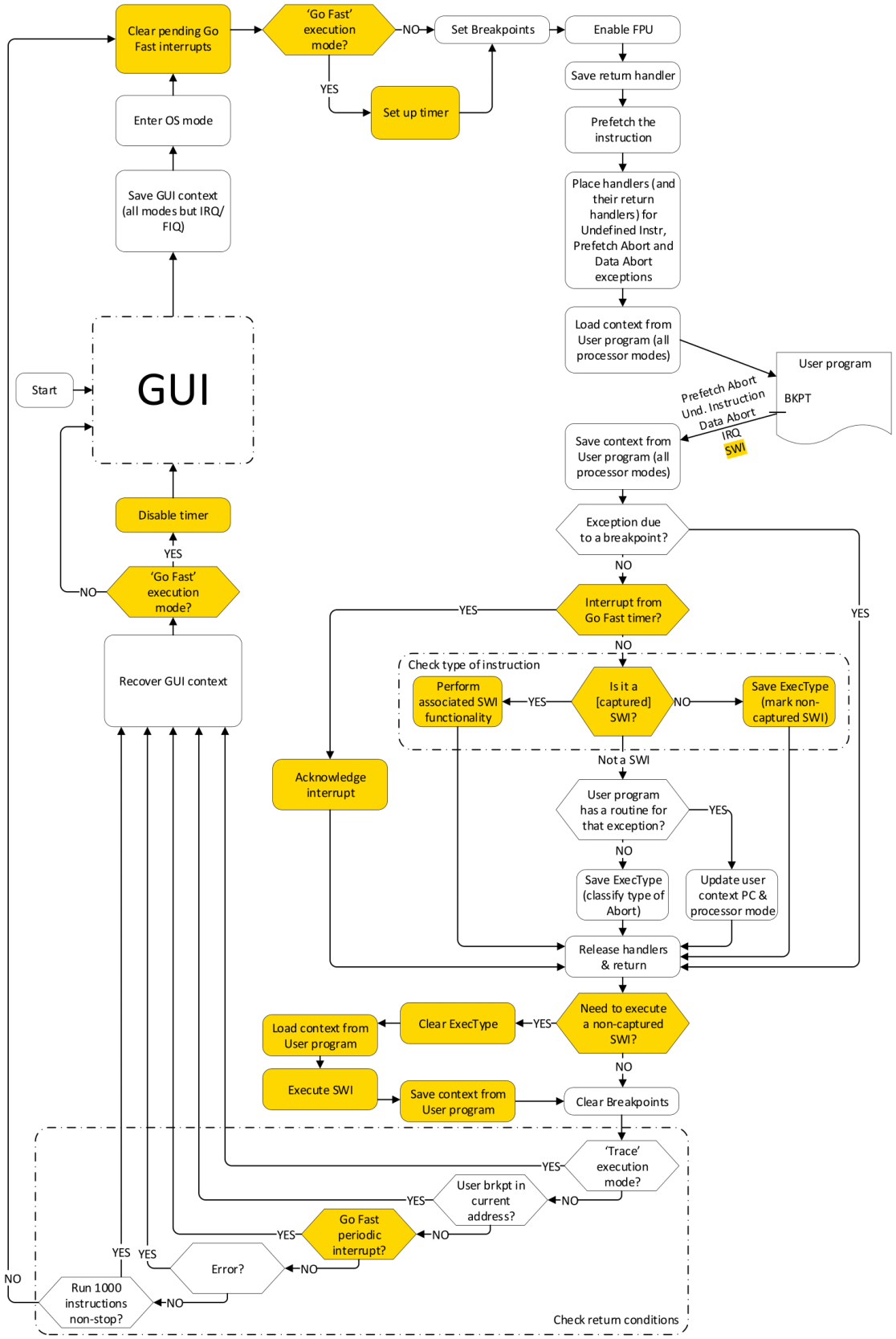

Figura 9 – Esquema de comportamiento del depurador !UCDebug tras añadir el modo Go-Fast. Se ha resaltado en color amarillo los bloques añadidos respecto al esquema original.

En la Figura 9 podemos observar el mismo esquema de comportamiento del depurador tras introducir el modo Go-Fast. Se resalta en amarillo las partes nuevas que se han añadido con respecto al de la Figura 7.

3.2. Handler para las excepciones SWI

Como pieza clave del modo Go-Fast se desarrolla este handler para las excepcione SWI. Este handler es llamado por cualquiera de las instrucciones SWIs que haya en el código del usuario, ocupándose de la gestión de las SWIs capturadas que antes realizaba el núcleo del depurador, detallado en la Sección 2.2.2.2. Las SWIs no capturadas son marcadas para ejecutarlas posteriormente desde el núcleo del depurador, tras haber restaurado los handlers nativos de RISC OS. Esto último es necesario porque, al ejecutar el handler nativo de RISC OS para atender dicha SWI, se pueden producir fallos de prefetch e incluso llamadas a otras SWIs del sistema, que producirían un comportamiento inestable si fuesen atendidas a través los handlers del depurador.

3.2.1. SWIs capturadas

En el caso de que la SWI sí sea capturada, el handler del depurador realiza la funcionalidad asociada, siguiendo la descripción que se dio en la Sección 2.2.2.2.

Todas las SWIs que estaban capturadas previamente es necesario que sigan recibiendo un tratamiento específico por las razones detalladas (necesidad de interactuar con la interfaz de ventanas, manejo dedicado de interrupciones en el código del alumno). Además, se hace necesario capturar dos instrucciones SWI adicionales, relativas a la entrada/salida del modo supervisor en la CPU.

Las SWIs para entrar y salir del modo supervisor han de ser capturadas, para evitar que se ejecuten dentro del núcleo del depurador, en la función que se ocupa de las SWIs no capturadas, la cual por defecto ya está un modo privilegiado. Esto nos obliga a realizar el cambio de modo con un implementación más “manual”:

- **OS_EnterOS**: Cambia el contexto del usuario para que pase a modo Supervisor. Guarda en el SPSR del modo Supervisor el CPSR actual, actualiza el CPSR para que pase a modo Supervisor y cambia el LR del modo Supervisor para que apunte al anterior PC.
- **OS_LeaveOS**: Cambia el contexto del usuario para que pase a modo User. Modifica el CPSR para que se cambie a modo User.

3.2.2. SWIs no capturadas

En el caso de que el handler de las SWIs detecte que es una de las SWIs no capturadas lo que hará es marcarlo y volver al núcleo del depurador, recuperando el estado y los handlers de RISC OS.

Una vez en el núcleo del depurador, habiéndose comprobado que se ha vuelto por una SWI no capturada se obtendrá qué instrucción del código del alumno ha provocado la interrupción. Ahora que ya tenemos la instrucción, debemos cargar su codificación y modificarla, eliminando la parte de ejecución condicional, ya que si se ha producido la interrupción que nos ha permitido llegar hasta este punto significa que ya se ha evaluado que se cumple la condición.

