Facultad de Ciencias

Análisis de la ubicación de los segmentos de código de un programa bajo RISC OS

Analysis of the location of the code segments of a program under RISC OS

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Saúl Fernández Tobías

Director: Pablo Fuentes Sáez

Co-Director: Fernando Vallejo Alonso

Julio - 2022

Resumen.

Las asignaturas *Introducción a los Computadores* y *Estructura de Computadores*, impartidas por la Universidad de Cantabria a los alumnos de ingeniería informática, se centran en explicar aspectos fundamentales del funcionamiento de un computador. Como parte de la formación práctica de las asignaturas, se solicita al alumnado que implemente diferentes programas en lenguaje ensamblador para la arquitectura ARM.

Una de las herramientas más utilizadas durante estas prácticas es el depurador. Esta herramienta permite a los alumnos ejecutar el código desarrollado controlando el número de instrucciones que se ejecutan, y mostrando qué tipo de fallos presenta el programa que se esta depurando. Sin embargo, el sistema operativo bajo el que se realizan las prácticas, RISC OS, no poseía un depurador de código abierto que se pudiese utilizar durante las prácticas. Por ello, desde el departamento de Ingeniería Informática y Electrónica de la UC se ha desarrollado un depurador propio denominado UCDebug, en el marco de un proyecto de innovación docente. Durante su desarrollo se primó la simplicidad del código y la pronta disponibilidad para su adopción en las asignaturas involucradas.

El depurador UCDebug ejecuta los programas del alumno como si fuesen parte de su propio código. Como restricción inherente a esta estrategia de diseño, para que un programa pueda ser depurado en la herramienta, se debe enlazar el código de este a una dirección diferente a la habitual en el sistema operativo. Esto provoca que el alumno deba enlazar el código en dos direcciones diferentes, una para ejecutarlo directamente sobre RISC OS y otra para poder depurarlo con la herramienta.

Durante este proyecto se realiza un estudio de la viabilidad de una propuesta de mejora para el depurador, mediante la cual ya no sería necesario enlazar el código que se quiera depurar a una dirección diferente a la usada por el sistema operativo. La modificación propuesta consiste en reubicar los segmentos de código que componen el ejecutable del depurador. Con este cambio, sería posible cargar en este espacio el código que se quiere depurar. Como resultado de esta modificación obtendríamos que el código que se depura en la herramienta se ubica en las mismas direcciones de memoria a las que se encontraría si se ejecutase directamente bajo RISC OS, no siendo necesario enlazar el programa del alumno a direcciones diferentes.

Este proceso no es trivial, ya que el depurador es también una aplicación que se ejecuta bajo RISC OS. Por ello, necesita ubicarse al comienzo del espacio de direcciones de memoria de usuario, donde se pretende colocar el programa del alumno. Además, durante el desarrollo de la herramienta se han utilizado de forma conjunta los lenguajes C y ensamblador, haciendo uso de varias librerías propias del sistema operativo RISC OS. Previsiblemente, el uso de estas librerías puede complicar el proceso de análisis, debido a que no hay información previa acerca de cómo estas librerías son tratadas por el compilador y el linker.

El trabajo consiste en un análisis de los diferentes segmentos de código que componen un fichero ejecutable, y, en concreto, los que componen el depurador, ideando consecuentemente diferentes estrategias de reubicación. Este análisis ha demostrado que es posible reubicar los segmentos de código que componen el ejecutable, pero que no es posible situarlos al comienzo del espacio de direcciones. Esto es debido a que los ficheros ejecutables comienzan con una cabecera con información sobre la estructura del programa y que no puede reubicarse.

Durante el desarrollo del proyecto se ha determinado que no es posible sobrescribir esta cabecera de forma completa. No obstante, el programa del alumno que se quiere depurar tiene una cabecera del mismo tamaño. Además, durante la depuración, el código del alumno se trata como parte del propio código del depurador, por lo que se ha determinado que no necesita de sus cabeceras originales para ser ejecutado. Por lo tanto, si se obvian durante el cargado del código las cabeceras, el programa seguiría situado en la misma dirección en memoria, no siendo necesario un enlazado adicional.

Se ha determinado, por tanto, que la propuesta de mejora es viable, y que para ser realizada se debe alojar un segmento con espacio reservado al comienzo del espacio de direcciones de memoria del depurador.

Además, se deben obviar las cabeceras durante el copiado del código del alumno al depurador, tal y como se muestra más detalladamente en este documento.

Palabras clave.

Análisis de código, RISC OS, Depurador, Docencia, GCC, UCDebug

Abstract.

The courses *Introduction to Computers* and *Computer Structures* given at the University of Cantabria to computer engineering students focus on explaining fundamental aspects of how a computer works. Among the topics covered are programming in assembler language under the ARM architecture. Consequently, in the practical part of the courses, students are asked to implement different programs in that language.

One of the most used tools during these sessions is the debugger. This tool allows students to execute their codes in a controlled fashion, showing what kind of failures the program being debugged presents. However, the operating system under which the practices are carried out, RISC OS, did not have an open source debugger with a graphical user interface.Therefore, a new tool called UCDebug has been developed within the Department of Computer and Electronic Engineering, as part of a teaching innovation project. During its development, the simplicity of the code and the prompt availability for its adoption in the courses involved were paramount.

UCDebug runs the student programs as if they were part of its code. As an inherent restriction of this design strategy, in order for a program to be debugged in the tool, its code must be linked to a different address than the usual one in the operating system. This causes the student to link the code in two different directions, one to run it directly on RISC OS and the other to run it on the tool during the debugging process.

During the project, a study of the feasibility of an improvement proposal for the debugger is carried out, by means of which it would no longer be necessary to link the code to be debugged to an address different from the one used by the operating system. The proposed modification consists of relocating the code segments that make up the debugger executable so that the initial segment remains available in memory. With this change, it would be possible to load in this space the code that you want to debug. As a result of this modification, the code debugged in the tool would be located in the same memory addresses as if it were executed directly under RISC OS, removing the need to link the student program to different addresses.

This process is not trivial, since the debugger is also an application running under RISC OS, so it needs to be located at the beginning of the user memory address space, where the student program is intended to be placed. In addition, during the development of the tool have been used together the C and assembler languages, making use of several libraries of the RISC OS operating system.

Therefore, an analysis of the different code segments that make up an executable file, and specifically those that make up the debugger, has been carried out, consequently devising different relocation strategies. This analysis has shown that it is possible to relocate the code segments that make up the executable, but that it is not possible to place them at the beginning of the address space. This is due to the presence of a header at the beginning of the executable file, indicating the memory structure of the program.

During the development of the project it has been determined that it is not possible to overwrite this header. However, the student program has a header with the same size and memory location. Additionally, during debugging, the student code is treated as part of the debugger code itself, so it has been determined that its headers are not needed for its execution. If the headers are skipped during code loading, the program will continue to be located at the same address in memory, not requiring an additional linking process.

Therefore, it has been determined that the improvement proposal is feasible, and that to be carried out, a segment with reserved space must be accommodated at the beginning of the debugger address space. Headers must also be skipped when copying the code from the student to the debugger.

Keywords.

Code analysis, RISC OS, Debugger, Teaching, GCC, UCDebug

Índice

Índice de figuras

1. Introducción y objetivos

Este capítulo está dedicado a exponer las causas que motivan la realización del proyecto. Se detalla el modelo de trabajo que se sigue en la parte práctica de las asignaturas pertenecientes a la mención de Ingeniería de Computadores, especialmente aquellas en las que se ejercita la programación en ensamblador ARM. Se muestra qué inconvenientes tiene el modelo de trabajo actual, y cómo se propone solucionarlo.

1.1. Motivación.

Durante su paso por la *Universidad de Cantabria*, los estudiantes de Ingeniería Informática deberán adquirir conocimientos relativos al diseño y organización de los computadores. Estos conocimientos se imparten en asignaturas pertenecientes a la rama de Ingeniería de Computadores, tales como *Introducción a los Computadores* y *Estructura de Computadores*. Dichas asignaturas son consideradas básicas en la titulación, y todo estudiante debe cursarlas para poder graduarse. En estas asignaturas se explican aspectos tales como los diferentes niveles de los lenguajes de programación o los *sets* de instrucciones de una arquitectura. Estas asignaturas toman como referente la arquitectura ARM [10]. Esta arquitectura sigue la filosofía RISC (*Reduced Instruction Set Computing*), presentando un número reducido de formatos en las instrucciones y teniendo estas siempre un tamaño constante. Además, una de las motivaciones por las que se ha escogido esta arquitectura es su amplia presencia en el mercado [11][4].

Uno de los aspectos más destacados de las asignaturas ya mencionadas es su alto componente práctico [21] [20]. El laboratorio se organiza mediante puestos basados en Raspberry Pi, ya que estos dispositivos presentan un bajo coste y una amplia disponibilidad. Estas cualidades son necesarias para que el alumnado tenga la capacidad de trabajar de forma autónoma. Como parte de las prácticas se centran en el uso de dispositivos de entrada/salida se ha decidido utilizar el sistema operativo RISC OS. Se ha elegido este sistema operativo principalmente por dos motivos. Por una parte, RISC OS dispone de una interfaz gráfica que simplifica trabajar con dicho sistema. En segundo lugar, RISC OS permite manejar los dispositivos conectados a muy bajo nivel, pudiendo controlar la entrada o salida de cada uno de los pines de las Rasbberry Pi a los que se conecta el periférico sin apenas intervención por parte del sistema operativo.

Durante las prácticas, el alumnado debe desarrollar diferentes códigos escritos en lenguaje ensamblador. Para completar satisfactoriamente la práctica, el código resultante debe ser funcional, y debe cumplir todos los requisitos indicados en el enunciado. Como parte de los desarrollos, los alumnos deben hacer uso de una forma de memoria dinámica denominada "pila"(*stack*). Esta memoria se gestiona modificando el valor de un registro que indica la primera posición con contenido válido. La pila se emplea comúnmente para almacenar el estado de programas llamadores: cuando una función necesita utilizar registros con contenido no temporal, debe guardar una copia y luego restaurarla. Para el desarrollo de los códigos, se dispone de un editor de texto con resaltado de sintaxis que viene preinstalado en el sistema, llamado *StrongEd*. Los alumnos también deben realizar las tareas de ensamblado y enlazado del código con el fin de generar un fichero ejecutable, utilizando para este fin la suite de compilación *GCC* [9].

Sin embargo, no existe en RISC OS una herramienta de depuración gratuita que disponga de interfaz gráfica y cuyo uso sea relativamente sencillo. Por ese motivo, desde la Universidad de Cantabria se ha desarrollado un depurador que cumple con dichos objetivos, denominado UCDebug [7]. Un depurador es una herramienta que permite comprobar el funcionamiento de un programa, mostrando los cambios que genera durante la ejecución. El uso de un depurador nos ayuda a determinar la causa de posibles fallos de ejecución y nos permite observar si el código se ajusta a la funcionalidad pedida, como parte del proceso de desarrollo del código [19].

Cualquier programa ejecutado en RISC OS se enlaza en la dirección *0x8000* [16], ya que esta es la dirección donde comienza el espacio de direcciones del usuario en este sistema operativo. Sin embargo, el depurador es a su vez una aplicación que se está ejecutando, y por tanto comienza en la dirección *0x8000*. Para solucionar este problema se enlaza el código del alumno para que comience en una dirección que en el código del depurador corresponde a una variable definida en memoria. Dicha variable reserva un espacio en blanco sin ningún contenido válido, para después copiar el código del alumno en dicho espacio

y ejecutarlo desde el depurador. Esto obliga al usuario a enlazar su código en dos direcciones diferentes: una para su ejecución normal sobre el sistema operativo *(0x8000)* y otra para su ejecución dentro del depurador *(0x18000)*. Además de los pasos extra que se necesitan realizar, este diseño impide que se analicen en el depurador códigos ya compilados.

Es por tanto interesante explorar posibles modificaciones del depurador que permitan ejecutar códigos enlazados para comenzar en la dirección por defecto (0x8000). Eliminar dicha restricción no solo simplifica el proceso de depurado, sino que también posibilita el análisis de aplicaciones ya compiladas.

Una posible solución sería reposicionar la variable del depurador sobre la que se aloja el código del alumno, situándola al comienzo del espacio de direcciones de ejecución de las aplicaciones. De este modo, las direcciones de memoria de una aplicación ejecutada en el depurador y directamente sobre el sistema operativo pueden coincidir, y por tanto puede ejecutarse tanto de forma aislada como dentro del depurador. Para ello, es necesario realizar un análisis de la estructura de los códigos, tanto de los programas desarrollados por los alumnos como del depurador UCDebug.

1.1.1. Trabajo del alumnado en el laboratorio

Durante las prácticas, el alumnado debe desarrollar pequeños programas escritos íntegramente en lenguaje ensamblador. Estos códigos deben implementar la funcionalidad descrita en el enunciado de la práctica proporcionado por el profesorado.

Una vez desarrollado, los alumnos ensamblan los ficheros de código a través de la herramienta *"as"*, obteniendo de este modo un fichero objeto, tal y como se muestra en el *Listado 1*.

```
as -o objeto.o código.s
```

Listado 1: Uso del comando de ensamblado *"as"*

Como se ve en el listado, el alumno indica con el flag *"-o"* el nombre que quiere darle al fichero objeto resultante (en el ejemplo llamado "objeto.o"), y proporciona el fichero con el código ensamblador (en el ejemplo "código.s").

Por último, para obtener el fichero ejecutable, los alumnos hacen uso de la herramienta *"ld"*, con la cual pueden enlazar los diferentes ficheros en los que se ha desarrollado el código. De igual modo a como ocurre con la herramienta *"as"*, el alumno indica el nombre del ejecutable resultante por medio del flag *"-o"*.

```
ld -o ejecutable objeto1.o [objeto2.o] [...]
```

Listado 2: Uso de la herramienta *"ld"*

En el ejemplo de uso descrito en el *Listado 2* se ve como el alumno obtiene el fichero ejecutable (con nombre "ejecutable" en el ejemplo) indicando un conjunto de ficheros objeto.

Sin embargo, este ejecutable no es apto para su análisis en el depurador, ya que en el enlazado se toma por defecto la dirección *0x8000*, que como se ha explicado es la dirección de comienzo de ejecución en RISC OS. Para poder usar el depurador, el alumno debe especificar la dirección de enlazado del ejecutable. Concretamente se debe enlazar en la dirección *0x18000*, ya que como se explicará más adelante en el documento, esta es la dirección en la que el depurador almacena el código del alumno para su análisis. La nueva instrucción que debe ejecutar el alumno se muestra en el *Listado 3*, donde se especifica la dirección de enlazado a través del flag *"-Ttext"*.

```
ld -Ttext=18088 -o ejecutable objeto1.o [objeto2.o] [...]
```

Listado 3: Uso de la herramienta de enlazado *ld*, adaptada para generar ficheros compatibles con UCDebug.

1.2. Objetivos.

El objetivo principal de este proyecto reside en el análisis de la estructura del código máquina generado mediante la suite *GCC* en RISC OS, así como el tipo de contenido alojado en cada dirección del fichero. De este modo, se puede establecer una estrategia que permita reubicar segmentos de código en los ficheros binarios ejecutables, y comprobar su viabilidad. Dicha reubicación facilitaría la carga del código del alumno al comienzo del espacio de direcciones del usuario (0x8000), agilizando de este modo el proceso de depuración y permitiendo la depuración con la herramienta de aplicaciones que ya se encuentran traducidas a código máquina.

Como parte fundamental de esta tarea, se pretende realizar un estudio sobre la estructura de los códigos generados. Dicho estudio diferenciará los distintos segmentos que componen un fichero ejecutable obtenido mediante *GCC* sobre RISC OS a partir de programas desarrollados en lenguaje C y/o ensamblador. Dada la cuota de mercado minoritaria del sistema operativo RISC OS, y que la versión de *GCC* para este sistema es una adaptación no oficial, existe una cierta carencia de documentación sobre el comportamiento de la ejecución de programas. Por este motivo, para establecer la funcionalidad e impacto de algunos segmentos del código, es necesario hacer diversas pruebas empíricas sobre los ficheros ejecutables de un programa.

Se pretende por medio de este estudio colocar una sección con espacio reservado al comienzo del espacio de direcciones de memoria del usuario, de forma que se pueda utilizar para ubicar en este espacio el programa del alumno que se quiere depurar, de modo que las direcciones de enlazado entre una ejecución normal y una del depurador no varíen.

Para ello, además, será necesario comprender de que modo se estructuran los ficheros bajo el formato estandarizado ELF [5]. Este formato indica la estructura que debe tener un fichero objeto, como por ejemplo un ejecutable o una librería. Como se verá más adelante, estos archivos deben comenzar con una cabecera que indique que el fichero sigue este estándar, además de las ubicaciones en memoria de estructuras clave del fichero. Además se definen dos vistas diferentes desde las que interpretar el contenido del documento, una en secciones para la etapa de enlazado y otra en segmentos para la etapa de cargado.

El resto del documento está organizado como se indica a continuación. En primer lugar, se describe el análisis de las características de los ficheros ejecutables en RISC OS, los cuales siguen el formato estandarizado ELF. A continuación, se identifican los elementos en el depurador, estableciendo una metodología de trabajo y una serie de pruebas, tanto con el código del depurador completo como con códigos auxiliares. Posteriormente se describen los resultados de las diversas pruebas realizadas, y se determina la viabilidad de la estrategia de reubicación de segmentos de código. Asimismo, se propone una propuesta de cambio sobre el depurador que permite alcanzar el propósito que ha motivado este trabajo: eliminar la necesidad de un proceso de generación de programa específico para poder ejecutarlos sobre UCDebug. Finalmente, se resumen las conclusiones más importantes del trabajo.

2. Estructura del fichero ejecutable

El primer paso para valorar la viabilidad de estrategias de modificación de los segmentos del fichero de una aplicación es comprender de forma precisa cómo está organizada dicha aplicación. En el caso de RISC OS y de programas generados mediante la suite GCC, como es el caso del depurador UCDebug, dicho fichero ejecutable sigue el formato ELF.

El capítulo comienza con una breve explicación de este formato, considerando sus características y aplicaciones de uso. A continuación se analiza la estructura en bloques del fichero ejecutable, detallando la organización en distintas secciones y las diversas cabeceras que se emplean en el fichero.

2.1. El formato ELF

El formato ELF (Executable and Linkable Format, formato de enlazado y ejecución) describe la organización y estructura de un fichero que contiene un programa en código máquina. Este tipo de ficheros se denominan objeto de forma genérica, y existen tres tipos distintos según su funcionalidad y uso. El primero de ellos son los archivos reubicables, los cuales establecen todas las direcciones de memoria de forma relativa, de modo que el código se pueda alojar en cualquier posición de memoria (adecuadamente alineada) y seguir ejecutándose correctamente. Los archivos reubicables suelen emplearse como paso intermedio, siendo posible enlazarlos con otros archivos objeto para obtener un fichero ejecutable o un archivo de objeto compartido. Los ejecutables son ficheros objetos que contienen un programa que puede ser ejecutado desde el sistema operativo. Por último, los objetos compartidos, al igual que los archivos reubicables, contienen código y datos que emplean direcciones de memoria relativas. Sin embargo, a diferencia de los objetos reubicables, los ficheros compartidos pueden usarse como librería compartida, permitiéndose su uso desde ficheros ejecutables mediante un enlazador dinámico. El enlazador dinámico supone que, en el momento de lanzarse la ejecución del programa desde el SO, se determina si el código compartido está actualmente en memoria o si se carga se carga en caso contrario. Además, el SO se asegura de actualizar los campos del ejecutable cargado en memoria para emplear la dirección de memoria donde se encuentra el fichero compartido. Esto permite a diferentes procesos usar de forma simultánea el mismo código de un fichero objeto compartido.

Los tres tipos de ficheros objeto tienen una estructura común, y una serie de aspectos en los que difieren. En nuestro caso nos centraremos en la estructura particular de los objetos ejecutables, ya que es la categoría a la que pertenece la aplicación objeto de estudio.

2.2. Bloques del fichero ejecutable

El objeto ejecutable está dividido en tres componentes fundamentales: las cabeceras, los segmentos y las secciones. Las cabeceras contienen información relativa al propio objeto ejecutable. Dichas cabeceras pueden ser de tres tipos diferentes. Por un lado está la cabecera ELF, que identifica al fichero como un objeto ejecutable, y contiene datos como los tamaños de las diferentes estructuras en las que se subdivide el fichero, y sus direcciones de inicio. Las otras dos cabeceras presentes en el fichero son la tabla de cabeceras de programa y la tabla de cabeceras de sección. Estas cabeceras dividen el contenido del fichero en segmentos y en secciones respectivamente. Estas estructuras subdividen el mismo contenido, pero mostrando dos perspectivas diferentes de este. Por una parte los segmentos son utilizados por el cargador, e indican qué partes del ejecutable deben ser cargadas en memoria, y qué permisos de lectura y escritura tiene cada una. Las secciones son utilizadas por el linker, y sirven para agrupar el contenido del fichero de modo que la herramienta de enlazado pueda organizar estas secciones en el fichero ejecutable en memoria.

Algo importante a destacar es que el tamaño de las cabeceras no es constante para todos los programas. Programas sencillos, como lo son los desarrollados por los alumnos durante el transcurso de las prácticas, no poseen una estructura demasiado compleja. Esto provoca que el contenido del fichero se estructure en un número reducido de secciones y segmentos. Por otra parte, el depurador es un programa mucho más complejo, que, entre otras cosas, hace uso de diferentes librerías y recursos del sistema operativo. Además, el depurador define un mayor número de secciones en las que distribuir el fichero. Estas variaciones, acordes a la complejidad del código, provocan un número variable de entradas en las tablas de

las cabeceras, causando la variación del tamaño de las mismas. Por ejemplo, las cabeceras en los códigos desarrollados por el alumnado ocupan los primeros 136 bytes de memoria, mientras que las cabeceras del depurador ascienden a 232 bytes.

Los tres componentes de los ficheros ELF se ven representados en la *Figura 1*. En primer lugar se sitúa la cabecera ELF, de modo que los primeros bytes del fichero lo identifiquen como objeto ejecutable, seguidamente se encuentran las cabeceras de programa. A continuación se localiza el contenido del fichero en cuestión, colocándose al final del documento las cabeceras de sección. En la figura se resaltan mediante flechas verdes las referencias entre la tabla de cabeceras de sección y las distintas secciones (*".text"*, *".data"*, *".interp"*, *".symtab"*). A su vez, la tabla de cabeceras de sección se referencia desde la cabecera ELF, como se observa en la flecha dorada.

Como ya se ha comentado, esta es una de las posibles interpretaciones de la estructura del fichero. Desde el otro punto de vista vemos la subdivisión en segmentos. En la figura se observa un segmentos conformado por la sección *".interp"* y otro conformado por un conjunto de secciones, entre las que se encuentran las secciones *".text"* y *".data"*. La información relativa a cada segmento se almacena en la tabla de cabeceras de segmento (resaltado en azul), a la cual nuevamente se accede a través de la referencia almacenada en la cabecera ELF (resaltado en dorado). Esta interpretación en segmentos puede no incluir todo el contenido presente en el fichero. Determinadas tablas, las cuales no están presentes en memoria durante la ejecución, no forman parte de ningún segmento. Este caso se ve en la figura con la sección *".symtab"*.

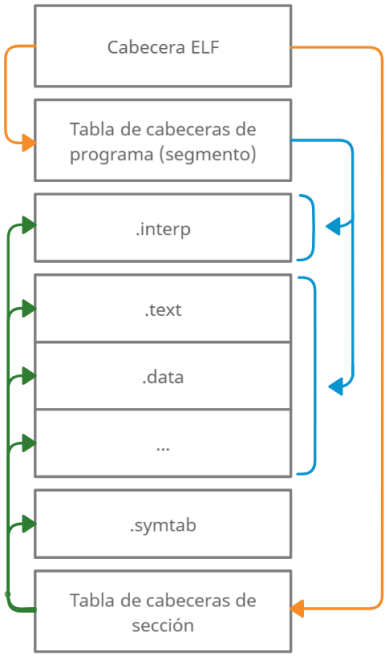

Figura 1: Estructura de un fichero con formato ELF

A continuación se van a analizar los elementos indicados en la figura, explicando la estructura del componente, así como qué función tiene dentro del fichero.

2.2.1. Cabecera ELF

El primer elemento con el que nos encontramos, situado al comienzo del espacio de direcciones, es la cabecera ELF, como se ha visto en la Figura 1. Esta cabecera indica que el fichero sigue la estructura

ELF, y detalla alguna de sus sus características; por ejemplo, la cabecera incluye un campo para indicar la arquitectura del procesador al que corresponde el código máquina. Además, esta cabecera reserva un espacio al final de los campos que la componen, de modo que pueda ser utilizado para nuevas aplicaciones en el futuro. La cabecera ELF también especifica aspectos estructurales como el número de segmentos y secciones que contiene el fichero, y el tamaño de cada uno de estos. En la Figura 2 se muestran los campos que componen la cabecera ELF.

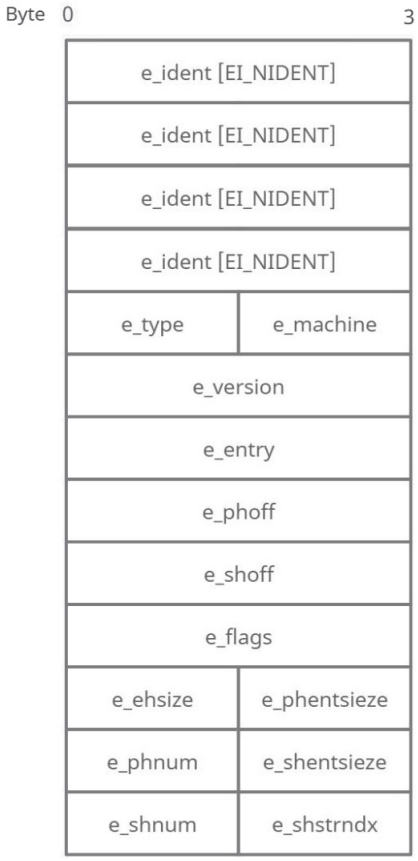

Figura 2: Cabecera ELF [5].

A continuación se describe el significado y características de los campos más representativos de la cabecera ELF:

- *e_ident:* Este campo identifica el objeto como un fichero elf. El campo en cuestión está conformado por una cadena de 16 caracteres, aunque no todas almacenan un valor válido.

  Las 4 primeras posiciones de la cadena de caracteres conforman lo que se denomina el "número mágico", una secuencia de 4 dígitos que identifica que el fichero sigue la estructura ELF. Los cuatro dígitos de este código mágico toman el valor 0x7F seguido de la representación de los caracteres 'E', 'L', 'F' en la codificación ASCII [1]. Esto facilita la identificación para un humano cuando se utilizan editores que muestran la interpretación ASCII del contenido de un fichero. La quinta posición indica el número de bits de la arquitectura soportado, con valor 1 para arquitecturas de 32 bits y 2 para arquitecturas de 64 bits. La sexta posición se usa para identificar el formato de datos utilizado, con valor 1 para little endian y valor 2 para big endian. La última posición indica el comienzo de los

bytes no utilizados de este campo, variando de valor a medida que se vayan especificando nuevos significados para los bytes de este campo.

- *e_type.* Este campo identifica el tipo de fichero objeto. Los tipos disponibles son: objeto reubicable (valor 1), objeto ejecutable (valor 2), objeto compartido (valor 3), objeto del núcleo (valor 4), o dos posibilidades para objetos dependientes del procesador (valores *0xFF00* y *0xFFFF*).

- *e_entry.* En este campo se almacena la dirección donde se sitúa el punto de entrada, es decir, la dirección a la que se transfiere el control una vez comience la ejecución del programa.

- *e_phoff.* Este atributo contiene el desplazamiento a la tabla de cabeceras de programa (program header table offset). Esta tabla contiene en sus entradas información relativa a la ubicación y contenido de los segmentos del fichero. Su contenido se describe en la *Sección 2.2.5*.

- *e_shoff.* Este atributo contiene el desplazamiento a la tabla de cabeceras de sección (section header table offset). Esta tabla contiene en sus entradas información relativa a la ubicación y contenido de los segmentos del fichero. Su contenido se describe en la *Sección 2.2.3*.

- *e_ehsize.* Este campo contiene el tamaño en bytes de la cabecera elf (ELF header size). El estándar define una serie de campos concretos comunes a todos los ficheros y arquitecturas que ocupan en total 52 bytes, pero la cabecera puede incorporar campos adicionales, en cuyo caso el campo "e_ehsize" será superior. Además del espacio ocupado por estos campos, se reserva al final de la cabecera un espacio que puede ser utilizado posteriormente.

- *e_phentsize.* En este atributo se almacena el tamaño en bytes de cada una de las entradas de la tabla de cabeceras de programa. Todas las entradas de la tabla tienen siempre la misma longitud.

- *e_phnum.* En este campo se encuentra el número de entradas de la tabla de cabeceras de programa.

- *e_shentsize.* En este atributo se almacena el tamaño en bytes de cada una de las entradas de la tabla de cabeceras de sección. Al igual que las cabeceras de programa, todas las entradas de la tabla de cabeceras de sección comparten la misma longitud.

- *e_shnum.* En este campo se encuentra el número de secciones en las que está estructurado el fichero, y por tanto el número de entradas en la tabla de cabeceras de sección.

Los campos descritos han presentado una gran utilidad durante el desarrollo del proyecto, por un lado los campos *e_ehsize*, *e_phentsize*, *e_phnum*, *e_shentsize* y *e_shnum* permiten calcular el tamaño de las diferentes partes en las que se divide el fichero ejecutable. Esta información, combinada con la obtenida de los campos *e_phoff* y *e_shoff*, permite crear un mapa preciso de la estructura del fichero. Además, se ha trabajado mucho con el punto de entrada del código, el cual se encuentra en el campo *e_entry*, ya que durante las pruebas desarrolladas se ha modificado este valor de forma ocasional.

El resto de campos presentes están descritos dentro de la documentación perteneciente al estándar ELF, incluida en la bibliografía [5].

2.2.2. Secciones

Las secciones son los bloques del fichero ELF en los que se estructuran los contenidos del programa. A excepción de la cabecera ELF y las tablas de cabeceras de programas y de secciones, todo la información del fichero está estructurada en secciones. Estas secciones trocean la memoria en fragmentos contiguos, catalogando el contenido del ejecutable en función de su naturaleza. Las secciones se caracterizan por definir una secuencia de bytes contigua en el fichero, sin solaparse entre sí. Cada sección está además identificada mediante una entrada en la tabla de cabeceras de sección, en la que se establece el propósito y características de la misma.

Gracias a estas secciones, la herramienta de enlazado es capaz de conformar el fichero ejecutable. Como se verá más adelante, el enlazador analiza los ficheros objeto de entrada en busca de las diferentes secciones que tengan definidas. Posteriormente, coloca las secciones de los ficheros de entrada en un único fichero de

salida, siguiendo un esquema que tiene predefinido en lo que se conoce como *"linker script"*. Este proceso determina de qué modo se estructura el programa en memoria. Cada sección esta definida por su cabecera, donde se registran, entre otros valores, su dirección de inicio, el tamaño de la misma y su nombre (el cual distingue mayúsculas de minúsculas). Estas cabeceras se almacenan en forma de tabla al final del fichero, y es posible acceder a ellas gracias a que su posición esta registrada en el campo *e_shoff* de la cabecera ELF.

En la *Figura 1* se muestran tres secciones distintas: *".text"*, *".data"*, *".rodata"*. Existen múltiples secciones definidas por el estándar, siendo algunas de ellas específicas de cada arquitectura de procesador o del sistema operativo. La sección *".text"* está concebida para agrupar todo el conjunto de instrucciones a lo largo del programa, mientras que las secciones *".data"* y *".rodata"* almacenan todos los datos definidos en el código, con la peculiaridad de que la sección *".rodata"* almacena los datos con permisos de solo lectura.