Para poder ejecutar la SWI se sustituye una instrucción placeholder con el código de la instrucción que habíamos procesado. Para asegurarnos de que la instrucción que hemos modificado sea la que se ejecute por el procesador utilizamos la SWI OS_SynchroniseCodeAreas, ya que como se explica en la documentación de la propia SWI [20], “si la CPU fuera a escribir

nuevas instrucciones a memoria entonces no hay ninguna garantía de que la unidad de fetch de instrucciones vaya a poder ver esas instrucciones. Las nuevas instrucciones puede que se queden en la cache de datos, esperando a que se haga writeback a memoria principal, o la cache de instrucciones puede que ya haya cacheado el área relevante – en cuyo caso no pedirá las nuevas instrucciones a la memoria principal."

Esta SWI procura que la instrucción modificada sea ejecutada invalidando la instrucción sustituida en memoria principal o cache, haciendo que lea la nueva instrucción.

Antes de ejecutar la SWI recuperamos el estado de los registros del usuario, y tras ejecutarlo guardamos los resultados en el estado de usuario. Lon únicos registros que necesitamos cargar y actualizar son desde R0 a R9, ya que el resto de los registros no se modifican tras ejecutar una SWI. Con esto quedaría concluido la ejecución de la SWI, con lo que se puede seguir con el resto de la ejecución del núcleo del depurador.

3.2.3. Ajustes necesarios en el código

La nueva presencia de un handler para las SWIs requirió cambiar parte del código previo del depurador. Esto es debido a que, una vez que sustituimos el handler nativo de RISC OS para las SWIs por el nuestro, todas las SWIs que se ejecuten tendrán que saltar a nuestro handler. El handler está pensado para ser llamado desde el código del usuario, por lo que trabaja con su contexto. Cuando estamos en el núcleo del depurador el contexto que estamos utilizando es el del depurador, el cual es independiente del que utiliza el usuario. Por lo tanto, si hubiese una instrucción SWI dentro del código del núcleo del depurador mientras está puesto el handler de las SWIs del depurador se intentaría ejecutar la función de la SWI con el contexto del usuario, consiguiendo resultados no deseados, y posiblemente afectando al contexto del programa del usuario.

La zona del depurador en la que no se pueden utilizar SWIs de forma directa queda delimitada entre el punto en el que se ponen los handlers de depurador hasta donde se restauran los handlers nativos de RISC OS. Los handlers del depurador se ponen al final de los preparativos para saltar a ejecutar el código del usuario. Quedan puestos hasta que se salta a alguno de los handlers del depurador desde el código del usuario, ya que todos los handlers terminan saltando a la función ReleaseHandler, donde se restauran los handlers de RISC OS. Esto supone que la mayor parte del código del depurador queda libre de esta limitación a la hora de usar SWIs. Este problema afecta a un total de tres instrucciones SWIs en el bloque de código afectado. Para solventar este problema se optó por modificar las tres llamadas a las SWIs afectadas.

La primera llamada que necesitamos modificar es a la SWI OS_ClaimProcessorVector, localizada dentro de la función ReleaseHandler, siendo la encargada de restaurar el handler nativo de las SWIs. Esta SWI es especialmente problemática, ya que se ejecuta cuando salimos del propio handler de las SWIs. Esto provocaría que se vuelva a saltar al handler, generando un bucle infinito.

En este caso se resuelve el impedimento saltando manualmente al handler de las SWIs nativo de RISC OS. Esto nos permite que se realice la función correcta sin tener que quitar o poner los handlers del depurador. Para saltar a este handler de forma manual tenemos que emular el proceso que se produce cuando se ejecuta una SWI de forma normal [17].

Este proceso se compone de las siguientes acciones: cambiar el modo del procesador a Supervisor, guardar en el registro SPSR del modo Supervisor el estado (registro CPSR) del modo desde el que se llamó la SWI, y poner en el registro LR del modo Supervisor la dirección de la

instrucción siguiente a la SWI que hemos ejecutado. En la Figura 10 vemos el código necesario para ejecutar la SWI OS_ClaimProcessorVector con este procedimiento.

```
msr cpsr, #MaskSvcMode+MaskDisInt @ Change to supervisor mode
@ Change the SPSR to return in system mode
mrs r0, cpsr
orr r0, r0, #MaskSysMode
msr spsr, r0

@ Release SWI handler
ldr lr,=SWI_release_end @ Load in the lr the next instruction to
the SWI we want the RISC OS handler to execute
mov r0,#0x2 @ release + SWI
ldr r1,=HandSWIRel
ldr r1,[r1] @ recover RISC OS handler
ldr r2,=SWIHandler @ remove our handler
mov pc, r1 @ Jump to the RISC OS SWI handler
swi OS_ClaimProcessorVector @ Is not gonna get executed directly
SWI_release_end:
```

Figura 10 – Código utilizado para ejecutar una SWI saltando manualmente al handler de las SWIs de RISC OS. Este es caso de la SWI OS_ClaimProcessorVector utilizada en la función RelesaeHandler.

De estos pasos el único no trivial es el de conseguir la dirección necesaria para el registro LR. La información que el handler de RISC OS recibe de este registro no es sólo la dirección a la que retornar cuando termine de ejecutar la rutina, sino que además le permite localizar la instrucción SWI que produce la excepción (la instrucción anterior a la apuntada por el registro LR), pudiendo averiguar así qué rutina de las disponibles debe ejecutar. Esto implica que nuestro código tendrá una instrucción SWI con el código de la rutina que pedimos al handler de RISC OS, pero que nunca debe llegar a ejecutarse, sirviendo sólo como referente para que el handler nativo sepa qué funcionalidad realizar.

Está solución no nos vale para todo los casos, ya que si el código de la propia SWI que queremos ejecutar produce que se salte a alguno de los handlers del depurador (por ejemplo, si contiene una llamada a otra SWI) será gestionada desde nuestro handler, provocando el proprio error que intentamos evitar.

En el caso de la SWI OS_ValidateAddress, localizada en el handler de las excepciones tipo Data Abort, no podemos utilizar directamente la solución anterior, debido a que en la implementación de esta SWI hay una llamada a la SWI OS_Memory. Para este caso el enfoque tomado fue sustituir la SWI por el propio código que RISC OS ejecuta cuando es llamada con los handlers nativos [21]. Este código se puede utilizar necesitando únicamente modificar la SWI OS_Memory y la llamada al servicio ValidateAddress, la cual se realiza con otra SWI. Estas SWIs podemos gestionarlas con el método utilizado para la SWI OS_ClaimProcessorVector.

La última SWI que es necesario modificar es la SWI OS_EnterOS, que aparece dentro del handler de las excepciones. La función en la que se encuentra esta SWI está destinada a gestionar el retorno desde la rutina de atención del usuario, haciendo que vuelva correctamente al punto

del código del usuario donde se había quedado la ejecución antes de producirse la interrupción. El acceso a esta función del handler es distinto al resto de casos que se dan en el depurador, puesto que es la única ocasión en el que el código del usuario salta explícitamente a una dirección del depurador. Esto implica que no sabemos en qué estado retornará el usuario al código del depurador.

El depurador debe contemplar un error de programación del usuario, en el que el código de la rutina de atención a una interrupción que haya programado regrese en un modo de CPU no privilegiado (User). Aunque se trata de un comportamiento no deseado, es un escenario de probable aparición, al tratarse de un depurador enfocado al uso docente. Si detectamos que ha ocurrido debemos cambiar a un modo privilegiado utilizando la SWI OS_EnterOS, ya que el depurador necesita estar en un modo privilegiado para proseguir con la ejecución del handler.

Sin embargo, esta SWI no la podemos ejecutar usando el handler nativo de RISC OS, ya que necesitamos estar en modo Supervisor antes de saltar a él, y ése es el propio cometido de la SWI. Tampoco podemos usar la implementación de nuestro handler de las SWIs por el problema antes detallado (estamos en el contexto del depurador). Por estas causas se optó por implementar una SWI nueva que solo se debe usar de forma interna, desde el código del depurador, la SWI OS_EnterOSAndSave.

3.2.3.1. SWI OS_EnterOSAndSave

Esta SWI se implementa como otra de las SWIs capturadas, aprovechando el número de SWI 0x45 que no tiene funcionalidad asignada en el sistema. La función que realiza es la de entrar en modo supervisor, guardar el contexto del usuario, hacer prefetch de la dirección que llamó a la SWI (evitando que se produzca un prefetch abort) y volver a la dirección llamadora sin quitar los handlers del depurador. Gracias al uso de esta SWI podemos aprovechar la sección encargada de guardar el estado del usuario que forma parte del handler de las SWIs.

Esto podría permitir en futuro centralizar la sección del guardado el contexto del usuario en un único punto, reduciendo así la cantidad de código redundante encargado de esta labor y convirtiéndole en el único punto de fallo para esta función, facilitando el depurado. Antes de tomar estas medidas se debe analizar el guardado del contexto en el resto de handlers, en busca de conflictos que obligasen a mantener dicha sección en el handler. Otro apartado a contemplar para trabajo futuro es optimizar esta SWI, ya que incluir una llamada a una SWI en los handlers introduce cierto overhead. Sin embargo, también aumenta la robustez y la predictibilidad de código, como ya se ha explicado.

Esta SWI se planteó implementarse como parte de la SWI OS_EnterOS, introduciendo una comprobación al comienzo de ésta que cambie el comportamiento si es llamada desde la función que gestiona la vuelta desde una rutina de atención de usuario. Finalmente se descartó dicha solución, al valorar la posible reutilización de la SWI OS_EnterAndSave en otras partes del código, como en el handler de interrupciones para detener la ejecución del modo Go-Fast.

3.3. Interrupciones periódicas

Para mejorar la usabilidad del modo Go-Fast se crea un mecanismo de interrupciones periódicas que permiten al usuario interactuar con la interfaz del depurador. En esta sección se detalla este mecanismo, viendo la motivación detrás de estas interrupciones, su implementación y su configuración.

3.3.1. Motivación

Uno de los problemas en el apartado de usabilidad de la implementación del modo Go-Fast es la total pérdida de control e interacción sobre el sistema mientras se ejecuta el código del usuario.

En la mayoría de los casos esto no supondría un inconveniente para el usuario ya que, si el código en ejecución ha sido debidamente probado, debería de finalizar correctamente, igual que en el resto de los modos del depurador. Si se produjera alguna SWI que requiriera la interacción con el usuario (como meter caracteres por consola) el usuario puede interaccionar como en el resto de los modos.

El problema surge cuando se produce algún error que no se manifiesta en todas las ejecuciones o errores que se producen al ejecutarse el código a mayor velocidad, y que no permiten que el código finalice. Esto conlleva que todo el sistema se quede en un estado no responsivo, obligándonos a reiniciar el equipo. Que el depurador no sea capaz de recuperar el control de la ejecución no sólo rompe el flujo de trabajo del usuario, sino que además nos impide ver en qué punto del programa se produjo el error, dificultando la depuración del código al no contar con ningún tipo de información.

3.3.2. Implementación

Para solventar este tipo de escenarios que pueden afectar gravemente a la usabilidad del depurador se optó por interrumpir periódicamente la ejecución del código del usuario para volver a la interfaz, leyendo así el input que el usuario introduzca por la consola. Esto le permitirá detener la ejecución mediante el uso de comandos, por ejemplo con el comando “stop”.

En la implementación de esta solución se utilizan las interrupciones del temporizador o *timer* del sistema de la Raspberry Pi. Este timer cuenta con 4 comparadores (numerados del 0 al 3) los cuales producen una interrupción cuando la parte baja del contador del timer tiene el mismo valor que uno de los comparadores. Debido a que el propio RISC OS utiliza el comparador 0 y que en las prácticas desarrollados en la universidad se utiliza habitualmente el comparador 3, se ha decidido utilizar el comparador 2. Esto deja el comparador 1 y el 3 para uso libre de los usuarios.

Para poder configurar el timer del sistema antes de entrar en el núcleo del depurador se mapea en memoria los registros que gestionan el timer. Para evitar un gasto de memoria o tiempo de ejecución innecesario en este mapeo, solo se realiza una vez durante el arranque del depurador.

La rutina de atención que atiende las interrupciones asociadas a este timer sigue un esquema similar a los handlers del depurador: empieza por salvar el estado del usuario (usando la SWI OS_EnterOSAndSave para simplificar el código), realiza su funcionalidad y termina saltando a la función ReleaseHandler. La función de esta rutina es reconocer la interrupción (Acknowledge) del timer y modificar el valor del comparador 2, estableciendo así la próxima interrupción.

La periodicidad elegida entre interrupciones del timer del sistema es de 65536 ciclos. La equivalencia en tiempo varía entre el hardware utilizado por sus distintas frecuencias de reloj. Aquí estudiaremos el caso de la Raspberry Pi 1B +, ya que es el hardware utilizado por los alumnos en las partes de laboratorio de las asignaturas de Estructura y Organización de Computadores, además del hardware utilizado durante el desarrollo de este proyecto. El timer del sistema tiene una frecuencia de 1 MHz, por lo que el tiempo entre interrupciones sería de aproximadamente 65,54 milisegundos, un periodo de tiempo que permite interactuar con el

depurado, aunque de forma menos fluida, sin afectar de forma significativa a los tiempos de ejecución del código de usuario.

Si en un futuro se deseara modificar el número de ciclos entre interrupciones, este valor se puede modificar fácilmente ya que está definido como una constante única (*GF_Timer_Cycles*) en el fichero de constantes del código ensamblador.

3.4. Evaluación

Haciendo uso de los programas de prueba de la suite de tests, que se detallarán en el Capítulo 1, se ha comprobado que la implementación del modo Go-Fast se comporta correctamente. Con esto podemos asegurar que los requisitos de funcionalidad básicos se cumplen.

Para evaluar la mejora en los tiempos de ejecución obtenidos con el modo Go-Fast se ha probado un código simple que realiza una gran cantidad de iteraciones en un bucle. Como referencia se ha ejecutado el mismo código en nativo y en el modo Go del depurador. En la Tabla 4 se muestran los resultados obtenidos.

Tipo de ejecución	Tiempo medio por instrucción
Nativo	1.62ns
!UCDebug en modo Go-Fast	1.49ns
!UCDebug en modo Go	37.66µs

Tabla 4 – Resultados de las mediciones de los tiempos de ejecución del código de prueba.