Además de declarar el comienzo de secciones ya definidas por el estándar, es posible la definición de secciones personalizadas, agrupando de este modo partes del código según conveniencia. En el *Listado 4* se muestra cómo se declara una sección de variables en memoria, y otra de código, que se corresponden con las secciones *.data* y *.text* respectivamente.

```
1 .data
2     .align 2
3     dirretorno: .space 10
4     guardado:   .space 16
5 .text
6 .align 2
```

Listado 4: Ejemplo de declaración de secciones de un programa ensamblador.

Salvo que se especifique lo contrario, todas las secciones definidas bajo un mismo nombre se agruparán bajo una única sección dentro del fichero ejecutable. Este proceso se realiza durante la etapa de enlazado, en el que se resuelven las referencias a los diferentes símbolos del programa.

El modo de resolver estas referencias depende tanto de la arquitectura del procesador como del sistema operativo del equipo destino sobre el que se va a ejecutar el programa, aunque este puede ser catalogado dentro de dos modelos de enlazado: estático y dinámico. Se utiliza un enlazado estático cuando el recurso es específico para el programa que se ha desarrollado, y se enlazará de forma dinámica aquellos recursos que sean compartidos por varios procesos, como las librerías del sistema.

Como se ha mencionado, las secciones *".text"* y *".data"* son la estructura fundamental de cualquier programa, ya que corresponden a los bloques de instrucciones en código máquina y de declaración de variables en memoria, respectivamente. No obstante, podemos hallar otra serie de secciones que se encuentran normalmente presentes en un fichero ejecutable estándar. En cuanto a secciones que contienen datos, encontramos además de *".data"* otras secciones como *".rodata"* (para datos de solo lectura) y *".bss"* (para datos no inicializados). Existen secciones que no se suelen incluir en la imagen en memoria del proceso, pero que son útiles para tareas como el depurado del mismo. En este grupo encontramos las secciones *".comment"*, *".line"*, *".debug_aranges"*, *".debug_info"*, *".debug_abbrev"* y *".debug_line"*.

Una última característica de las secciones es que estas pueden ocupar diferentes cantidades de memoria cuando nos referimos al binario o cuando hablamos del fichero que se está ejecutando. Principalmente se debe a que ciertas secciones, debido a su naturaleza, no necesitan ocupar espacio hasta el momento de la ejecución. Un ejemplo de esto es la sección *".bss"*, dedicada a los datos sin inicializar. Durante la ejecución, las variables no inicializadas necesitan estar ubicadas en memoria para poder hacer uso del espacio que estas reservan, pero en el binario no tiene sentido desaprovechar memoria en variables que

no contienen un valor válido.

Esta situación se da en códigos complejos, como son el depurador o programas desarrollados en lenguajes de alto nivel o de forma híbrida (combinando por ejemplo C y ensamblador), pero no es algo común en los códigos de los alumnos, ya que estos presentan una menor complejidad que los citados.

Como ya se ha mencionado al principio del capítulo, códigos más complejos cuentan con un mayor número de secciones. Para ilustrar esto se va a exponer una comparativa entre las secciones del código que pudiera desarrollar un alumno y las del propio código del depurador, comenzando con el caso más sencillo de los dos.

En primer lugar, se han analizado las secciones presentes en un programa ensamblador de funcionalidades básicas, en las que se imprime un valor por consola (a modo de "hola mundo"). Este programa está compuesto por únicamente 6 secciones (7 si contamos a la sección nula). En primer lugar encontramos las secciones *".text"* y *".data"*; aunque podríamos incluso prescindir de *".data"* si no se hiciese uso de una sección de datos. Como se ha expuesto, estas son las secciones más básicas presentes en un ejecutable. Las 4 secciones restantes están presentes únicamente en el binario generado, pero no se cargan en memoria durante la ejecución. De estas, 3 corresponden con las secciones de tablas *".shstrtab"* (almacena los nombres de las secciones), *".symtab"* (almacena la tabla de símbolos) y *".strtab"* (almacena la tabla de strings). Estas estructuras se describen de forma más detallada al final del capítulo. La sección restante es dependiente de la arquitectura sobre la que se ejecuta el programa, y se denomina *".ARM.attributes"*.

Por otra parte, el depurador se estructura en 27 secciones (28 contando con la sección nula). La causa de esto es que es un código mucho más complejo que hace uso de librerías presentes en el sistema operativo y combina el uso de varios lenguajes de programación. Además de las ya presentes en el código del alumno, en el depurador se encuentran secciones propias de las librerías de C, como por ejemplo las secciones *".ctors"* y *".dtors"*, donde se ubican los constructores y destructores de C, o las secciones *".init"* y *".fini"*, desde las que comienza y termina la ejecución de la aplicación.

Además, como se muestra en la *Sección 4.1*, durante la etapa de enlazado del depurador se hace uso de librerías dinámicas. Estas librerías son compartidas por varios procesos, de modo que es necesario resolver sus referencias en tiempo de ejecución, en lugar de durante la etapa de enlazado. De este modo, además de las secciones *".strtab"* y *".symtab"* se declaran las secciones *".dynstr"*, *".dynsym"* y *".dynamic"*, las cuales si que se cargan en memoria durante la ejecución.

Las secciones mencionadas constituyen sólo una parte de la estructura del depurador. Sin embargo, su contenido ha mostrado tener mayor relevancia durante el desarrollo de este proyecto. Información que si podría ser relevante, como la dirección en la que se sitúa cada una de ellas y el tamaño que tienen, se encuentra registrado en las cabeceras de sección. Estas estructuras vienen descritas en el siguiente apartado.

2.2.3. Cabeceras de sección

Para poder encontrar las diferentes secciones en memoria hace falta recurrir a unas estructuras que contengan datos fundamentales de las mismas. Las cabeceras de sección contienen entonces toda la información necesaria para delimitar la ubicación de una sección en memoria, así como datos extra referentes al contenido que se almacena en ella.

Estas cabeceras de sección se localizan en la parte inferior del espacio de memoria ocupado, como se muestra en la *Figura 1*. El conjunto de las cabeceras se agrupan formando una tabla, de modo que cada entrada contiene la cabecera de una sección. La dirección de comienzo de esta tabla está indicada en la cabecera ELF, así como el número de entradas de la tabla, y el tamaño de cada una. La estructura de una cabecera de sección tiene los siguientes campos, indicados en la *Figura 3*.

Figura 3: Entrada de la tabla de cabeceras de sección.

- *sh_name.* Este atributo especifica el nombre de la sección. El valor almacenado es el índice correspondiente al nombre de la sección almacenado en la tabla de strings.
- *sh_type.* Este miembro cataloga el contenido de la sección en diferentes tipos, indicando además la semántica del mismo. Los tipos más comunes, y los utilizados por nuestros programas, son los siguientes:
  - *SHT_PROGBITS.* Este tipo de sección contiene información definida por el propio programa, el formato y significado de su contenido depende únicamente del programa.
  - *SHT_HASH.* Las secciones de este tipo contienen una tabla de hash.
  - *SHT_DYNSYM* y *SHT_SYMTAB.* Las secciones de este tipo contienen una tabla de símbolos.
  - *SHT_REL.* Este tipo de secciones contienen entradas de reubicación.
  - *SHT_DYNAMIC.* Este tipo de secciones contienen información necesaria para el enlazado dinámico.
  - *SHT_NOBITS.* Este tipo de secciones se comportan igual que las secciones de tipo *SHT_PROGBITS*, con la salvedad de que no ocupan espacio en el fichero objeto.
  - *SHT_STRTAB.* Las secciones de este tipo contienen una tabla de strings.
- *sh_addr.* Este campo especifica la dirección de memoria en la que se debe situar el primer byte de la sección en caso de que esta forme parte de la imagen de memoria del proceso.
- *sh_offset.* Este campo contiene el desplazamiento en memoria a la sección, es decir, el número de bytes desde el comienzo del espacio de direcciones al primer byte de la sección. Las secciones de tipo *SHT_NOBITS* guardan un valor no nulo.
- *sh_size.* Este campo contiene el tamaño de la sección en bytes. Las secciones de tipo *SHT_NOBITS*, pese a no ocupar espacio en el fichero, tendrán un valor no nulo.

Durante el trabajo se ha utilizado estos campos en concreto para identificar las secciones presentes en los códigos analizados. Cada sección ha sido identificada por su nombre, y situada dentro del fichero gracias al campo *sh_addr* (o en su defecto *sh_offset*) y al campo *sh_size*. Además, ha sido posible obtener una primera interpretación de la utilidad de su contenido gracias al campo *sh_type*.

El resto de los campos no resultan de utilidad para el propósito de este proyecto. No obstante, la aplicación que tiene cada uno de ellos viene descrita en la documentación del estándar ELF, presente en la bibliografía [5].

2.2.4. Segmentos

Al igual que las secciones, los segmentos representan un punto de vista de la estructura del objeto. En este caso los segmentos son utilizados por el cargador, ya que agrupan el contenido del fichero en función de características como si deben o no deben ser cargados en memoria, o en caso de hacerlo, qué permisos de acceso se tiene sobre ellos. Es por esto que los segmentos más representativos de un ejecutable son del tipo *PT_LOAD*, ya que en su interior se agrupan las partes del fichero que deben ser cargadas en memoria, como datos e instrucciones del programa. Los códigos de los alumnos, dada su sencillez, solo pueden tener segmentos de tipo *PT_LOAD*. Por otra parte, contenidos más orientados a, por ejemplo, la depuración del programa o tablas de enlazado estático no necesitan estar presentes en memoria durante la ejecución del código, de modo que por medio de los segmentos el cargador evita copiarlos en memoria.

A diferencia de las secciones, los segmentos no están necesariamente contiguos en memoria, y pueden estar contenidos unos dentro de otros. La posición y tamaño de los diferentes segmentos se conoce gracias a la tabla de cabeceras de programa. De igual forma a la tabla de cabeceras de sección, esta tabla contiene para cada segmento una entrada con datos referentes a la posición en memoria, tamaño, o tipo de cada segmento. La ubicación de esta tabla también está registrada en la cabecera ELF.

2.2.5. Cabeceras de programa (segmento)

Como se ha mencionado con anterioridad, estas cabeceras definen segmentos usados por el sistema operativo en la fase de preparación del programa para la ejecución. Cada entrada de la tabla de cabeceras sigue la estructura representada en la figura *Figura 4*. A continuación, se detallan dichos campos:

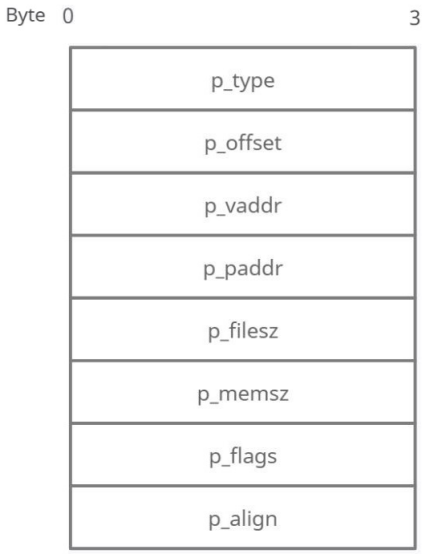

Figura 4: Entrada de la tabla de cabeceras de segmento.

- *p_type.* Este miembro indica el tipo de segmento descrito por la entrada, junto a cómo interpretar los campos de la estructura. Los tipos de segmentos existentes son los siguientes:
  - *PT_LOAD* indica que el segmento agrupa contenido cargable. Los bytes del fichero se mapean al comienzo del segmento de memoria.
  - *PT_DYNAMIC* indica información referente al enlazado dinámico.
  - *PT_INTERP* indica la localización de la ruta desde la cual se llamará al intérprete.
  - *PT_NOTE* indica la localización y tamaño de información auxiliar.
  - *PT_PHDR* indica la ubicación de la propia tabla de cabeceras de programa.
- *p_offset.* La entrada indica la ubicación de la propia tabla de cabeceras de programa.
- *p_vaddr.* Este miembro indica la dirección virtual del primer byte del segmento correspondiente a la cabecera.
- *p_paddr.* Este campo solo aparece en sistemas en los que el direccionamiento físico sea relevante de cara al enlazado, almacenando la dirección física del comienzo del segmento.
- *p_filesz.* Este miembro almacena el número de bytes ocupados por el segmento en el fichero objeto.
- *p_memsz.* Este miembro almacena el número de bytes ocupados por el segmento una vez cargado en memoria. Este valor intuitivamente debería ser igual al tamaño ocupado por el segmento en el fichero *(p_filesz)* Sin embargo, secciones del tipo *SHT_NOBITES* (como la sección *".bss"* dedicada a datos sin inicializar) no ocupan espacio en el fichero, pero sí en memoria.

2.2.6. Tablas auxiliares

Existen varias estructuras, usualmente en forma de tabla, que se utilizan en todo el proceso de referenciación entre partes del código. Las estructuras más destacables son la tabla de *"strings"* (cadenas de caracteres) y la tabla de símbolos.

Cada entrada de las tablas de cabeceras de sección y de segmento tienen un tamaño fijo, indicado como se ha visto en los campos *e_phentsize* y *e_shentsize* de la cabecera ELF. No obstante, cada entrada corresponde con un segmento o sección, los cuales tienen un nombre cuya longitud es variable. La forma de mantener el tamaño de las cabeceras constante consiste en almacenar los nombres de las secciones y segmentos en una segunda tabla, denominada tabla de *"strings"* [13], y almacenar en la cabecera un índice a esta tabla. Los índices, a diferencia de los nombres, ocupan el mismo espacio en las diferentes entradas de la tabla de cabeceras. De este modo es posible hacer uso de nombres en las secciones y segmentos manteniendo constante el tamaño de las cabeceras. Esta misma estrategia se sigue en otras estructuras de datos, como por ejemplo la tabla de símbolos.

La tabla de símbolos [13] contiene toda la información necesaria para localizar y reubicar todas las definiciones y referencias simbólicas dentro del programa (como por ejemplo la llamada a una función). Estas tablas las podemos encontrar en las secciones *".symtab"* (no presente en la imagen de memoria del programa) y *".dynsym"* para el caso de la tabla de símbolos dinámica (presente en la imagen en memoria del programa, usada durante la ejecución) [3].

Cada símbolo representa una referencia dentro del código. Durante el desarrollo del programa estas referencias no pueden ser resueltas, ya que no se conoce cómo se va a estructurar el código en memoria, ni por tanto la dirección en la que se situará cada símbolo. Por este motivo, a cada referencia se le asigna el nombre identificativo del lugar al que apuntan. Durante el enlazado se resuelven las referencias estáticas, ya que estas no pueden variar su posición en memoria durante la ejecución. Por este motivo no es necesario mantener la tabla de símbolos estática (*".symtab"*) en memoria mientras se ejecuta el código.

Por otra parte, los objetos dinámicos, como las librerías del sistema operativo, pueden ser compartidos por varios procesos de forma simultánea. Esto provoca que su posición en memoria pueda cambiar a lo

largo del tiempo. Por este motivo, este tipo de referencias no pueden ser resueltas durante la fase de enlazado, y deben ser resueltas en tiempo de ejecución. Estas referencias se almacenan entonces en la tabla de símbolos dinámica (*".dynsym"*), que sí se carga en memoria al ejecutar el programa.

Para agilizar la búsqueda en estas tablas dinámicas se hace uso de una tabla denominada tabla de hash de símbolos [2] [6], situada en la sección *".hash"*. La tabla de hashes dispone un número fijo de ranuras o *"buckets"*, en las cuales se almacenan índices de la tabla de símbolos. Para cada símbolo se determina mediante un algoritmo el *"bucket"* de la tabla en el que almacenar su índice. De este modo, si se quiere conocer el índice de un determinado símbolo solo es necesario recorrer las entradas de uno de los *"buckets"*, en vez de toda la tabla de símbolos.