Como podemos observar los resultados obtenidos en el modo Go-Fast superan con creces los obtenidos en el modo Go, que son aproximadamente 20000 veces más lentos. Pero no sólo esto, si no que el tiempo de ejecución es menor que en nativo. Esto se debe a que mientras se ejecuta el código en el depurador con el modo Go-Fast se intenta aislar la ejecución deshabilitando todas las interrupciones que no correspondan con el código del usuario o con la interrupción periódica. De este modo, se atiende un menor número de interrupciones que en nativo, provocando que los tiempos de ejecución disminuyan.

De los resultados obtenidos se concluye que la combinación del uso del handlers de las SWIs con interrupciones periódicas no ralentiza en el código del usuario.

Sólo nos falta comprobar que el depurador sea capaz de ejecutar códigos que utilicen cualquiera de los periféricos planteados para las asignaturas, que era el objetivo inicial del proyecto. Para ello se prueba el manejo del sensor de humedad y temperatura DHT11 mostrado en la Figura 11.

Las pruebas se realizaron con uno de los códigos desarrollados durante el proyecto de innovación docente en el que se trataba el uso de periféricos en el depurador. Los resultados obtenidos con este código sobre el depurador antes de este proyecto eran de fallos de comunicación con el periférico. Como se observa en la Figura 12, actualmente cuando se ejecuta en el modo Go-Fast del depurador el código funciona correctamente y las mediciones nos devuelven valores de temperatura y humedad correctos.

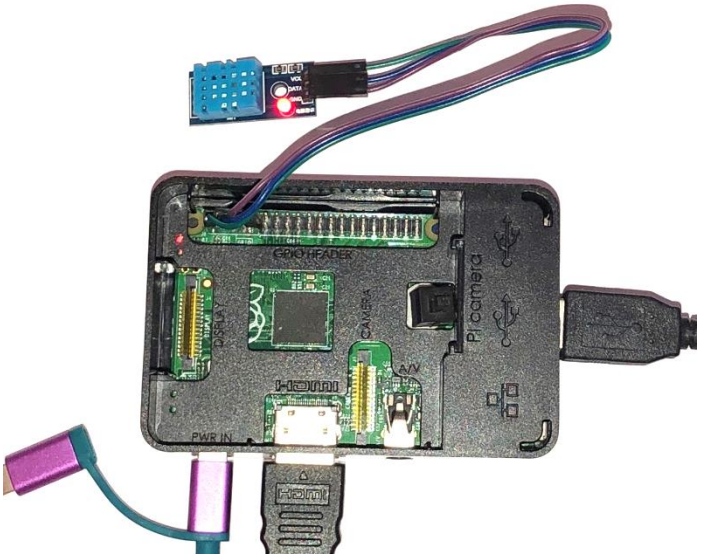

Figura 11 – Dispositivo DHT11 conectado a la Raspberry Pi.

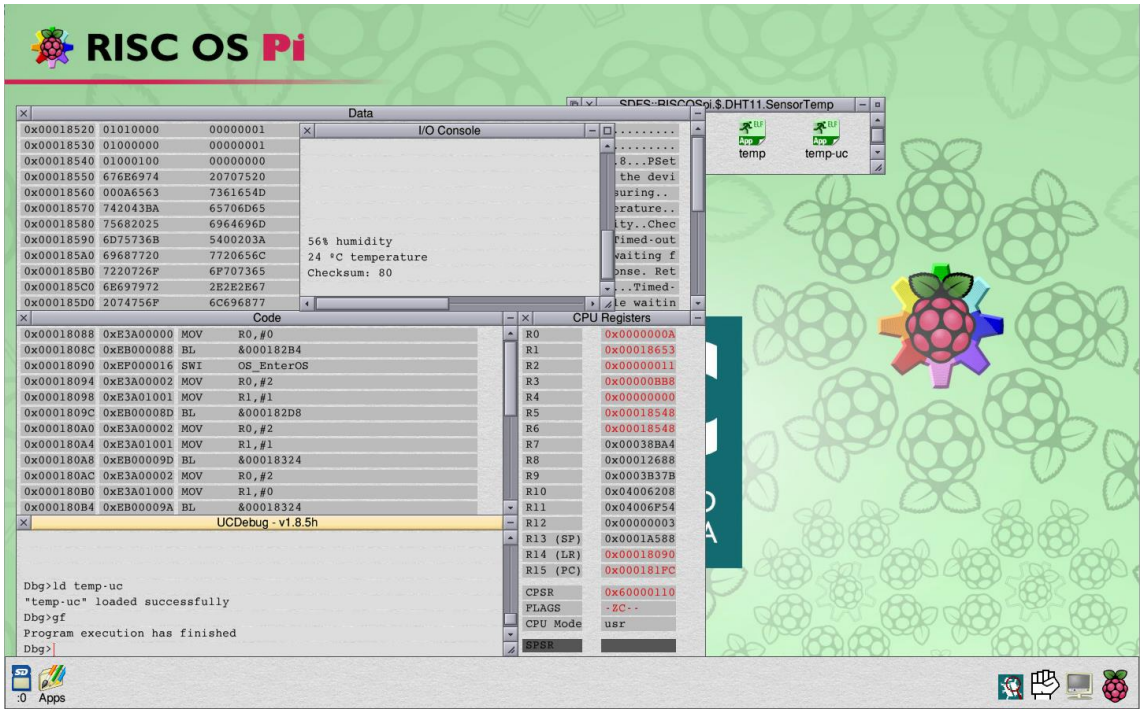

Figura 12 – Captura de pantalla del depurador ejecutando con éxito en modo Go-Fast un código que emplea el dispositivo de medición de temperatura y humedad DHT11.

Finalmente, en cuanto a los resultados de usabilidad obtenidos con la implementación de las interrupciones periódicas, la experiencia a la hora de interactuar con el depurador durante la ejecución de un código es de alta interactividad, cumpliéndose las expectativas planteadas para el mecanismo. El usuario es capaz de introducir caracteres e interactuar con el ratón sin notar de forma significativa el efecto de la ejecución del código.

4. Suite de tests

Durante el desarrollo del depurador se corre el riesgo de que al modificar una parte del código alguna de las piezas del software que ya funcionaban correctamente dejen de hacerlo. Para poder desarrollar el depurador y poder comprobar que los nuevos cambios no afectaban a las funcionalidades previas del depurador se desarrolló una suite de tests que verifican una parte específica de la funcionalidad esperada en el depurador.

Cada uno de estos tests comprueba que no se hayan dañado las funcionalidades del sistema. Estos tests no sólo devuelven una respuesta binaria de error o éxito al terminar, si no que siempre que sea posible se indica qué punto de la funcionalidad que está siendo testeada es la que falla, agilizando así el proceso de depuración.

Cada uno de los tests de la suite deben ejecutarse manualmente en el depurador. Esto se debe a que el depurador no cuenta con un método para ejecutar código y mostrar resultados que no sea a través de la interfaz. Además, existe una limitación severa sobre la funcionalidad que se puede desarrollar como script en la línea de comandos de RISC OS. Esto impide la automatización de los tests, que sería el escenario más deseable. Adicionalmente, algunos de los tests presentan limitaciones específicas: por ejemplo, la lectura de caracteres se realiza necesariamente desde teclado.