3. Análisis del código

En este capítulo se explican los procedimientos seguidos a la hora de desarrollar las diferentes pruebas a lo largo del proyecto. Además, se realiza un análisis del código del depurador con el objetivo de comprender de forma precisa cómo se llevan a cabo las operaciones susceptibles de cambio durante el desarrollo del proyecto, y entender de qué modo funciona internamente la herramienta. Por último, se explica la estructura que siguen los diferentes programas desarrollados durante la fase de pruebas.

3.1. Estrategia seguida

Para comprender el funcionamiento de la aplicación ha sido necesaria la realización de pruebas y análisis específicos del código. No obstante, el depurador es una aplicación compleja y de gran extensión, lo que dificulta establecer de forma precisa los resultados obtenidos en cada prueba. Para agilizar el análisis de la funcionalidad de cada bloque en el fichero ejecutable, se ha empleado un código más sencillo y de funcionalidad limitada, pero que mantiene la misma estructura que el código del depurador. Posteriormente se han ratificado las hipótesis establecidas con pruebas sobre la aplicación completa.

La estrategia seguida ha sido dividir el proyecto en dos fases. La fase inicial se centra en establecer y replicar aspectos clave de la estructura del depurador en programas más sencillos. Sobre estos programas más simples (también llamados "de juguete") se han realizado pruebas de cara a determinar la viabilidad de diferentes estrategias para alojar una sección en blanco al comienzo del programa.

La segunda fase consiste en determinar si se pueden trasladar los cambios realizados sobre el programa de pruebas al código del depurador. En esta segunda fase es necesario realizar diversas comprobaciones. La primera de ellas es cerciorarse de que los cambios se apliquen de igual manera en el depurador y en el programa de pruebas. Aunque el código de pruebas se diseñe para tener una estructura semejante al depurador, la complejidad de dicha aplicación y el uso de librerías pueden generar resultados diferentes a los obtenidos inicialmente. Además, es necesario verificar que el funcionamiento del depurador no se vea afectado, de modo que se pueda seguir empleando para depurar los códigos de los alumnos.

3.2. Metodología de trabajo

Acorde a la estrategia mencionada para abordar el proyecto, la organización del trabajo está estructurada en la siguiente sucesión de pasos. En primer lugar, se procede a un análisis de las diferentes partes en las que se divide un ejecutable, así como del contenido y significado de sus cabeceras. Esta fase está marcada por la documentación y búsqueda de información en manuales y artículos, y la información recopilada se ha detallado en el Capítulo *2*. La documentación específica de la arquitectura y sistema operativo empleados no es completamente exhaustiva, por lo que también se ha analizado documentación pertinente a otros casos de uso, como la arquitectura x86 bajo el sistema operativo Solaris [14].

El segundo paso consiste en aplicar el conocimiento obtenido a los casos concretos con los que se trabaja. Se pretende encontrar el número de secciones en las que se divide el programa, así como su contenido. Se busca además delimitar el rango de direcciones en el que se sitúan las diferentes secciones a lo largo del espacio de memoria. Para extraer esta información se hace uso del decodificador de cabeceras ELF [12]. Esta herramienta online extrae el contenido de las cabeceras del ejecutable presentando para cada campo el valor encontrado. La información generada se puede obtener también mediante un análisis del contenido del fichero en formato hexadecimal a través de un editor de textos, como StrongEd [17] o Notepad++ [15], pero el uso del decodificador de cabeceras ELF agiliza el proceso.

El siguiente paso se basa en el desarrollo de un programa sencillo con características semejantes al depurador en lo referente a su organización en memoria: programación híbrida en lenguaje ensamblador ARM y C, y uso de librerías del sistema. A continuación, se realizan pruebas de reposicionamiento de las secciones del programa. En caso de no ser posible reubicar alguna de las secciones, se realizan pruebas de "pisado"de dichas secciones, sobrescribiendo el contenido de sus direcciones de memoria. El "pisado"de secciones pretende replicar la acción de alojar código del alumno sobre secciones del binario del depurador,

y se realiza insertando valores aleatorios. El objetivo es determinar si el funcionamiento del programa se mantiene pese al borrado de partes del mismo.

Las pruebas de pisado están motivadas por la falta de documentación de algunos apartados, ya que la documentación disponible no indica de forma exhaustiva los contenidos y funcionalidad de todos los bloques del fichero ejecutable. Por medio de estas pruebas no solo se trataría de colocar un bloque de espacio libre al comienzo del ejecutable, sino que además se podría determinar qué secciones no son de obligada presencia en el código del alumno. Con esta información se podría limitar el copiado del código del alumno a las secciones indispensables, ya que actualmente el depurador copia el bloque del fichero por completo, sea necesario para su ejecución o no.

El último paso consiste en trasladar todos los cambios propuestos al código del depurador, repitiendo las pruebas relacionadas con dichos cambios. Con estas pruebas se garantiza que los resultados logrados en los códigos "de juguete" se mantienen en el programa final, sin afectar a la funcionalidad del depurador: ejecutar de forma controlada código desarrollado por el alumno.

3.3. Análisis del código del depurador

Para entender la estructura del fichero binario del depurador es necesario comprender cómo está programada la herramienta, así como el flujo de control de la CPU durante la ejecución del programa.

3.3.1. Estructura del código

El código del depurador se estructura en dos bloques que se ejecutan de forma conjunta: la interfaz gráfica y el núcleo del depurador. Estos bloques están desarrollados en dos lenguajes de programación distintos: C y ensamblador ARM, respectivamente. De forma adicional, durante la ejecución del depurador se añade un tercer bloque compuesto por el código del alumno a depurar, el cual se ejecuta como si fuese parte del propio depurador. Los códigos desarrollados por los alumnos están siempre desarrollados en ensamblador ARM.

El núcleo del depurador está conformado por un grupo de funciones que se ejecutan en torno al código del alumno, con la ayuda de varios manejadores de excepciones que permiten al depurador retomar el control de la ejecución y mostrar diferentes errores de ejecución del programa del alumno. Para alcanzar la granularidad de ejecución necesaria para ofrecer información precisa del estado de la CPU, el núcleo del depurador se ha programado en lenguaje ensamblador ARM.

La interfaz gráfica es la parte del depurador con la que puede interaccionar el usuario para hacer uso de la herramienta. A través de ella, se muestra toda la información relevante para el depurado, como contenido de registros, contenido de memoria, instrucciones a ejecutar y otros datos acerca del estado de la ejecución. También permite al usuario establecer puntos de ruptura a lo largo del código con el fin de detener la ejecución en el momento en el que se alcancen, cargar ejecutables para su análisis, y seleccionar el modo de ejecución del código a depurar (el depurador ofrece varios modos de ejecución, según el nivel de intervención que se desea por parte de la herramienta). Esta parte de la herramienta está programada en lenguaje C, para agilizar su desarrollo y facilitar el uso de librerías comunes, como las relativas a la interfaz de ventanas del sistema. Además, representa el comienzo de la ejecución de la aplicación, ya que esta parte incluye la definición del programa principal (*main*). Se puede plantear invertir esta estrategia, haciendo que el programa principal esté escrito en lenguaje ensamblador ARM y desde él se llame al código en C. Durante la fase posterior de pruebas se analizarán ambos enfoques, para determinar el más efectivo de cara a la reubicación de segmentos.

El código del alumno es el conjunto de instrucciones que se quiere depurar. Para cumplir este cometido, el depurador hace una copia del ejecutable del alumno a su propio espacio de memoria, desde donde lanza su ejecución. Es necesario que el núcleo del depurador recupere el control del procesador en el transcurso de la ejecución del código del alumno. RISC OS es un sistema operativo cooperativo, en el que los programas deben renunciar al uso de la CPU para permitir la ejecución de otras tareas. Para

detener la ejecución del código del alumno y retornar al depurador se genera una excepción insertando un punto de ruptura en el código del alumno. Estos puntos de ruptura se introducen mediante una instrucción específica *(0xE1200071)* conocida como *BKPT 1* definida en el ISA de ARM [10]. Al ejecutarse esta instrucción, el procesador lanza una excepción conocida como *"prefetch abort"* interrumpiendo la ejecución del código. El depurador instala un manejador propio para esta excepción, de modo que al producirse la excepción recupera el control de la ejecución. Antes de comenzar a ejecutar el código del alumno, el depurador sustituye el contenido de las direcciones de memoria donde se quiera recuperar el control por esta instrucción. Con el fin de preservar el contenido original de las direcciones, los valores previos se almacenan en una tabla junto a la dirección en la que estaban almacenados. Tras producirse la excepción, el depurador emplea la información almacenada en la tabla para restaurar el contenido original del programa del alumno.

3.3.2. Manejo de excepciones

Como se expone en la *Sección 1.1*, un caso común en los códigos de alumno es la existencia de errores que impiden completar su ejecución de forma exitosa. Dichos errores van a producir una excepción en el depurador; por ejemplo, si se accede a una dirección de memoria que requiere permisos especiales, se produce un *"Data abort"*. Al producirse la excepción se detiene la ejecución del código que la ha provocado y se pasa a ejecutar el manejador definido para tratar dicha excepción. Por defecto, el sistema operativo define una serie de manejadores para cada tipo de excepción existente en la arquitectura del procesador. En la mayoría de casos, esto supone la finalización del programa con errores. Sin embargo, para el sistema operativo, el código del alumno es parte del programa depurador, por lo que una excepción en el programa del alumno terminaría la ejecución de UCDebug. Para corregir este comportamiento, el depurador reemplaza algunos de los manejadores de excepciones por otros propios, asegurando que el tratamiento de las excepciones se adecúe a las necesidades particulares de la herramienta. De este modo el depurador identifica qué fallos produce el código que se pretende depurar, y puede informar al usuario de su presencia por medio de un mensaje de error.

Es por tanto necesario que el depurador defina un total de 5 manejadores que eviten que RISC OS interfiera en la ejecución del código si el fallo lo produce el programa del alumno. Estos manejadores se asignan a las siguientes excepciones:

- *Prefetch abort:* Esta interrupción se genera en caso de que la instrucción a ejecutar no pueda obtenerse de la dirección de memoria correspondiente, o en caso de ejecutar una instrucción BKPT. Permite recuperar el control de la ejecución al núcleo del depurador, y en caso de que haya un fallo para obtener la instrucción muestra un mensaje de error al alumno.
- *Instrucción no definida y data abort:* Estas interrupciones son comunes durante la ejecución de códigos de estudiantes, donde es propenso encontrarse con accesos a direcciones de memoria erróneas. Su tratamiento por parte del depurador, además de evitar finalizar su ejecución, proporciona información al alumnado para su corrección.
- *Peticiones de interrupción (IRQ):* Estas interrupciones son generadas por diferentes eventos, especialmente cuando se está esperando en operaciones de Entrada/Salida (por ejemplo, a través un dispositivo hardware). Es tarea del depurador comprobar si hay un manejador definido por el alumno para la interrupción que se ha producido. En caso de no haberlo, se deriva la interrupción a los manejadores propios de RISC OS.
- *Interrupciones por software (SWI):* Estas interrupciones son también conocidas como llamadas al sistema. Se invocan desde el propio código con el fin de solicitarle al sistema operativo que realice un servicio. El uso de excepciones para gestionar dichas llamadas evita tener que mantener todos los criterios del convenio de uso de registros en llamadas a funciones, evita tener que conocer la dirección de salto en el código, y permite la ejecución de código del sistema operativo en forma privilegiada. El manejador propio del depurador para excepciones SWI se interpone para cualquiera de las instrucciones SWI, ya que el comportamiento de este tipo de operaciones necesita ser adaptado

al funcionamiento del depurador. Por ejemplo, la SWI empleada para indicar al sistema operativo la finalización del código debe interceptarse para que no detenga la ejecución del depurador. La gestión de salida de caracteres por pantalla también debe tratarse, ya que el depurador emplea una ventana propia al efecto. En caso de que la SWI no necesite ser adaptada a la ejecución en el depurador, se deriva al manejador del sistema operativo, ejecutándose la rutina original para la interrupción.

3.3.3. Modos de depuración

Existen diferentes modos de ejecución en función de la interacción entre el código del alumno, el núcleo del depurador, y la interfaz de usuario. Estos son el modo traza, el modo de ejecución normal, los modos *"go to"* y *"go direct"*, y finalmente el modo *"go fast"*. En todos los casos, salvo en el modo *"go fast"*, el núcleo del depurador recupera el control de la ejecución tras ejecutar una sola instrucción del código del alumno.

- *Modo traza:* Este modo permite una depuración detallada del código, ejecutándose una única instrucción antes de devolver el control al usuario, regresando el flujo de ejecución de la CPU al código de la interfaz.
- *Modo Normal (Go):* Este modo de ejecución se basa en la ejecución continua del código, devolviendo el control al usuario únicamente cuando se alcanza un punto de ruptura, se genera una excepción o se finaliza el código.
- *Modo Go-Direct y modo Go-To:* Estos modos están ideados para ser usados de forma complementaria al modo anterior, con la única diferencia de que estos modos no ceden el control al usuario en caso de llegar a un punto de ruptura.

  La idea tras estos modos es evitar tener que eliminar los breakpoints si se tiene intención de seguir haciendo uso de ellos en un futuro. Ambos modos se detienen igualmente en el caso de que se genere una excepción, pero el modo Go-Direct está pensado para ejecutar el código completo, mientras que el modo Go-To recibe una dirección como argumento, en la cual detendrá la ejecución del código del alumno y cederá el control al usuario.
- *Modo Go-Fast:* Este modo permite una ejecución rápida del código, ya que evita que el núcleo del depurador recupere el control en cada instrucción del código alumno. Esto puede resultar útil a la hora de depurar aplicaciones con altos requisitos temporales, como puede ocurrir en aplicaciones con operaciones de entrada/salida.

  Para permitir esta ejecución rápida, el depurador cambia el proceso de recuperación del flujo de ejecución desde el código del alumno al núcleo del depurador. En lugar de emplear la instrucción BKPT1 para asegurar que se ejecute una sola instrucción del código del alumno, se emplea un temporizador del sistema operativo. Este temporizador genera una excepción de tipo interrupción, para garantizar que el flujo de ejecución no quede acaparado por el código del alumno en ausencia de fallos de ejecución (excepciones) y llamadas a servicios (SWI). De forma periódica se devuelve puntualmente la ejecución a la interfaz de usuario, para que éste pueda introducir comandos.

3.3.4. Carga de código del alumno

El proceso de cargar el código del alumno en el depurador se divide principalmente en dos partes fundamentales: la reserva del espacio necesario para albergar dicho código (área de depurado), y el copiado de las instrucciones del fichero ejecutable al espacio reservado. Ambas tareas están implementadas en el lenguaje de programación C. El *Listado 5* muestra el código del depurador correspondiente a la copia del código del alumno.

La reserva de espacio actualmente se efectúa por medio de la declaración en C de un array de caracteres lo suficientemente grande como para albergar el código del alumno. La dirección de comienzo de este área

es dependiente del resto de código del depurador. Como la sección que contiene el código se sitúa antes en memoria que la sección que alberga el espacio reservado, la modificación del mismo y la consiguiente variación del número de instrucciones provoca que el comienzo del espacio reservado se sitúe en posiciones diferentes.

Para solucionar este problema se parte de la premisa de que el comienzo del área de depurado está situado antes de la dirección *0x18000*. Bajo esta condición, el código se copiará siempre a partir de la dirección *0x18000*, dejando un espacio libre en direcciones inferiores a modo de margen para posibles variaciones provocadas por la modificación del código. En caso de superar dicho margen, el área de depurado se vería desplazado a la dirección *0x28000*. Esto supone que la estructura original del depurador no sólo dificulta la tarea de depuración de los alumnos, sino que es fuertemente dependiente del volumen de código del depurador de cara a añadir funcionalidades futuras. En la versión actual (v1.8.11) la variable comienza en la dirección *0x1304C*, por lo que existen 4kB de área libre que pueden emplearse para añadir código a la aplicación, pero sigue siendo una limitación conceptual del desarrollo.