Los test desarrollados se pueden dividir en tres categorías, dependiendo de la función que comprueban: la ejecución correcta de las SWIs, los cambios de modo del procesador o el correcto reinicio al estado inicial del depurador tras finalizar la ejecución de un código del usuario.

4.1. Tests de las SWIs

Estos tests se encargan de comprobar que el funcionamiento de las SWIs capturadas y algunas de las no capturadas (las que se emplean más frecuentemente) es el esperado.

Es necesario comprobar el funcionamiento de cada una de las SWIs capturadas ya que su implementación ha sido sobrescrita por el código del depurador, con lo que necesitamos asegurarnos de que su funcionalidad da los mismos resultados que si se ejecutara nativamente sobre el sistema.

En el caso de las SWIs no capturadas sólo se comprueba un subconjunto reducido de las mismas. Esto se debe a que su funcionalidad no ha sido sobrescrita, por lo que lo único que tenemos que comprobar es si el depurador es capaz de ejecutar una SWI no capturada cualquiera.

Los test de las SWIs deben ejecutarse en un orden preciso, ya que los tests hacen uso de otras SWIs distintas a la que testan. Si no se siguiera ese orden no se podría asegurar que los errores detectados se deban a la SWI en cuestión comprobada en el test, dificultando la depuración del error. Los tests de las SWIs están numeradas y debe seguirse un orden ascendente en su ejecución.

4.1.1. SWI OS_Exit (0x11)

Esta SWI cuenta con un test (test 001_exit_def) cuya ejecución comprueba que se finalice la ejecución del programa tras la llamada a la SWI OS_Exit. Cuando se ejecute este test debemos observar un mensaje que nos indique que la ejecución ha terminado exitosamente. Este mensaje lo veremos siempre que se termine un código con esta SWI, siendo el mensaje que recibe el usuario cuando su código ha terminado correctamente.

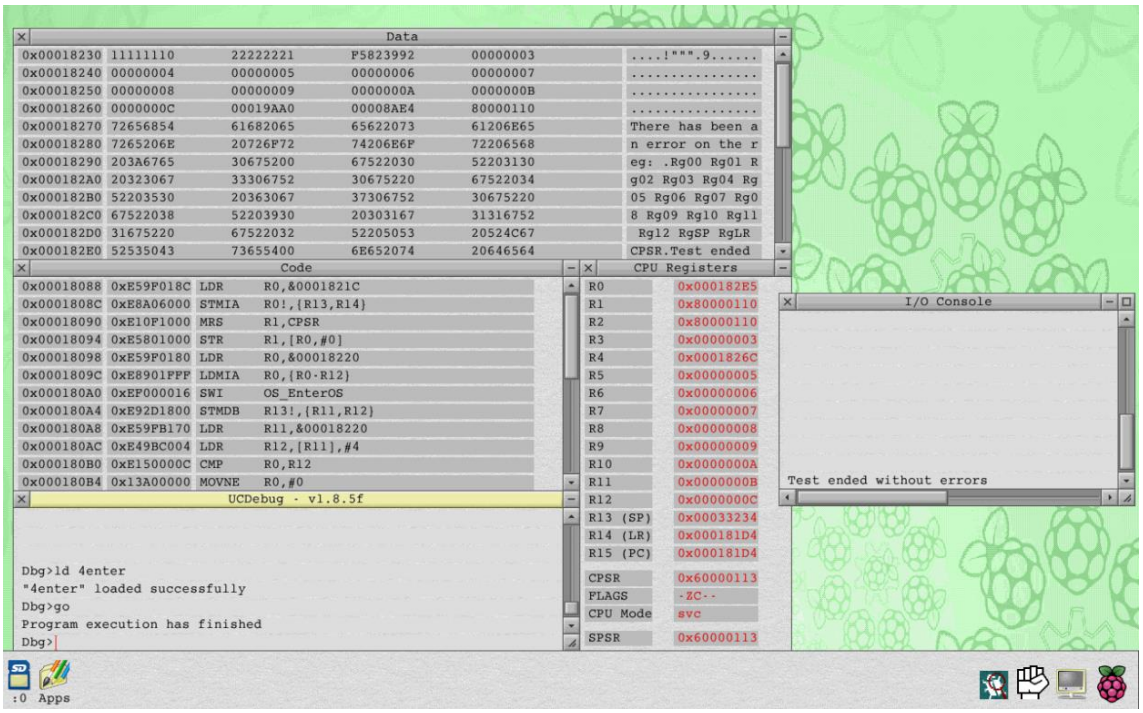

Figura 13 – Captura del depurador tras ejecutar el test 401_enteros. Al terminar la ejecución imprime un reporte con los resultados, en este caso, finalizado sin errores.

Esta SWI admite tres parámetros para el tratamiento de errores, aunque el depurador actualmente no los soporta, por lo que el test no lo tiene en cuenta. En un futuro, si fuera implementada esta funcionalidad, se necesitarían nuevos tests para comprobar su correcto funcionamiento.

4.1.2. SWI OS_WriteC (0x00)

Esta SWI cuenta con un test (test 101_writec) que imprime varios caracteres por pantalla. Este test comprueba que los caracteres sean imprimidos por pantalla, y que caracteres de varias categorías (alfanuméricos, caracteres especiales o letras acentuadas entre otros) se impriman correctamente. En la documentación y en los comentarios del código del test está descrito el resultado esperado por el test para facilitar la verificación del mismo.

4.1.3. SWI OS_Write0 (0x02)

Esta SWI cuenta con un test (test 201_write0) que imprime tres cadenas de caracteres por pantalla. Una de las cadenas de caracteres imprimidas está compuesta por caracteres de varias categorías (alfanuméricos, caracteres especiales o letras acentuadas entre otros) para comprobar que es capaz de imprimir todo el espectro de caracteres disponibles. Además de esto, el test comprueba que tras ejecutar la SWI el puntero a la cadena de caracteres se actualice, apuntando al final de la misma. Para comprobar que esto ocurre se hacen tres llamadas a la SWI seguidas, habiéndose almacenados tres cadenas de caracteres consecutivas. Esto provocará que se ejecuten todas esas cadenas de caracteres en orden. En la documentación y en los comentarios del código del test está descrita la salida que se espera recibir por pantalla en caso de funcionamiento correcto.

Durante el desarrollo de este test se detectó un error en la implementación de la SWI. Al ejecutar el test se imprimía por pantalla tres veces la primera cadena de caracteres. Tras analizar más en

detalle qué estaba sucediendo se descubrió que la SWI no actualiza el puntero a la cadena de caracteres tras su ejecución. Al no ser una funcionalidad que afecte el uso regular de la SWI se ha dejado marcado como un desarrollo pendiente para actualizaciones del depurador futuras.

4.1.4. SWI OS_ReadC (0x04)

Esta SWI cuenta con un test (test 301_readc) que le pide al usuario que introduzca una serie de caracteres de varias categorías (como en las SWIs OS_WriteC y OS_Write0). Cuando el programa registra el carácter lo imprime por pantalla para comprobar que el carácter leído ha sido procesado por la SWI correctamente.