Antes de realizar la copia es necesario efectuar una serie de comprobaciones sobre el código que se pretende depurar. Las diferentes lecturas realizadas sobre el código del alumno, incluyendo la lectura del fichero al completo, se realizan por medio de las funciones propias de C *fseek* y *fread* a una dirección indicada. La primera de ellas actualiza el puntero de lectura en el archivo, mientras que la segunda copia un total de bytes indicados desde el puntero de lectura, quedando a su finalización el puntero apuntando al primer byte no copiado.

En primer lugar, el depurador debe comprobar que el fichero indicado es un ejecutable ELF. Para realizar dicha comprobación se leen los 4 primeros bytes del fichero y se comprueba que coincidan con el valor del "número mágico"descrito en la *Sección 2.2.1*: un byte *0x7F* y los caracteres "ELF". Esta acción se corresponde con las líneas 6 a 8 del *Listado 5*.

Se debe comprobar además que el tamaño del fichero no supere el espacio disponible para almacenarlo. La extensión del archivo se obtiene situando el puntero de lectura al final del fichero (línea 21, se le indica a la llamada *fseek* que lo sitúe 0 bytes contando desde el final) y determinando la posición de dicho puntero mediante la llamada *ftell*, como se ve en la línea 22.

Por último, es necesario asegurarse de que el punto de entrada del código esté alineado en memoria, ya que las direcciones usadas por un programa deben ser múltiplo de 4 (deben estar alineadas al tamaño de una palabra, es decir, 4 bytes). Nuevamente, se accede a la cabecera ELF, esta vez al campo *e_entry* (desplazamiento *0x18*, ubicado en línea 29, desde el comienzo del fichero ELF, como se expuso en la *Sección 2.2.1*) donde se almacena la dirección del punto de entrada. Una vez obtenida, se comprueba en la línea 31 que dicha dirección sea múltiplo de 4.

Si se cumplen todos los requisitos descritos, se procede al copiado del contenido del objeto ejecutable al espacio reservado, situando para ello el puntero de lectura en la primera dirección del fichero (línea 37) e indicando en la llamada a *fread* la dirección de memoria donde se quiere colocar el código del alumno. Como puede verse, el binario con el código del alumno se copia de forma completa al espacio reservado, sin analizar su contenido.

```
...

/* Check if file is an ELF */
int elfSize = 0, prHeadSize = 0;
char auxBuf[4];
fread(auxBuf, 1, 4, file);
if (*(int *) &auxBuf != ELF_FILE_BEG) {
        snprintf(aux_msg, ConsLinSize + 1, NOT_ELF_MSG);
} else {
        /*Check if link addresses are correct */
        fseek(file, OffElfHeadSize, SEEK_SET);
        fread(&elfSize, 1, 2, file);
        fread(&prHeadSize, 1, 2, file);
        fseek(file, elfSize + OffVirtAddr, SEEK_SET);
        fread(auxBuf, 1, 4, file);
        if (*(int *) &auxBuf != AddrElfFile) {
                snprintf(aux_msg, ConsLinSize + 1, "%s 0x%08X", MIS_LINKED_MSG,
                AddrElfFile);
        } else {
                /* Check if ELF file fits in debugging area */
                fseek(file, 0, SEEK_END);
                flength = ftell(file);
                if (flength > ((int) &DEBUG_AREA[DebugAreaSize] - AddrElfFile -
                StackSize)) {
                        snprintf(aux_msg, ConsLinSize + 1, "%s (max %d bytes)",
                        FILE_TOO_BIG_MSG, DebugAreaSize);
                } else {
                        /* Check if entry point is aligned to word */
                        fseek(file, OffEntryPoint, SEEK_SET);
                        fread(&AddrCodeStart, 1, 4, file);
                        if (AddrCodeStart & 3 != 0) {
                                snprintf(aux_msg, ConsLinSize + 1, UNALIGN_ENTRY_MSG,
                                AddrCodeStart);
                                AddrCodeStart = AddrCodeCur;
                        } else {
                                /* Load ELF file and update data and code windows */
                                fseek(file, 0, SEEK_SET);
                                fread((char *) (AddrElfFile), 1, flength, file);

...
```

Listado 5: Comprobaciones y carga del codigo del alumno en el depurador.

3.4. Análisis del código de prueba

Siguiendo la estructura del depurador, nuestro programa de prueba combina el uso de dos lenguajes de programación diferentes, estando definido el programa principal (comienzo de la ejecución) en lenguaje C. Esta parte representa el código del depurador, y será donde se implemente una funcionalidad sencilla que sirva para comprobar que los cambios ocasionados no interfieran en la ejecución del código. Se hace uso de código complementario escrito en lenguaje ensamblador ARM, desde donde se gestionará el uso de la memoria realizando una reserva de espacio similar al necesario para almacenar el código del alumno.

La funcionalidad del código escrito en C consistirá en la suma de dos variables, almacenando el resultado en una tercera. El programa en ensamblador, por su parte, reservará espacio declarando una variable, la cual intentaremos situar al comienzo del espacio de direcciones del fichero ejecutable. El objetivo de este código es comprobar si se puede reservar espacio al principio del fichero para así alojar el ejecutable del alumno.

Respecto al modo en el que se organiza el flujo de ejecución de ambos códigos, nos encontramos con dos alternativas. La primera posibilidad es comenzar la ejecución en el código escrito en C, y llamar desde ahí a una función en ensamblador que haga la reserva de memoria, tal y como sucede actualmente en el depurador. Alternativamente, se puede comenzar la ejecución directamente en el código escrito en ensamblador, y tras manejar el espacio de memoria al comienzo del fichero, saltar al código en C. Esta estrategia alberga interés si permite simplificar los cambios necesarios para implementar la carga del código del alumno al comienzo del fichero ejecutable del depurador. Como parte del trabajo, se han explorado ambas variantes, para determinar cuál se ajusta mejor a los requisitos del proyecto.

Como se explicó en la *Sección 2.1*, el punto de entrada de un ejecutable indica la primera instrucción del programa que debe ser ejecutada. Es importante comprender que el punto de entrada no corresponde necesariamente con la primera instrucción declarada en el ejecutable, y es posible encontrar código declarado en direcciones inferiores al punto de entrada. En el caso de códigos escritos en C, el punto de entrada se define dentro de las librerías básicas del lenguaje. De este modo, se asegura la correcta inicialización de todo el entorno, así como configuraciones y comprobaciones dependientes de cada implementación de la librería. Concretamente, en el caso de códigos compilados en GCC para RISC OS el punto de entrada se localiza en un fichero llamado crt0 [18], donde el nombre proviene de una abreviatura de "C Runtime", y el '0' representa el comienzo de la ejecución. Tras realizar las comprobaciones iniciales, se efectúa un salto a la primera instrucción del código desarrollado en C, que se identifica mediante la etiqueta "main".

La primera prueba a realizar como parte del análisis del código de prueba es situar este punto de entrada en el código escrito en C o en ensamblador. En la primera variante del programa de prueba, el código principal está escrito en C, a semejanza del depurador. Es posible realizar directamente una llamada a una función en código ensamblador desde un programa escrito en C, no obstante, la forma que más información acerca del proceso nos ofrece ha sido compilar el programa en C para obtener el código ensamblador. Una vez están ambos programas en lenguaje ensamblador, las llamadas a las diferentes funciones se realizan por medio de instrucciones de salto *"BL"*. En el caso de ser necesario pasar argumentos a la función, esto se realiza por medio de los registros *R0-R3*.

Al analizar el código ensamblador ARM obtenido del programa escrito en C se observa cómo el programa principal *"main"* que hemos definido en C no es el punto de entrada, sino que lo es una función definida en una librería propia del sistema denominada *crt0.o*, tal y como se explicó en la *Sección 2.1*. Esto queda reflejado en el *Listado 6* perteneciente al mapa de símbolos del ejecutable, en el que se ven las direcciones de memoria de los símbolos *"main"*, que es el programa principal en C, y *_start*, que es la etiqueta del punto de entrada por defecto. En el listado se puede ver que la etiqueta *_start* toma la dirección *0x844C* y está definida en el fichero *crt0.o*, mientras que el símbolo *"main"* está situado en la dirección *0x8694*. Este análisis se ha verificado comprobando el valor del campo *e_entry* de la cabecera ELF, que tiene el valor *0x844C*, correspondiendo a la librería *crt0.o*.

En este listado se muestra un extracto del mapa de símbolos del ejecutable. Para cada símbolo, se indica en una primera línea la sección en la que se sitúa, seguido de la dirección exacta donde se ubica la referencia. Se muestran además el tamaño ocupado por la referencia (en el listado esto corresponde con los valores *0x1C* y *0x54*) y la ruta completa al fichero objeto que contiene la función referenciada. En una segunda línea se muestra nuevamente la dirección exacta del símbolo, junto al nombre del mismo.

```
.text         0x0000844c      0x1c    /SDFS::RISCOSpi.$/Apps/Development/!GCC/bin/../lib/
                                      gcc/arm-unknown-riscos/4.7.4/../../../crt0.o

              0x0000844c     _start

.text         0x00008694      0x54    /tmp/ccDTHwQK.o

              0x00008694      main
```

Listado 6: Direcciones de memoria de los simbolos *main* y *_start*.

En segundo lugar, se observa que los códigos en C con llamadas a otras funciones fuerzan la realización de una comprobación de memoria. La comprobación asegura que el tamaño disponible en la pila o *stack* garantice la preservación del estado del programa llamador, y se realiza independientemente de la complejidad del código en C. Este es un aspecto crítico, porque el código del depurador siempre va a involucrar llamadas a funciones, lo que se conoce como código multinivel.

Sabiendo que desde la librería se hará una llamada a la etiqueta *"main"*, hemos llamado a nuestra función en ensamblador con esta etiqueta *"main"*, y hemos renombrado al programa principal. La idea bajo estos cambios es que la ejecución del programa comienza normalmente en las librerías del lenguaje, pero se interpone una rutina en ensamblador antes de la llamada al código principal desarrollado en C. La rutina intermedia realiza la reserva de memoria y después salta al programa principal del código de prueba. De este modo habríamos insertado nuestro código ensamblador sin alterar el comportamiento del programa original.

La otra alternativa consiste en comenzar la ejecución desde la sección en ensamblador mediante la creación de un nuevo punto de entrada. De este modo, una vez reservado el espacio necesario en nuestro código en ensamblador, se prosigue con la ejecución natural del programa en C, mediante un salto a la etiqueta *"_start"* definida en la librería del sistema. Este comportamiento puede lograrse porque el comando del linker (ya sea el comando *"ld"*, o la modalidad de *"gcc"* que llama de forma interna al linker) permite especificar como entry point una sección distinta de *"_start"*. Dicho programa realiza toda la función de arranque de librerías y pasa después al código de la aplicación escrita en lenguaje C.

En ambos casos hay una particularidad importante y es que, si no se indica nada, el espacio reservado desde el código ensamblador se agrupa en una sección junto a datos similares. Por ejemplo, en caso de ser un espacio no inicializado se incluiría en la sección *".bss"*, mientras que si inicializamos los datos este espacio se incluirá en la sección *".data"*.

4. Pruebas desarrolladas

En esta sección se describen las pruebas desarrolladas con el objetivo de determinar la viabilidad de ubicar una sección en blanco en las primeras direcciones del depurador.

4.1. Compilación del código

La compilación es el proceso mediante el cual se transforma el código fuente de un programa en un fichero objeto ejecutable. Este proceso engloba las etapas de ensamblado y enlazado. Durante la etapa de ensamblado se transforman los ficheros con el código fuente en ficheros objeto. En la etapa de enlazado se combinan los diferentes ficheros objetos que conforman el programa, añadiendo además las librerías necesarias para generar el fichero ejecutable.

Tanto para la realización de las prácticas de los alumnos como para compilar el depurador se emplea la suite de compilación GCC (GNU Compiler Collection). Esta suite ofrece al usuario un conjunto de herramientas con las que trabajar sobre los programas desarrollados, permitiendo, por ejemplo, ensamblar un fichero con código o enlazar ficheros objeto para formar un ejecutable. Además, esta suite permite al usuario automatizar el uso de las herramientas disponibles a través del comando *"gcc"*, siendo posible a través de este único comando generar binarios ejecutables proporcionando códigos desarrollados en lenguajes de alto nivel. Este comando puede recibir una serie de argumentos o *"flags"* con los que controlar su funcionamiento. En el caso particular estudiado en este proyecto se tiene especial interés en la etapa de enlazado, ya que durante esta etapa se organizan las secciones en el fichero ejecutable.

El objetivo de este bloque de pruebas es determinar de qué modo trabaja el comando *"gcc"*, y en qué se diferencia un ejecutable generado a través de este comando de uno ensamblado y enlazado de forma independiente mediante los comandos *"as"* y *"ld"*. Las comparaciones se realizan sobre un programa de pruebas escrito en C con la funcionalidad descrita en la *Sección 3.4*, en el que únicamente se suma el contenido de dos variables y se guarda el resultado en una tercera.

Para efectuar el ensamblado y enlazado de forma manual es necesario compilar el código de alto nivel para obtener el código ensamblador correspondiente al programa desarrollado. De acuerdo con la documentación de *"gcc"*, podemos obtener este código ensamblador incluyendo la opción *"-S"* durante el uso de la herramienta.

El ensamblado del código permite obtener el fichero objeto correspondiente a un programa en lenguaje ensamblador. En RISC OS esto se realiza mediante el uso de la herramienta *"as"*, proporcionando en primer lugar el código que se quiere ensamblar, y en segundo lugar y de forma opcional el nombre del fichero objeto resultante por medio del flag *"-o"*.

De forma similar, el enlazado del código en RISC OS se efectúa por medio de la herramienta *"ld"*, proporcionando todos los ficheros objetos, y de forma opcional y a través del flag *"-o"* el nombre del archivo ejecutable resultante.

En nuestro caso de prueba, la obtención del código ensamblador a través de *"gcc"* y el ensamblado de este por medio de la herramienta *"as"* concluyen sin ningún imprevisto. Sin embargo, la fase de enlazado genera un error indicando que no se puede resolver la referencia a "_rt_stkovf_split_small". Esta referencia se usa en una instrucción de salto situada en el código ensamblador, y corresponde con una función definida en las librerías básicas de C. El propósito de la función es asegurarse de que el tamaño del "stack"permita albergar el estado del proceso durante un cambio de contexto provocado, por ejemplo, por una llamada a una función [22].

La conclusión que se obtiene de estos resultados es que, mientras que el comando *"gcc"* incluye por defecto las librerías de C durante la etapa de enlazado, si queremos generar el fichero ejecutable paso a paso es necesario indicar todas las librerías involucradas. Es necesario entonces conocer los ficheros objetos

correspondientes a las librerías de C, para poder incluirlos durante la etapa de enlazado.

De nuevo, la escasa documentación encontrada para el sistema RISC OS dificulta el encontrar información relativa a la ubicación de las librerías. Sin embargo, es posible obtener la localización de los ficheros objeto necesarios de forma experimental. La estrategia se basa en que el comando *"gcc"* es capaz de resolver la ubicación de dichas librerías en el paso de enlazado, por lo que intentamos pedirle que nos de un desglose de los pasos que sigue.

Como ya se ha explicado, *"gcc"* usa de forma interna la herramienta de enlazado, proporcionándole la ubicación de las librerías de C. Es posible indicarle flags a *"ld"* a través de *"gcc"* por medio de la opción *"-Wl"* siempre que esté precedido de una coma. De este modo, es posible compilar un programa con *"gcc"* de forma normal, pero pasándole el comando "–trace" a la herramienta *"ld"* durante la fase de enlazado. Este flag indica al *"ld"* que muestre por consola los nombres de los ficheros objetos a medida que los vaya procesando. El resultado obtenido se muestra en el *Listado 7*, en el que se ve la salida del comando cuando se pide que genere un fichero objeto a partir de un código en C (fases de compilación y ensamblado).

```
*gcc -Wl,--trace holamundo.c -o pruebalib
/SDFS::RISCOSpi.$/Apps/Development/!GCC/bin/ld: mode armeld_riscos
/SDFS::RISCOSpi.$/Apps/Development/!GCC/bin/../lib/gcc/arm-unknown-riscos/4.7.4/crti.o
/SDFS::RISCOSpi.$/Apps/Development/!GCC/bin/../lib/gcc/arm-unknown-riscos/4.7.4/../../
../crt0.o
SDFS::RISCOSpi.$/Apps/Development/!GCC/bin/../lib/gcc/arm-unknown-riscos/4.7.4/crtbegin.o
/tmp/ccRVBCiN.o
-lgcc_s (/GCCSOLib:/abi-2.0/libgcc_s.so)
-lunuxlib (/GCCSOLib:/abi-2.0/libunixlib.so)
/SDFS::RISCOSpi.$/Apps/Development/!GCC/bin/../lib/gcc/arm-unknown-riscos/4.7.4/crtend.o
/SDFS::RISCOSpi.$/Apps/Development/!GCC/bin/../lib/gcc/arm-unknown-riscos/4.7.4/crtn.o
*
```

Listado 7: Salida por consola al usar el flag *"-trace"* con el comando gcc, para generar un objeto a partir de un código en C.