4.1.5. SWI OS_EnterOS (0x45) y OS_LeaveOS (0x7C)

La verificación del comportamiento de estas SWIs forma parte de los tests de cambio de modo y los tests de las SWIs simultáneamente, debido a que estas SWIs se encargan de entrar y salir del modo supervisor.

Los tests 401_enteros y 501_leaveos se ocupan de comprobar la correcta entrada y salida del modo supervisor, respectivamente. Una descripción más detallada de estos dos tests se encuentra en la sección 4.3 en la que se explica el funcionamiento de los test de cambio de modo.

4.1.6. SWI OS_ConvertInteger4 (0xDC)

Para testear que el funcionamiento de las SWIs no capturadas sea el esperado se decidió implementar un test que compruebe el funcionamiento de una SWI no capturada sencilla. Para ello se creó el test 601_convert_int que comprueba el funcionamiento de la SWI OS_ConverInteger4, que convierte un entero de 32 bits a una cadena de caracteres imprimible por consola. Este test comprueba que los registros que se modifican con la SWI tengan los valores esperados.

Este test verifica que el depurador procesa las SWIs no capturadas adecuadamente. Esto se debe a que, para que este test pase sin detectar errores, el depurador tiene que llamar a la SWI adecuada, realizar el paso de parámetros recibidos y devolver los resultados correctamente. Si se hacen correctamente estos tres pasos se puede concluir que cualquier fallo detectado en una SWI no capturada ocurre dentro del handler nativo de las SWIs de RISC OS.

4.1.7. SWI OS_Hardware (0x7A)

La SWI OS_Hardware es otra de las SWIs no capturadas a las que se le ha asignado un test propio, el test 701_hardware. Esta SWI tiene incorporadas multitud de funciones, pero el test se aplica sólo a la llamada de la rutina HAL (*Hardware Abstraction Layer*) con el punto de entrada “HAL_TimerDevice”, que permite usar la capa de abstracción HAL para identificar el número de dispositivo asociado en las interrupciones IRQ del procesador a un timer concreto del sistema.

Esta SWI tiene el requerimiento especial de que necesitamos ejecutarla con los handlers nativos de RISC OS. En la versión original del depurador la única forma de que esto pasara era siendo una de las SWIs capturadas por el depurador, razón por la que se crea este test. Actualmente todas las SWIs no capturadas se ejecutan desde el núcleo del depurador, con lo que ya no necesita un tratamiento aislado.

Se decidió mantener el test ya que sirve como test adicional para comprobar que las SWIs no capturadas funcionan correctamente, y porque esta SWI se puede llegar a usar en las prácticas de gestión de dispositivos de entrada/salida para mostrar la relación entre un dispositivo y la interrupción asociada.

4.1.8. SWI OS_ClaimDeviceVector (0x69) y OS_ReleaseDeviceVector (0x4C)

Las SWIs OS_ClaimDeviceVector y OS_ReleaseDeviceVector se encargan de asignar y retirar las rutinas de atención a las interrupciones generadas por un dispositivo, respectivamente.

Por la necesidad de usar estas SWIs en conjunto, los test desarrollados comprueban ambas funcionalidades simultáneamente. Los tests 801_claim_release y 802_claim_without_release hacen uso del timer del sistema, utilizando el comparador 3, para comprobar que la SWI OS_ClaimDeviceVector asigna correctamente la rutina a la interrupción.

El test 801_claim_release comprueba que el comportamiento normal de estas SWIs es el esperado. El test programa una interrupción del timer del sistema y se queda en un bucle haciendo encuesta a una variable de control que se modifica desde la rutina de atención a la interrupción. Cuando detecta el cambio en la variable, el programa principal desactiva las interrupciones y termina. En este programa de prueba se utilizan la SWI OS_ClaimDeviceVector para asignar la rutina de atención a la línea de interrupción del timer del sistema, y la SWI OS_ReleaseDeviceVector para retirar la rutina.

El test 802_claim_without_release se debe ejecutar seguido del test 803_release_without_claim. El test 802 utiliza la SWI OS_ClaimDeviceVector para asignar la rutina de atención a la línea de interrupción del timer del sistema, de similar manera al test 801, pero termina la ejecución sin desasociar la rutina a la interrupción del timer. A continuación se cargaría el test 803, que intentará retirar la rutina que se dejó puesta en el test 802. Al haberse cargado un código nuevo en el depurador, la rutina que se dejó sin desasociar de la interrupción en el test 802 debería haberse eliminado de la tabla de interrupciones que maneja de forma interna el depurador, con lo que el test 803 debería retornar un mensaje de error que indica que se está intentado quitar una rutina que no estaba puesta. En caso de que el programa termine sin errores puede indicar que la rutina sigue asignada desde el anterior programa, por lo que al cargar un código el depurador no reinicia su tabla de interrupciones, o que la propia SWI OS_ReleaseDeviceVector no está funcionando correctamente.

4.2. Tests de cambio de modo

Cada modo del procesador cuenta con una serie de registros privados (excepto los modos usuario y sistema, que comparten todos los registros entre sí), como se puede observar en la Tabla 2 y se detalló en la Sección 2.1.1. El depurador gestiona un contexto por cada modo de ejecución, y en ellos guarda los valores de estos registros privados. Para evitar sobrescribirlos cuando se cambia de modo, el depurador se encarda de salvar el estado de los registros privados del modo previo, y de cargar los asociados al modo al que se va a entrar.

Se han desarrollado tests relativos a cada modo privilegiado para comprobar exhaustivamente que todos los registros toman los valores esperados. Para ello se comprueba que los registros mantienen su valor tras cambiar de modo, menos los privados, que deberían cambiar al valor almacenado en el contexto del modo al que hemos cambiado. Cada modo cuenta con dos tests, comprobando uno de ellos la entrada a ese modo, y el otro la salida.

Estos test comprueban registro a registro que los valores tras la operación de entrada o salida del modo son los esperados, y en caso contrario imprimen un mensaje por pantalla que nos indica cuál ha sido el primer registro que presentó un valor erróneo. En el caso del modo supervisor se reutilizan los test 401_enteros y 501_leaveos, los cuales ya hacen las comprobaciones requeridas.

Para la ejecución de estos test se necesita por lo menos haber pasado correctamente los tests de las SWIs OS_Exit, OS_Write0, OS_EnterOS y OS_LeaveOS, ya que estas SWIs son utilizadas en todos los tests de los modos (de otra manera no se podría asegurar que los errores reportados sean debidos a el cambio de modo del procesador).

4.3. Tests de reinicio del depurador

Se ocupan de comprobar que el estado del depurador vuelve a su configuración inicial cuando cargamos un código del usuario en el depurador después de haber ejecutado previamente otro código. Esto implica que el sistema vuelva a modo usuario, que las pilas de los distintos modos del procesador se reinicien (el Stack Pointer vuelve a su valor inicial) y que el registro CPSR vuelva a su estado inicial.

Cada uno de los test de este apartado se componen de dos programas que se han de ejecutar consecutivamente:

- Un primer programa que modifica el componente del sistema a testar y finaliza sin devolverlo a su estado inicial.
- Un segundo programa que comprueba que el componente del sistema a testar ha vuelto su estado inicial tras haberse cargado un código nuevo.