Como se muestra en la figura, *"ld"* procesa varias librerías localizadas en el sistema. Cabe destacar los ficheros *"crt0.o"*, *"crtend.o"* y *"crtbegin.o"* mencionados en la *Sección 3.4*. El fichero *"crt0.o"* contiene la funcionalidad con el punto de entrada del programa. *"crtbegin.o"* alberga las funciones de librería ejecutadas en la inicialización antes de pasar al código del programa desarrollado. *"crtend.o"* contiene las funciones que se ejecutan tras finalizar el programa desarrollado, antes de devolver el control de la ejecución al sistema operativo.

Al repetir la prueba de enlazado paso a paso, indicando esta vez los ficheros objeto de las librerías de C mostrados en el Listado 7 se vuelve a obtener un error de enlazado. Esta vez el error reside en que no se están reconociendo las librerias *"libgcc_s.so"* y *"libunixlib.so"*. A diferencia del resto de librerías empleadas, estos archivos son objetos dinámicos, cuya ejecución se comparte entre diferentes programas que hagan uso de la librería.

Un enlazado estático involucra archivos objeto y librerías que se usarán de forma exclusiva por el ejecutable. De este modo, todo el contenido necesario para la ejecución completa del código está contenido en el archivo final. Sin embargo, durante un enlazado dinámico se utilizan objetos y librerías compartidos por diferentes procesos. Esta estrategia permite reducir los recursos de memoria necesarios durante la ejecución de los programas, puesto que los códigos comunes como las librerías del sistema se cargan una única vez y se comparten entre todos los procesos que los requieran. Por este motivo, después de la etapa

de enlazado estos recursos deben seguir estando disponibles en el sistema. En este caso, se resuelven referencias tanto en la fase de inicialización como durante la propia ejecución.

Por lo tanto, estos ficheros requieren un tratamiento distinto del resto de objetos de librerías del sistema. Se ha intentado sortear este inconveniente tratando de forzar un enlazado estático, en el que se copiaría el contenido de todas las librerías en el binario ejecutable generado. Sin embargo, no es posible convertir una librería dinámica en una estática, ya que estas no contienen toda la información necesaria para ser enlazadas de este modo. Además, las librerías dinámicas tienen un formato especial en el que el punto de entrada y ciertas referencias problemáticas están resueltas con anterioridad. Por lo tanto, es necesario hacer uso de la versión estática de la librería, cuya localización desconocemos, y que puede no estar siquiera presente en el sistema.

Esta situación ha provocado que se prescinda de la estrategia de ensamblar y enlazar el ejecutable manualmente. En este punto no se tenía la certeza de que la versión estática de las librerías estuviesen presentes en el sistema, y en caso de estarlo no se disponía de una estrategia clara para encontrarlas. Ante esto se ha decidido que no compensaba invertir recursos en realizar el ensamblado y enlazado manualmente, únicamente para obtener un mayor control de la fase de enlazado.

En su defecto, se ha decidido continuar usando el comando *"gcc"*, el cual, como se ha comentado, hace uso de forma interna de la herramienta *"ld"*. La forma de controlar la etapa de enlazado se basará a partir de ahora en el paso de argumentos al comando *"ld"* a través del propio comando *"gcc"* por medio del flag *"-Wl"*.

4.2. Reubicación del código

El objetivo de las pruebas realizadas es determinar si es posible reubicar un espacio vacío al comienzo del código del depurador donde alojar el código del alumno, y de serlo, de qué modo se puede realizar. Inicialmente las pruebas se realizan sobre los códigos de prueba, y una vez se comprueba que los resultados obtenidos son los esperados se trasladan al depurador.

En primer lugar hay que declarar el espacio vacío dentro del código de nuestro programa. Esta parte se ha desarrollado en lenguaje ensamblador ARM, debido a que este lenguaje ofrece mayor libertad y nivel de granularidad en la definición del espacio. Además, este espacio se ha declarado en una sección propia, llamada *"studentcode"*. El modo en el que se ha construido esta sección se muestra en el *Listado 8*. La sección se declara por medio de la directiva *".section"* seguida del nombre que se le quiera dar. Además, se ha indicado que esta sección puede ser asignada en memoria y tiene permisos de ejecución y escritura por medio de las etiquetas *"axw"*.

```
1 .section studentcode, "axw"
2 .globl arranque
3 arranque:
4 b _start
5 .space 0x20000
```

Listado 8: Declaración de un espacio en blanco dentro de la sección *studentcode*.

La estrategia seguida, como se ha explicado en el análisis del código de prueba, consiste en desplazar el punto de entrada a esta sección definida. Esto se lleva a cabo gracias a la etiqueta *"arranque"* definida de manera global, como se ve declarado en las líneas 2 y 3 de el Listado 3. Nada más entrar en la sección declarada se salta al punto de entrada original del programa desarrollado en C, etiqueta *"_start"*, como se ve en la instrucción de la línea 4 de la figura. Por último, en la última línea se ve cómo se declara el

espacio en blanco donde irá situado el código del alumno.

Al ver este fragmento de código uno se podría preguntar qué sentido tiene arrancar la ejecución del código en esta sección, si no se va a ejecutar otra cosa más que el salto al programa original. Esta acción de colocar la sección al comienzo de la ejecución se realiza para solucionar el siguiente problema: si no se ejecuta ninguna instrucción, *"gcc"* no asigna direcciones de memoria a la sección. Es decir, aunque la sección se declare de forma correcta, esta no se carga en memoria con el ejecutable, por lo que es imposible usarla para alojar el código del alumno. De esta forma, ejecutando esta instrucción de salto al inicio del código, nos aseguramos que el área reservada aparezca en memoria.

En primer lugar, es necesario conocer la distribución en memoria del ejecutable en estas condiciones. La *Listado 9* muestra el contenido de la cabecera de una de las secciones del fichero ejecutable obtenido. Dicha cabecera de sección sigue el formato descrito en la *Sección 2.2.3*, y describe la sección correspondiente al bloque *"studentcode"*. Como se puede ver en la figura, la sección comienza en la dirección *0x8780* (campo *"SH_ADDR"*). Comprobamos en esta imagen, además, que el espacio se ha creado de forma correcta, ya que el tamaño ocupado por la sección es de *0x20004* bytes (los 4 bytes extra corresponden con la instrucción de salto, campo *"SH_SIZE"*).

```
Section header table: 9
------------------------

SH_NAME         0x00020C38      0x0000004E      studentcode
SH_TYPE         0x00020C3C      0x00000001      SHT_PROGBITS (Defined by program)
SH_FLAGS        0x00020C40      0x00000006      Alloc|Exec
SH_ADDR         0x00020C44      0x00008780
SH_OFFSET       0x00020C48      0x00000780
SH_SIZE         0x00020C4C      0x00020004
SH_LINK         0x00020C50      0x00000000
SH_INFO         0x00020C54      0x00000000
SH_ADDRALIGN    0x00020C58      0x00000004
SH_ENTSIZE      0x00020C5C      0x00000000
```

Listado 9: Valores iniciales de la cabecera de la sección *studentcode*.

En la figura, la primera columna se corresponde con el nombre que tiene el campo de la cabecera, y la segunda columna indica la dirección de memoria en la que se localiza dicho campo. Por otra parte, se muestra en la tercera columna el valor que almacena cada campo. Por último, se puede indicar de forma adicional en la cuarta columna el valor o las características interpretadas del campo, como por ejemplo el valor de la tabla de strings en la posición *0x4E* en el caso del nombre.

El objetivo será entonces reubicar esta sección, explorando para ello las diferentes opciones que nos brindan los comandos *"ld"* y *"gcc"*. En primer lugar, se intenta desplazar la sección por medio de la opción *"–section -start"* de *"ld"*. Al usar esta opción es necesario proporcionar el nombre de la sección que se quiere mover y la dirección de comienzo de la misma.

Siguiendo esta estrategia se ha tratado de reubicar la sección *"studentcode"*. Como se vio en la *Sección 2.2*, en la estructura de un ejecutable ELF en memoria se sitúa primero la cabecera ELF, seguida de las cabeceras de segmentos y de las secciones que conforman el ejecutable. La información extraída de estas cabeceras nos permite crear un mapa acerca de la estructura del fichero. En primer lugar la cabecera ELF termina en la dirección *0x8048*, a continuación se ubican las cabeceras de segmentos, las cuales finalizan en la dirección *0x80E8*. Posteriormente se encuentran las secciones, comenzando con la sección *".interp"* situada entre las direcciones *0x80E8* y *0x80f8*. Esta sección es especialmente relevante, ya que además

conforma un segmento ella sola. Todos los desplazamientos se realizan dentro del segmento que contiene a las partes afectadas.

En la primera prueba realizada se ha intentado colocar la sección *"studentcode"* en la dirección *0x80E8*. Con esta configuración, se produce el fallo de enlazado descrito en el *Listado 10*.

```
*gcc -Wl,-earranque,--section-start,studentcode=80e8,-Map=rellocE8.map -o rellocE8
arranque2.o holamundo.o
*
/SDFS::RISCOSpi.$/Apps/Development/!GCC/bin/ld: section .interp loaded at
[00008108,00008116] overlaps section studentcode loaded at [000080e8,0000280eb]
collect2: error: ld returned 1 exit status
```

Listado 10: Error producido al superponer secciones.

Como se ve, el fallo se produce ya que la sección *"studentcode"* se está solapando con la sección *".interp"*, la cual contiene la ruta al intérprete del sistema operativo. Por tanto, especificar de este modo la dirección de comienzo de una sección no parece asegurar que el resto de secciones vayan a ser desplazadas, dejando espacio para la sección insertada. Para solucionar este problema se ha planteado especificar la dirección de comienzo de todas las secciones, encadenando de forma manual una detrás de otra, evitando de este modo que las secciones se pisen entre sí.

A modo de prueba se ha indicado únicamente las direcciones de comienzo de la sección *".interp"* y de la propia sección *"studentcode"*. Al enlazar el ejecutable de este modo no se produce ningún tipo de error durante la compilación, y al analizar las cabeceras de las secciones se ve que todas las secciones están puestas de forma consecutiva. Una pequeña prueba, desplazando únicamente la sección *".interp"*, muestra como al desplazar una sección todas las secciones siguientes sufren el mismo desplazamiento, moviéndose "en bloque".

Sin embargo, esta solución descrita no resulta válida, ya que el ejecutable resultante provoca un error. Concretamente se indica que el fichero *"SharedLibs:lib."* no se reconoce. Analizando estos resultados se ha llegado a la siguiente conclusión: la estrategia de forzar reubicación de secciones en el fichero ejecutable, además de ser laboriosa y suponer problemas de escalabilidad si se implementan nuevas funcionalidades, genera un error de ejecución desconocido, atribuido al uso de librerías compartidas. Es por tanto necesario explorar otras alternativas a la ubicación del segmento al comienzo del ejecutable.

Pese a no poder indicar de forma explícita la posición de memoria de las diferentes secciones, existe una alternativa para controlar de qué modo se mapean estas secciones de los ficheros de entrada al fichero de salida. La herramienta *"ld"* puede hacer uso de un fichero denominado *linker script* [8], en el que se detallan todos los parámetros necesarios para llevar a cabo el enlazado del código. Entre estos aspectos se encuentran el tratado de ficheros de entrada, el formato de los ficheros involucrados, la organización del fichero resultante y las direcciones de memoria de las diferentes secciones. Este script se puede obtener al forzar que *"ld"* imprima por consola las sentencias que usa para enlazar el fichero. Una vez obtenido, es posible modificarlo e indicar que se debe usar este fichero en un nuevo enlazado con *"ld"*. Concretamente, para obtener el script solo es necesario indicar la opción *"–verbose"* del *"ld"* durante el proceso de compilación como se muestra en la siguiente entrada:

```
gcc -Wl,--verbose,-earranque -o obtenerLinker arranque2.o holamundo.o
```

De este modo se imprime por pantalla todo el contenido del linker script usado por defecto durante el enlazado.

El lenguaje en el que se escribe el script está basado en sentencias, que sirven tanto para configurar aspectos concretos del comportamiento de la herramienta a través de palabras clave, como para configurar de qué modo se relacionan los ficheros de entrada y de salida. Además, el lenguaje define su propia sintaxis para tratar los diferentes tipos de dato numérico, identificando la base sobre la que se están expresando. Sin embargo, existen dos comandos en particular con un impacto fundamental en el fichero de salida. Estos son el comando *"SECTIONS"* y el comando *"MEMORY"*.

El comando *"SECTIONS"* se encarga de describir la imagen estructural del fichero de salida, pudiendo especificar diferentes niveles de detalle. Este comando es el único comando imprescindible para el enlazado del código, por lo que ha de estar presente en el fichero de *linker script*. Por contra, el resto de sentencias del fichero son completamente opcionales, incluyendo el comando *"MEMORY"*. Este comando describe la memoria disponible en la arquitectura sobre la que se ejecuta el linker, por lo que resulta importante para el proceso, pese a ser opcional. En caso de no estar presente, la herramienta supondrá que el bloque de memoria tiene suficiente espacio contiguo como para almacenar todo el documento.

Para el caso particular del proyecto, el comando *"SECTIONS"* es de especial interés. Dentro de este comando se describen aspectos como el punto de entrada al fichero, se asigna un valor a diferentes símbolos del programa, y se describe la colocación de las secciones en el fichero de salida. Para cada sección que se quiera especificar en el fichero de salida es necesario indicar qué secciones de qué ficheros de entrada se deben agrupar. El orden en el que se declaren las secciones en el comando indicará el orden en que irán colocadas en el fichero de salida. Además, se puede indicar en este punto la dirección de memoria en la que se quiere situar la sección.

En el ejemplo mostrado en el *Listado 11* se están definiendo en el fichero de salida las secciones *".text"*, *".DATA"*, *".data"* y *".bss"*, colocándose en el orden en el que se han citado. Entre las llaves están definidos para cada sección del fichero de salida qué secciones de los ficheros de entrada se deben agrupar. Las secciones de entrada se detallan entre paréntesis, y los ficheros de los que obtenerlas se indican inmediatamente delante. En el ejemplo, la sección *".text"* del fichero de salida contendrá el conjunto de las secciones *".text"* que se encuentren en cualquiera de los ficheros de entrada (el símbolo * indica que se deben buscar en todos los ficheros). En la sección *".DATA"* se agruparán todas las secciones *".data"* presentes en los ficheros de entrada que comiencen con una letra mayúscula. El resto de secciones *".data* de los ficheros de entrada que no se hayan incluido en el caso anterior se agruparán en la sección *".data"* del fichero de salida.

```
1 SECTIONS {
2   .text : { *(.text) }
3   .DATA : { [A-Z]*(.data) }
4   .data : { *(.data) }
5   .bss : { *(.bss) }
6 }
```

Listado 11: Ejemplo 1 de definición de secciones.