4.3.1. Reset del CPSR

Este test modifica los bits del CPSR que no provocan cambios en la configuración del procesador, siendo estos los bits Q, GE, A, I y F.

El programa de comprobación compara el valor actual del CPSR con el valor inicial guardado de forma estática. Si alguno de los bits no corresponde con su valor esperado retornará un mensaje de error indicando cual es el primer bit detectado con un valor no reiniciado. Si se reiniciaron todos correctamente mostrará un mensaje de éxito.

4.3.2. Reset de las pilas de los modos

Este test modifica las pilas de los distintos modos del procesador y termina sin restaurarlas a su posición inicial.

El programa de comprobación compara la dirección almacenada en el registro Stack Pointer (SP) con los valores iniciales esperados. Si el registro SP de alguno de los modos no ha vuelto a su valor inicial se retorna un mensaje de error indicando cual ha sido el primer SP que se ha detectado con un valor no reiniciado. En caso de que todos los registros SP se hayan reiniciado correctamente se retorna un mensaje de éxito.

La dirección a la que apunta cada pila originalmente no se puede almacenar de forma estática en código ya que con cada modificación del código del depurador cambia su tamaño, y con él la dirección original de cada pila (las pilas están declaradas como parte de una variable de gran tamaño que comprende el segmento de memoria que puede emplear el usuario del depurador, y su dirección física cambia con las variaciones del código). Por ello, la primera parte del test que se encarga de modificar las pilas también se encarga de guardar en un fichero auxiliar la dirección origen de las distintas pilas, permitiendo así que la segunda parte del test pueda leer de allí los valores.

4.3.3. Reset del modo del procesador

Para completar los tests relativos a los modos del procesador solo falta por comprobar que se reinicia al modo usuario al cargar un nuevo programa, aunque se haya terminado el anterior programa en cualquiera de los modos privilegiados.

Este test está compuesto por siete programas, de los cuales seis se encargan de cambiar a cada uno de los modos privilegiados (terminando sin volver al modo usuario).

El programa de comprobación se encarga de leer el CPSR y retornar un mensaje de éxito si detecta que se ha iniciado el programa en modo distinto usuario, o retornar un mensaje de error si se inició en cualquier otro modo.

5. Conclusiones

El objetivo inicial de este proyecto era desarrollar un nuevo modo de ejecución para !UCDebug, un depurador de código en ensamblador ARM. Este modo se requiere para poder incluir periféricos adicionales a las prácticas de algunas de las asignaturas del área de Estructura y Organización de Computadores que se imparten en la Universidad de Cantabria. Hasta ahora no era posible manejarlos debido a que estos periféricos tienen requisitos temporales muy estrictos. Los modos disponibles en el depurador introducen un gran overhead en la ejecución, provocando que los tiempo de ejecución aumenten en cuatro ordenes de magnitud respecto a la ejecución en el sistema nativo, lo cual se aleja en gran medida de los requerimientos de estos periféricos.

Para corregir esta situación se decide desarrollar una nueva implementación del modo Go-Fast que se ajuste a estos requisitos. Este modo permite ejecutar el código con un menor control sobre la ejecución, consiguiendo a cambio tiempos de ejecución más cercanos a los del sistema nativo.

En primer lugar, se ha desarrollado una suite de tests para comprobar que las modificaciones introducidas sobre el depurador durante este proyecto no dañan las funcionalidades base del depurador. Esta suite está formada por varios tests independientes que comprueban las diferentes características del depurador. Para facilitar la depuración de errores cada tests presenta información sobre los errores encontrado con gran nivel de detalle. Además, cada test cuenta con una lista de dependencias, para asegurar que un fallo detectado sólo se deba a una funcionalidad especifica.

Gracias al empleo de la suite de tests se consiguió localizar y solventar numerosos errores durante el desarrollo del proyecto, errores que de otra forma habrían pasado inadvertidos a una versión pública del depurador, con su consiguiente impacto en los usuarios y el trabajo añadido de depuración. Por ello el desarrollo de la suite de tests ha supuesto un ahorro de recursos a la hora de desarrollar en un futuro el depurador y consigue detectar más fallos con mayor agilidad, asegurando un mayor nivel de calidad del depurador como herramienta de depuración de código. Además, esta suite de tests ha permitido descubrir la presencia de un error de funcionalidad con la SWI 0x02, que estaba antes del comienzo de este trabajo, y que los desarrolladores originales desconocían.

Para el modo Go-Fast se ha seguido una estrategia basada en el desarrollo de un nuevo handler para las SWIs. Este handler permite gestionar de forma correcta y eficiente la captura de aquellas SWIs que es necesario tratar de forma especial y dejar el estado adecuado para que el sistema atienda directamente las demás SWIs. Esto permite que cuando se ejecuta código en modo Go-Fast solo se tenga que interrumpir la ejecución por breakpoints de usuario, cuya gestión es mucho más liviana para el depurador que el manejo de las SWIs.

Como funcionalidad adicional al modo Go-Fast, se implementó un sistema de interrupciones periódicas con el objetivo de permitir al usuario interactuar con el depurador durante la ejecución, principalmente para detenerla. Estas interrupciones provocan que el flujo de ejecución regrese al núcleo del depurador, permitiendo que el depurador pueda leer los inputs del usuario. Esto mejora en gran medida la usabilidad del modo Go-Fast ya que cualquier fallo que provoque que el código itere indefinidamente en un bucle no supondrá reiniciar el sistema.

Se ha comprobado que el tiempo de ejecución en el modo Go-Fast se ajusta a los requerimientos temporales que se marcaron al comienzo del trabajo. Los resultados obtenidos son muy

positivos, ya que se observa una ejecución de aproximadamente 20000 veces más rápido que el modo Go, y además se ejecuta un 7.5% más rápido que la ejecución en nativo, ya que el depurador proporciona aislamiento respecto al sistema.

Finalmente se ha comprobado que el depurador permite el manejo de periféricos con requisitos temporales muy exigentes. Para ello se ha probado la ejecución de un código que utiliza el sensor de temperatura y humedad DHT11, uno de los periféricos que el depurador no era capaz de gestionar. La ejecución de este código ha sido exitosa, por lo que el desarrollo realizado ha cumplido su motivo original. Por los resultados obtenidos en este proyecto se concluye que el uso de nuevos periféricos con restricciones similares es posible, facilitando su uso en futuras prácticas de las asignaturas.

5.1. Trabajo futuro

A lo largo del proyecto se han planteado nuevas funcionalidades que, por alejarse del objetivo inicial o por requerir demasiado tiempo, se ha decido no implementar. Como referencia para futuros desarrollos se listan aquí varios proyectos de interés:

- Durante la primera ejecución de los tests del depurador se encontró un error en la SWI 0x02 OS_Write0, presente en el código original del depurador. Al no ser error un qué imposibilite el uso normal de la SWI se deja como tarea pendiente a desarrollar.
- Durante el desarrollo del proyecto se ha identificado que en ocasiones ocurren excepciones FIQs en mitad de la ejecución del código del alumno. Minimizar estas ocurrencias aseguraría que no interfiere con la ejecución del código del usuario. Así un alumno vería su código como una ejecución completamente aislada, que es como se espera para una labor de aprendizaje. Además, estas interrupciones pueden provocar volver al núcleo del depurador, lo cual implica una pérdida de rendimiento que también interesaría eliminar.
- Una nueva característica que ayudaría con la labor de depuración podría ser añadir la función de registrar las instrucciones ejecutadas en un archivo de texto. Podría implementarse como un nuevo modo o como una opción para el modo Go.
- El depurador podría llegarse a usar en un futuro por otras universidades para realizar una labor docente similar. Al encontrarse con programas de estudio distintos esto puede producir requerimientos de nuevas funcionalidades para que el depurador se adapte a sus necesidades.