Si se quiere estructurar el fichero de salida de forma más precisa se pueden definir las secciones como se muestra en el *Listado 12*. En el fichero de salida se definen las secciones personalizadas *"outputa"*, *"outputb"* y *"outputc"*, en ese mismo orden. La sección *"outputa"* se ubicará en la dirección *0x10000*, y agrupará las secciones *".input1"* de los ficheros de entrada *"all.o"* y *"foo.o"*. La sección *"outputb"* agrupará la sección *".input2"* del fichero de entrada *"foo.o"* y la sección *".input1"* del fichero . *"foo1.o"*. En la sección *"outputc"* se agruparán el resto de secciones *".input1"* y *".input2"* del resto de ficheros que no se hayan tratado en los casos anteriores.

```
SECTIONS {
  outputa 0x10000 :
    {
    all.o
    foo.o (.input1)
    }
  outputb :
    {
    foo.o (.input2)
    foo1.o (.input1)
    }
  outputc :
    {
    *(.input1)
    *(.input2)
    }
}
```

Listado 12: Ejemplo 2 de definición de secciones.

Esta configuración es solamente una pequeña parte de las especificaciones que trae el *linker script* por defecto a la hora de usar la herramienta *"ld"*. Para asegurarnos de que la funcionalidad de la aplicación queda intacta, es necesario indicar las modificaciones pertinentes sobre el mismo script que se ha estado usando por defecto en el resto de enlazados, obteniéndolo tal y como se ha mostrado anteriormente.

Ahora solo hace falta copiar el contenido a un fichero para poder pasarlo como argumento al *"ld"* más adelante. El efecto buscado en el fichero de salida es situar la sección *"studentcode"* declarada en los ficheros de entrada en la primera posición. Para ello, tal y como se ha explicado, es necesario modificar el comando *"SECTIONS"* encontrado en el script. Concretamente, hace falta añadir al comienzo de este la sentencia *"studentcode : *(studentcode)"*. De este modo se indica que en el fichero de salida deben agruparse bajo la sección *"studentcode"* las secciones con el mismo nombre localizadas a lo largo de los ficheros de entrada, sin necesidad de especificar en qué fichero se declara dicha sección.

Estos cambios quedan reflejados en el *Listado 13*. En la imagen solo se muestra una pequeña fracción del script, ya que el fichero indica configuraciones muy específicas sobre el código y que no afectan al problema planteado.

```
...
SECTIONS
{
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x00008000));
  . = SEGMENT_START("text-segment", 0x00008000) + SIZEOF_HEADERS;
  PROVIDE (Image$$RO$$Base = .);

  studentcode     : { *(studentcode) }
  .interp         : { *(.interp) }
  .note.gnu.build-id : { *(.note.gnu.build-id) }
  .hash           : { *(.hash) }
  .gnu.hash       : { *(.gnu.hash) }
  .dynsym         : { *(.dynsym) }
...
```

Listado 13: Fragmento del *linker script* tras la modificación.

Ahora solo queda indicar que este nuevo script debe ser utilizado durante la compilación, en lugar de emplear la configuración por defecto. Para ello es necesario indicarle a la herramienta, a través del flag *"-T"*, el nombre del fichero que contiene el script. El comando que se debe ejecutar para compilar el código con los cambios realizados es el siguiente:

```
gcc -Wl,-T,linkerScript,-earranque -o obtenerLinker arranque2.o holamundo.o
```

La compilación del código con dicho comando finaliza de forma exitosa. Al analizar la cabecera ELF con el fin de comprobar que la reubicación se haya llevado a cabo tal y como se ha indicado en el script, encontramos la estructura mostrada en la *Listado 14*. Como se ve en dicha imagen, la sección reservada para alojar el código del alumno comienza en la dirección *80E8* (campo *SH_ADDR*), y su cabecera ocupa la primera entrada de la tabla de cabeceras de sección. La prueba de ejecución del código resultante no muestra error alguno, manteniéndose de forma completa la funcionalidad inicial del código.

```
Section header table: 1
------------------------

SH_NAME           0x00020B28          0x0000001B      studentcode
SH_TYPE           0x00020B2C          0x00000001      SHT_PROGBITS (Defined by program)
SH_FLAGS          0x00020B30          0x00000006      Alloc|Exec
SH_ADDR           0x00020B34          0x000080E8
SH_OFFSET         0x00020B38          0x000000E8
SH_SIZE           0x00020B3C          0x00020000
SH_LINK           0x00020B40          0x00000000
SH_INFO           0x00020B44          0x00000000
SH_ADDRALIGN      0x00020B48          0x00000001
SH_ENTSIZE        0x00020B4C          0x00000000
```

Listado 14: Valores iniciales de la cabecera de la sección *studentcode*.

Los resultados analizados muestran que se ha logrado reubicar la sección *"studentcode"* de forma satisfactoria. No obstante, y como se ve en la *Figura 7*, la dirección de comienzo de la sección es *80E8*. Aun habiendo logrado desplazar la sección, no podemos cargar en dicha sección el código del alumno sin especificar una dirección de comienzo en la fase de enlazado, ya que los primeros 232 bytes (valor *0xE8* en hexadecimal) están ocupados por otros contenidos del fichero. Para evitar dicho paso es necesario no solo que la primera sección sea el espacio reservado, sino que este espacio comience en la dirección *0x8000*. Recordando lo explicado en la *Sección 2.1*, en la estructura del fichero ELF se encuentra en primer lugar la propia cabecera ELF, seguida de la tabla de cabeceras de segmento, y finalmente las propias secciones con el código a ejecutar. En la siguiente sección se analiza la importancia y contenido de dichas cabeceras para la ejecución del programa, y la capacidad de reemplazar su contenido de forma controlada.

4.3. Borrado de cabeceras

Como se ha visto, las modificaciones efectuadas consiguen ubicar el espacio reservado en la primera sección del programa. Sin embargo, la presencia de las cabeceras en las primeras direcciones de memoria provocan que la sección *"studentcode"* se ubique en la dirección *0x80E8*.

Esta situación hace que se plantee la posibilidad de hacer uso del espacio de memoria empleado por las cabeceras para alojar otro contenido durante la ejecución del programa. A priori, las cabeceras de segmento solo deberían utilizarse durante las etapas cargado y de enlazado, ya que contienen información usada por el cargador y el linker para estructurar el fichero. De igual modo, una vez iniciada la ejecución del código no debería ser necesario hacer uso de las cabeceras ELF.

La idea final detrás de este análisis es reaprovechar el espacio ocupado por cabeceras innecesarias una vez empezada la ejecución del código. De este modo, se encadenaría el espacio ocupado por las cabeceras con el espacio reservado en la sección *"studentcode"*, ubicada al comienzo de las secciones del código. Realizando esta modificación se situaría el comienzo del área reservada para el código del alumno en la dirección *0x8000*, permitiendo ejecutar cualquier código dentro del depurador, sin necesidad de realizar un enlazado adicional a una dirección diferente.

Como se ha mencionado se necesitaría poder reescribir dos estructuras, la cabecera ELF y la tabla de cabeceras de sección. Para facilitar el análisis de los resultados de las pruebas primero se ha procedido a borrar cada cabecera por separado, de forma individual. Esta tarea se ha realizado por medio de un bucle en ensamblador, que una vez se ejecuta, sobrescribe el contenido de las direcciones de memoria especificadas. Como la cabecera ELF ocupa hasta la dirección *0x8048*, se ha procedido en primer lugar a sobrescribir el contenido almacenado en memoria entre las direcciones *0x8048* y *0x80e8*.

El bucle utilizado para este propósito se describe en el siguiente listado. En primer lugar, se definen en los registros r0 y r1 las direcciones de memoria entre las que se va a realizar la escritura. En el registro r2 se indica qué contenido se quiere escribir en memoria. Es necesario considerar que reescribir el bloque con bits a 0 no aporta suficiente información, ya que hay campos de cabeceras que toman dicho valor. Esta prueba no provocaría el fallo del código, haciendo ver que es una solución viable. Sin embargo, una vez cargado el código del alumno la ejecución fallaría, ya que el contenido de dicho código probablemente no sean bits nulos. Se ha optado entonces por sobrescribir el contenido en memoria con el valor *0x616C6F68*, el cual se traduce por la palabra *"hola"* de acuerdo a la codificación ASCII. Esto facilita la comprobación de que la escritura sobre los bloques ha sido exitosa, por lo que una prueba de ejecución exitosa daría una mayor garantía del funcionamiento de la estrategia.

Tras declarar las direcciones sobre las que trabajar y el contenido que se va a escribir sobre ellas, se ejecuta el bucle. En primer lugar se evalúa si la dirección que va a ser sobrescrita es inferior al límite del área que se quiere borrar. De no ser así se da por finalizado el bucle y se continua con la ejecución normal del código. Si la dirección aún está dentro del intervalo especificado, se almacena el contenido de r2 en la posición de memoria almacenada en r1. Para continuar, se incrementa en 4 la dirección a sobrescribir (hay que recordar que cada dirección de memoria tiene un tamaño de 4 bytes) y se salta de nuevo al comienzo del bucle.

```
...
    ldr r0, =8048
    ldr r1, =80e4
    ldr r2, =616C6F68
INICIO:
    cmp r0, r1
    bge FIN
    str r2, [r0, #0]
    add r0, r0, #4
    b INICIO
FIN:
...
```

Listado 15: Bucle usado para sobrescribir direcciones de memoria.

La ejecución de este código se efectúa sin ninguna complicación. La siguiente figura muestra el contenido de las direcciones de memoria correspondiente a la tabla de cabeceras de segmento tras la ejecución. Como se ve, las direcciones han pasado a contener el valor especificado en el registro, confirmando que la escritura se ha realizado de la manera esperada.

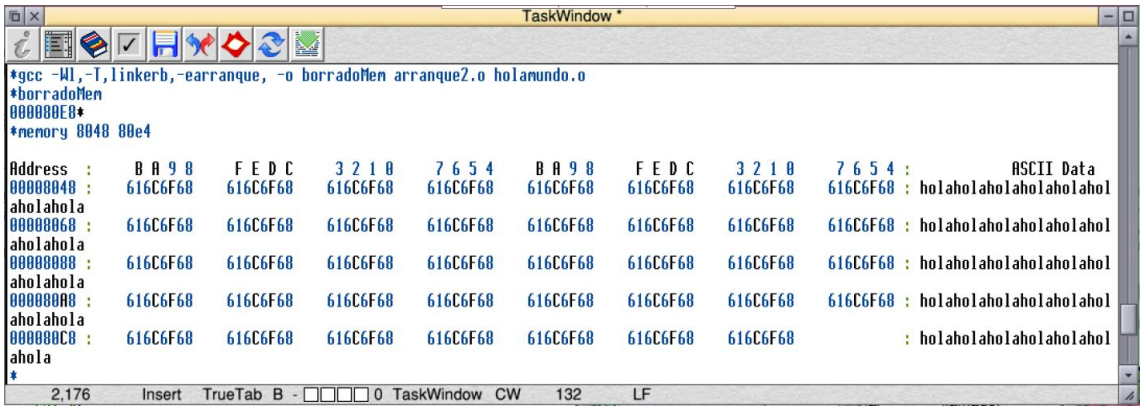

Figura 5: Contenido de la tabla de cabeceras de segmentos tras ser sobrescrita.

Tras observar que no surgen problemas de ejecución al sobreescribir la tabla de cabeceras de segmento, es necesario comprobar que se puede hacer lo mismo con la cabecera ELF. Para esta prueba se aprovecha el mismo código del *Listado 15*, pero sustituyendo las direcciones de comienzo y final indicadas en los registros r0 y r1 por las direcciones *0x8000* y *0x8048*, respectivamente.

El programa nuevamente compila sin ningún tipo de error. Sin embargo, ejecutarlo provoca que el sistema operativo se quede congelado, de forma que no se puede interactuar ni realizar ningún tipo de tarea sobre el equipo. Para solucionar el fallo es necesario reiniciar el equipo Raspberry Pi.

El siguiente paso es determinar concretamente qué parte de la cabecera ELF no puede ser sobrescrita. Es necesario destacar que todos los binarios analizados durante el trabajo constan de un tamaño de cabecera ELF de 72 bytes (valor *0x48* en notación hexadecimal). Sin embargo, el estándar ELF únicamente define campos en los primeros 52 bytes (valor *0x34* en hexadecimal) de cabecera. Existe por tanto un hueco de 20 bytes al final de la cabecera ELF cuyo contenido no está especificado.

En este punto no se dispone de información acerca del contenido en los últimos 20 bytes de la cabecera, ni por qué reescribirlo pueda causar un fallo del sistema operativo. En el análisis de la cabecera se concluyó que ninguno de los campos especificados contiene información que deba usarse durante la ejecución del código. Para comprobar dicha afirmación, se ha vuelto a repetir la prueba de borrado de la cabecera, pero únicamente de los primeros *0x34* bytes.

En esta ocasión el programa resultante ejecuta sin mostrar ningún error. Se puede concluir por tanto que en el espacio restante de la cabecera ELF hay guardada información necesaria para la ejecución del código. Para conocer más acerca de qué contenido está provocando el fallo de ejecución, se ha procedido a reescribir de forma individual el contenido almacenado en memoria entre las direcciones *0x8034* y *0x8048*. Se ha determinado de este modo que las direcciones conflictivas son las comprendidas entre *0x8034* y *0x8040*. Alguna de las direcciones de este rango pueden sobreescribirse sin que se congele el sistema operativo; sin embargo, después de sobrescribirlas, cualquier comando ejecutado en la Raspberry Pi falla.

El contenido de estas direcciones es el mostrado en el *Listado 16*. Como se puede ver, la primera y la última dirección mostradas contienen el valor *0x00000000*. Ya se comentó esta posibilidad al explicar las pruebas de sobrescritura de memoria, y por esta razón es importante que el contenido con el que se sobrescriban no tenga posibilidad de aparecer de forma normal en memoria. Es importante destacar que el contenido de las 2 direcciones intermedias no es constante, sino que varía en cada compilación del código.

```
...
00008034  :  ....  : 00000000
00008038  :  H :I  : 493AA048
0000803C  :  t :I  : 493AA074
00008040  :  ....  : 00000000
...
```

Listado 16: Contenido de las direcciones de memoria conflictivas.

Debido a la escasa documentación disponible, no ha sido posible determinar qué función tiene el contenido almacenado en las direcciones mencionadas. Sin embargo, existe una posible forma de sobrescribir las direcciones conflictivas sin bloquear el sistema operativo: hacer una copia de respaldo del contenido original de dichas direcciones. De este modo, podrían sobrescribirse las cabeceras del depurador con el código del alumno y volver a reponer su contenido original una vez finalizada la ejecución de este.

Para valorar si esta solución es viable se ha definido un pequeño caso de prueba. En primer lugar, se reserva un espacio en memoria de 16 bytes (para las 4 direcciones). Seguidamente se guarda en dicho espacio el contenido que se quiere sobrescribir. Una vez guardado el contenido que se quiere preservar, se sobrescriben las direcciones de interés con el valor *0x616C6F68*. Finalmente, se repone el contenido original guardado previamente.

Sin embargo, la ejecución del código continúa mostrando los mismos problemas de bloqueo del sistema operativo. Este resultado indica que el fallo no se produce cuando se intenta acceder desde el propio ejecutable al contenido de las direcciones y se encuentra un valor inesperado. En su lugar, el fallo se produce inmediatamente después de sobrescribir el contenido de la dirección.

A la vista de los resultados, se concluye que no es posible disponer del espacio en memoria desde el comienzo del ejecutable del depurador. Para soslayar esta limitación, se explora el uso de una copia parcial del fichero ejecutable de los programas de alumno, omitiendo parte de las cabeceras de programa.

4.4. Ejecución de un programa sin cabecera ELF

Como se ha visto, no se puede sobrescribir una parte de la cabecera ELF. No obstante, se podría hacer funcionar en el depurador un código ensamblador que comience en la dirección por defecto (*0x8000*), empleando únicamente el depurador, con el espacio conseguido hasta el momento. La idea propuesta consiste en no copiar el contenido de la cabecera ELF del código del alumno. De este modo no sería necesario sobrescribir la cabecera del depurador, evitando el fallo que esto genera. Esta estrategia es posible gracias a tres factores fundamentales.

El primero de ellos es que todas las cabeceras ELF de los programas que se han analizado tienen el mismo tamaño. De este modo, si se copia el programa del alumno ignorando la cabecera ELF, todas las instrucciones restantes, y en general cualquier referencia presente en el código, va a estar situada en la misma dirección respecto de la que estaría si se ejecutase el código directamente sobre el sistema operativo.

El segundo hecho que permitiría esta estrategia es que, como se comenta en la *Sección 3.3*, a ojos del sistema operativo el código del alumno que se ejecuta dentro del depurador se ve como código del propio depurador. Esto significa que, en la práctica, la aplicación que se está ejecutando es el depurador. Por tanto, teóricamente solo son necesarias las instrucciones y los datos del código del alumno para su depurado, puesto que el depurador ejecuta estas instrucciones como propias.

Por último, las pruebas de borrado de la cabecera ELF muestran que los campos declarados en el estándar no se usan durante la ejecución. Además, las direcciones que provocan el fallo lo hacen a nivel del sistema operativo, no al nivel de la aplicación que se ejecuta. Como el sistema operativo no distingue la ejecución del código de alumno del de la propia herramienta, debería poderse ejecutar el código del alumno sin la necesidad de copiar los contenidos de su cabecera ELF.

Para evaluar la estrategia planteada, se realiza una comprobación empírica de su validez. Dado que los problemas de bloqueo del sistema surgen durante la ejecución del programa, las pruebas realizadas consisten en ejecutar códigos escritos en ensamblador dentro del depurador, sin copiar el contenido de la cabecera ELF de los mismos. Para esta prueba se utiliza el código del depurador presente en su última versión, modificado para no copiar la cabecera ELF del programa del alumno. El resto de características de la ejecución se mantienen, incluyendo el comienzo del ejecutable de alumno en la dirección *0x18000*, pues el propósito es únicamente establecer la necesidad de disponer de la cabecera ELF del código de alumno durante su ejecución.

En la *Sección 3.3.4* se explica cómo se realiza el proceso de copia del programa del alumno dentro del espacio de memoria del depurador. El procedimiento descrito se realiza mediante el código del *Listado 5*. Dentro de dicho código, en la línea 37 se coloca el puntero de acceso al fichero apuntando a la primera dirección del mismo, usando la función *fseek* con un desplazamiento de 0 bytes.

Para realizar la copia sin incluir la cabecera ELF, se actualiza el argumento de desplazamiento para que tome el valor *0x48* (tamaño de la cabecera ELF). Asimismo, se avanza el puntero con la dirección del espacio de memoria del depurador donde se va a realizar la copia. Con esto se asegura que las direcciones del código se mantengan, omitiendo únicamente la copia del contenido de la cabecera ELF.

Realizando estas modificaciones el nuevo código queda del siguiente modo:

```
35 ...
36     /* Load ELF file and update data and code windows */
37     fseek(file, 0x48, SEEK_SET);
38     fread((char *) (AddrElfFile + 0x48), 1, flength, file);
39 ...
```

Listado 17: Carga del código del alumno sin cabecera ELF.

Para verificar la correcta ejecución en el depurador del código de alumno sin su cabecera ELF, se ha optado por utilizar los programas de verificación presentes en el repositorio de GitHub del depurador [7]. Estos programas, aparte de comprobar el funcionamiento de facetas específicas del depurador, tienen descrito el comportamiento que deben presentar. De este modo se puede saber con exactitud si los programas mantienen su ejecución normal.

El uso de estos programas es especialmente útil, ya que varios de ellos están ideados para producir fallos durante su ejecución, verificando la capacidad de la herramienta de gestionar los mensajes de aviso correspondientes. De este modo se puede comprobar que los programas sin cabecera ELF se comportan como un ejecutable normal, incluido en el caso de presentar fallos en el código.

Durante las pruebas realizadas, los programas ejecutados en el depurador muestran el comportamiento esperado. Se comprueba por tanto que no es necesaria la presencia de la cabecera ELF durante la ejecución del código del alumno en el depurador. De este modo, se puede seguir la estrategia planteada al comienzo de esta sección: copiar el contenido del programa del alumno tras la cabecera ELF del depurador, aprovechando las direcciones en las que originalmente se almacenan las cabeceras de sección del ejecutable y el bloque de memoria situado en la primera sección.

4.5. Comprobaciones en el depurador

Como se explica en la sección *Sección 3.1*, la mayoría de las pruebas detalladas en este capítulo se han realizado sobre programas simples. Tras analizar los resultados de dichas pruebas y establecer las modificaciones que deben emplearse, se comprueba que la estrategia también funciona correctamente con el código completo del depurador.

Para albergar el espacio reservado donde colocar el código del alumno, se ha declarado en un nuevo fichero una sección personalizada bajo el nombre de *"studentcode"*. En el mismo fichero se ha incluido una función de sobrescritura, igual a la mostrada en la *Sección 4.3"*.

De igual modo a como se ha realizado en los programas de prueba, se ha obtenido el *linker script* usado por defecto durante la compilación del depurador UCDebug. De igual modo se le ha modificado para colocar la sección *"studentcode"* tras las cabeceras de sección.

Las pruebas de enlazado y ejecución del programa demuestran que la estrategia cumple su cometido y no presenta errores ni bloqueos del sistema. Para comprobar que todas las funcionalidades del depurador se desarrollan como deberían, se ha hecho uso de los test de prueba mencionados en la *Sección 4.4*. Los resultados obtenidos muestran que el depurador es capaz de ejecutar y depurar código con los mismos resultados que el ejecutable original del depurador.

Por limitaciones de tiempo no se ha explorado la ejecución del código de alumno en el nuevo espacio reservado al comienzo del depurador. El rediseño de la aplicación para actualizar la ejecución del código en la nueva ubicación requiere un estudio en mayor profundidad de la herramienta, y se deja como trabajo futuro. Sin embargo, se puede concluir que la estrategia de reubicación del código de alumno en el espacio de memoria del depurador ha sido exitosa, por lo que puede rediseñarse la herramienta para admitir códigos de alumno que comiencen en la dirección por defecto (*0x8000*). Se ha cumplido por tanto el objetivo del trabajo, tanto en cuanto al estudio de la estructura de los programas como al análisis de viabilidad de la estrategia de reubicación.

5. Conclusiones

El objetivo principal de este proyecto ha sido realizar un estudio de la viabilidad de una propuesta de mejora sobre el depurador UCDebug, y en caso de resultar viable desarrollar una estrategia de mejora. La modificación propuesta consiste en reubicar los segmentos de código de los ejecutables del programa, permitiendo reaprovechar el espacio ocupado por las cabeceras de sección del depurador, y omitir la copia de la cabecera ELF del programa del alumno. De este modo, se puede cargar y ejecutar el código que se va a depurar al comienzo del espacio de direcciones.

El depurador es una herramienta muy útil que permite a los programadores comprobar el correcto funcionamiento de una aplicación, facilitando la detección y corrección de errores. Esto es particularmente cierto para alumnos que están aprendiendo a desarrollar códigos, y que inicialmente introducen un alto número de fallos en sus programas. En el laboratorio de algunas asignaturas del área de Estructura y Organización de Computadores se emplea UCDebug, un depurador para el sistema operativo RISC OS desarrollado en la Universidad de Cantabria. Sin embargo, este depurador ejecuta el código del alumno como si fuese parte de la herramienta, por lo que necesita un proceso de enlazado específico para que el código no empiece al comienzo del espacio de direcciones. Es es uno de los motivos por los cuales me ha parecido un proyecto interesante, ya que me ha permitido comprender un poco mejor qué ocurre cuando se pretende ejecutar un programa, además de ofrecerme la posibilidad de contribuir en la mejora de la herramienta que usan los alumnos de Informática de la UC.

La conclusión obtenida a raíz del desarrollo del proyecto es la siguiente: es posible reorganizar las secciones en las que se estructuran los binarios ejecutables de una aplicación. Es por tanto posible posicionar una sección cuyo único contenido sea un espacio vacío, con el fin de sobrescribirlo con el código que se pretende depurar.

Como se ha ido mostrando en las pruebas realizadas, el proceso a seguir se basa en declarar una sección de forma manual donde se reserva el espacio pertinente. Una vez declarada la sección, esta puede ser reubicada en primera posición a través del *linker script*. De este modo es posible copiar el programa que se quiera ejecutar siempre y cuando se obvien los primeros 0x48 bytes, correspondientes a la cabecera ELF.

Durante el desarrollo del trabajo he podido complementar los conocimientos que he adquirido en el transcurso de mis estudios en el Grado de Ingeniería Informática, especialmente de aquellos relacionados con la rama de Ingeniaría de Computadores. He podido profundizar en aspectos como la estructura de un ejecutable, su descomposición en secciones y qué utilidad tienen cada una de ellas.

He desarrollado todo el trabajo mayoritariamente en lenguaje ensamblador bajo el sistema operativo RISC OS. Este sistema operativo resulta muy adecuado para el desarrollo de prácticas por parte del alumnado, pero cuenta con una documentación muy limitada. Esto ha provocado que comprender ciertas mecánicas, especialmente relativas al enlazado de código, me haya resultado costoso, teniendo que recurrir constantemente a la experimentación, elaborando hipótesis las cuales comprobar a través de las pruebas desarrolladas. Por este motivo, considero que la información obtenida puede ser especialmente provechosa de cara a modificaciones y trabajo futuro.

5.1. Propuesta de mejora

A raíz de la realización del proyecto se han descubierto un conjunto de mejoras que pueden ser implementadas en el depurador como líneas de trabajo futuro.

La primera modificación propuesta, cuya viabilidad ha sido el objeto de estudio de este proyecto, consiste en modificar el depurador de acuerdo a los pasos indicados para posibilitar el depurado de código sin requerir el enlazado a la dirección *0x18088*. Como se ha demostrado, esta modificación requiere de la reubicación de secciones de código del depurador, pero también de la modificación de todos los métodos y funciones del mismo que referencien el espacio de memoria donde se almacena el código del alumno.

Durante el análisis de las secciones que componen el fichero ejecutable se ha podido localizar la sección que contiene la tabla de símbolos, y por tanto es posible determinar su dirección exacta a través de la tabla de cabeceras de sección. Podría resultar interesante incluir en el depurador la opción de mostrar una nueva ventana con el contenido de la tabla de símbolos, de modo que los alumnos que quieran conocer la ubicación de algún símbolo puedan hacer uso de ella.

Además, se ha observado que el depurador accede a las secciones del código del alumno por medio de desplazamientos fijos, definidos en el propio depurador. Esta estrategia ha funcionado hasta el momento ya que los códigos depurados son desarrollados íntegramente en ensamblador, y poseen una funcionalidad sencilla. Esto supone que el tamaño de las cabeceras de sección y segmentos sean iguales en todos los códigos a depurar. Sin embargo, una estrategia más sólida sería obtener los desplazamientos de las secciones a las que se quiere acceder consultándolas en la tabla de cabeceras ELF, asegurando de este modo que cualquier código, sin depender de su estructura, pueda ser ejecutado.

Por otra parte, el depurador no valora si el tamaño del fichero ejecutable en memoria es igual al del binario generado. Esta diferencia de tamaños es producida por secciones como la sección de datos sin inicializar (*".bss"*), que no ocupa espacio en el binario, pero requiere de espacio reservado en memoria durante la ejecución. Actualmente esto no supone un gran inconveniente debido a la reducida complejidad de los códigos desarrollados por los alumnos, pero tener esta posibilidad en consideración sería una buena práctica en el desarrollo del depurador. Además, esta modificación posibilitaría el depurado de códigos híbridos, desarrollados por ejemplo combinando lenguaje C y ensamblador.

Bibliografía.

[1] ANSI. *ANSI (1975-12-01). ISO-IR-6: ASCII Graphic character set. ITSCJ/IPSJ.* URL: https://www.itscj-ipsj.jp/ir/006.pdf.

[2] Ali Bahrami. *GNU Hash ELF Sections. Último acceso: Julio 2022.* URL: https://blogs.oracle.com/solaris/post/gnu-hash-elf-sections.

[3] Ali Bahrami. *Inside ELF Symbol Tables. Último acceso: Julio 2022.* URL: http://www.linker-aliens.org/blogs/ali/entry/inside_elf_symbol_tables/.

[4] Fernando Vallejo. Carmen Martínez. *Informe Final del proyecto de innovación docente: Docencia de Estructura y Organización de Computadores basada en la Arquitectura ARM, III Convocatoria de Proyectos de Innovación Docente de la Universidad de Cantabria.* URL: https://web.unican.es/consejo-direccion/vcprimeroyprofesorado/Documents/InnovacionDocente/Arm.pdf/.

[5] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. Mayo 1995. Version 1.2.* URL: https://refspecs.linuxfoundation.org/elf/elf.pdf.

[6] Ulrich Drepper. *How to write shared libraries. 19 de Noviembre, 2002.* URL: https://www.ukuug.org/events/linux2002/papers/pdf/dsohowto.pdf.

[7] Pablo Fuentes. *Repositorio de github del UCDebug. Último acceso: Julio 2022.* URL: https://github.com/fuentesp/UCDebug.

[8] gnu.org. *Command Language: Linker Scripts. Último acceso: Julio 2022.* URL: https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_3.html#SEC17.

[9] gnu.org. *GCC, the GNU Compiler Collection. Último acceso: Julio 2022.* URL: https://gcc.gnu.org/.

[10] ARM Limited. *ARM Architecture Reference Manual.* URL: https://developer.arm.com/documentation/ddi0487/latest.

[11] ARM Limited. *Arm limited roadshow slides Q2 2020.* URL: https://group.softbank/system/files/pdf/ir/presentations/2020/arm-roadshow-slides_q2fy2020_01_en.pdf.

[12] Bastian Molkenthin. *Online ELF Viewer. Último acceso: Julio 2022.* URL: http://www.sunshine2k.de/coding/javascript/onlineelfviewer/onlineelfviewer.html.

[13] Oracle. *Linker and Libraries Guide. Noviembre 2011.* URL: https://docs.oracle.com/cd/E23824_01/pdf/819-0690.pdf.

[14] Oracle. *Solaris x86 Assembly Language Reference Manual. Marzo 2010.* URL: https://docs.oracle.com/cd/E19253-01/817-5477/817-5477.pdf.

[15] Notepad++ org. *Notepad++. Último acceso: Julio 2022.* URL: https://notepad-plus-plus.org/.

[16] RISC OS. *Programmer's Reference Manual: Memory Management. Último acceso: Julio 2022.* URL: http://www.riscos.com/support/developers/prm/memoryman.html.

[17] RISC OS. *StrongED. Último acceso: Julio 2022.* URL: http://stronged.iconbar.com/.

[18] osdev.org. *Creating a C Library: Program Initialization. Último acceso: Julio 2022.* URL: https://wiki.osdev.org/Creating_a_C_Library#Program_Initialization.

[19] C. Martínez P. Fuentes C. Camarero y V. Mateev F. Vallejo D. Herreros. *Addressing Student Fatigue in Computer Architecture Courses. IEEE Transactions on Learning Technologies. IEEE. Marzo 2022. DOI: 10.1109/TLT.2022.3163631.*

[20] C. Martínez P. Fuentes C. Camarero y F. Vallejo. *Tecnología low-cost para motivar al alumno. Actas de la XXV Edición de las Jornadas sobre la Enseñanza Universitaria de la Informática (JENUI 2019), Murcia, 3-5 Julio 2019. ISSN: 2531-0607.* URL: https://repositorio.unican.es/xmlui/bitstream/handle/10902/18474/Tecnolog%c3%adalow-cost.pdf?sequence=3&isAllowed=y.

[21] F. Vallejo P. Fuentes C. Camarero y C. Martínez. *La importancia del uso de hardware real en la docencia de Estructura y Organización de Computadores. XVI Foro Internacional sobre la Evaluación de la Calidad de la Investigación y de la Educación Superior (FECIES 2019), Santiago de Compostela, 29-31 Mayo 2019.* URL: https://personales.unican.es/fuentesp/refs/FECIES'19.pdf.

[22] Universidad Estatal Electrotécnica de San Petersburgo. *GNU development tools man-pages. Último acceso: Julio 2022.* URL: `https://studfile.net/preview/429551/`.