Bibliografía

[1] P. Fuentes, C. Camarero, C. Martínez y F. Vallejo, «Tecnología low-cost para motivar al alumno,» de *JENUI*, Murcia, 2019.

[2] A. B. Ferreira, *Evaluación de la plataforma Raspberry Pi para la docencia de Microprocesadores,* Trabajo de fin de grado, Universidad de Cantabria, 2015.

[3] «Raspberrypi.org,» [En línea]. Available: https://www.raspberrypi.org/. [Último acceso: 20 Agosto 2020].

[4] «Raspberry Pi model 1 B+,» [En línea]. Available: https://www.raspberrypi.org/products/raspberry-pi-1-model-b-plus/. [Último acceso: 3 Septiembre 2020].

[5] E. Z. S. Santamaría, *Desarrollo de un entorno de prácticas de laboratorio en ensamblador de ARM para un sistema Raspberry Pi con el sistema operativo RISC OS,* Trabajo de fin de grado, Universidad de Cantabria, 2017.

[6] «Raspberry Pi Model B+ specifications,» [En línea]. Available: https://cdn-shop.adafruit.com/datasheets/pi-specs.pdf. [Último acceso: 3 Septiembre 2020].

[7] S. L. Harris y D. M. H. Waltham, Harris & Harris (Digital Design and Computer Architecture), Massachusetts: Morgan Kaufmann, 2016.

[8] «Riscosopen.org,» [En línea]. Available: https://www.riscosopen.org/content/. [Último acceso: 3 Septiembre 2020].

[9] J. W. Guttorm Vik, C. Hetherington y F. Graute, «StrongEd – a programmer's text editor for RISC OS,» !StrongED, 2020. [En línea]. Available: http://stronged.iconbar.com/. [Último acceso: 3 Septiembre 2020].

[10] «GCC, the GNU Compiler Collection,» Free Software Foundation, Inc, [En línea]. Available: http://gcc.gnu.org/. [Último acceso: 3 Septiembre 2020].

[11] «BerryClip+ User Guide,» [En línea]. Available: https://bitbucket.org/MattHawkinsUK/rpispy-berryclip-plus/downloads/. [Último acceso: 3 Septiembre 2020].

[12] «DHT11 Product Manual,» [En línea]. Available: https://components101.com/sites/default/files/component_datasheet/DHT11-Temperature-Sensor.pdf. [Último acceso: 3 Septiembre 2020].

[13] «Hitachi HD44780 Display Controller/Driver Product Manual,» [En línea]. Available: https://www.sparkfun.com/datasheets/LCD/HD44780.pdf. [Último acceso: 3 Septiembre 2020].

[14] «HC-SR04 Datasheet,» [En línea]. Available: https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf. [Último acceso: 3 Septiembre 2020].

[15] «RC522 RFID Module Datasheet,» [En línea]. Available: https://www.hobbytronics.co.uk/datasheets/sensors/MFRC522.pdf. [Último acceso: 3 Septiembre 2020].

[16] A. Holdings, «Arm FY2018 Q2 Key Performance Indicators report,» Septiembre 2018. [En línea]. Available: https://www.arm.com/-/media/global/company/investors/Financial%20Result%20Docs/Arm_SBG_Q2_2018_Financial_Data_NonFinancial_KPIs_FINAL.xlsx. [Último acceso: 3 Septiembre 2020].

[17] «ARM Architecture Reference Manual,» ARM Limited, Copyright © 1996-1998, 2000, 2004, 2005.

[18] «ARM Developer Suite Developer Guide - System mode,» [En línea]. Available: https://developer.arm.com/documentation/dui0056/d/handling-processor-exceptions/system-mode. [Último acceso: 3 Septiembre 2020].

[19] P. Fuentes, «!UCDebug,» [En línea]. Available: https://fuentesp.github.io/UCDebug/. [Último acceso: 3 Septiembre 2020].

[20] «RISC OS open - SWI OS_SynchroniseCodeAreas,» [En línea]. Available: https://www.riscosopen.org/wiki/documentation/show/OS_SynchroniseCodeAreas. [Último acceso: 3 Septiembre 2020].

[21] «RISC OS Repository - ArthurSWIs,» [En línea]. Available: https://gitlab.riscosopen.org/RiscOS/Sources/Kernel/-/blob/master/s/ArthurSWIs. [Último acceso: 3 Septiembre 2020].

Anexo 1

Entornos de desarrollo

En este proyecto se ha trabajado en dos entornos de desarrollo, en los que se ha utilizado los mismos equipos principales: una Raspberry Pi modelo 1B+ con el SO RISC OS y un PC con el SO Windows 10. Aunque todo el desarrollo se podría haber realizado únicamente con la Raspberry Pi, se optó por utilizar un equipo auxiliar ya que el equipo Windows nos permite trabajar con el sistema de control de versiones Git, además de contar con una mayor variedad de editores de código. La Raspberry Pi se ha utilizado principalmente para compilar y ejecutar el código del depurador y de la suite de tests.

En el primer entorno los dos equipos tenían puestos de trabajo independiente. Ambos equipos estaban conectados a una red privada y se comunicaban a través de una carpeta compartida.

En el segundo entorno la Raspberry Pi se gestionaba a través del equipo Windows. Se utiliza el programa *VNC viewer* para controlar la Raspberry Pi, la cual conectamos mediante un cable Ethernet. El traspaso de ficheros se realizó utilizando una memoria USB.

Ensamblado y enlazado de código

Para generar un programa que se pueda ejecutar con el depurador !UCDebug debemos de ensamblar y enlazar el código. En la Figura 14 se muestran los dos comandos necesarios, siendo el primero el de ensamblado y el segundo el de enlazado. El comando de enlazado requiere el parámetro “Ttext” para indicarle que comience el código del usuario en la dirección 0x18088.

```
as -o <objeto> <programa>

ld -Ttext=18088 -o <ejecutable> <objeto1> <objeto2> … <objetoN>
```

Figura 14 – Comandos de ensamblado y enlazado para generar un programa ejecutable por el depurador.

Anexo 2

Código utilizado para la evaluación de los tiempos de ejecución

Para evaluar los tiempos de ejecución del sistema nativo, el depurador en modo Go y el depurador en modo Go-Fast se ha utilizado el código de la Figura 15:

```
.data
Cycles: .word 500000 @ Number of iterations

.text
.global _start

_start:
      ldr r0, =Cycles
      ldr r0, [r0]
loop:
      sub r0, r0, #1
      cmp r0, #0
      bne loop
      swi 0x11
.end
```

Figura 15 – Código utilizado para evaluar los tiempos de ejecución.