INTEGRACIÓN DE LUA SOBRE ZEPHYR PARA DISPOSITIVOS SATELITALES

INTEGRATION OF LUA ON TOP OF ZEPHYR FOR SATELLITE DEVICES



TRABAJO FIN DE MÁSTER
CURSO 2021-2022

AUTOR
ÓSCAR BASELGA LAHOZ

DIRECTORES
LUIS PIÑUEL MORENO
CHRISTIAN TOMAS TENLLADO VAN DER REIJDEN

CONVOCATORIA: SEPTIEMBRE 2022
CALIFICACIÓN: 9

MÁSTER EN INTERNET DE LAS COSAS
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

RESUMEN

INTEGRACIÓN DE LUA SOBRE ZEPHYR PARA DISPOSITIVOS SATELITALES

Satlink es una empresa tecnológica que desarrolla soluciones IoT para la industria pesquera con el fin de alcanzar una gestión sostenible y eficiente de los recursos oceánicos. Desde 2013, ha estado trabajando en una plataforma para el desarrollo de servicios en un dispositivo satelital en colaboración con el grupo de investigación ArTeCS ("Architecture and Technology of Computing Systems") de la UCM (Universidad Complutense de Madrid). Inicialmente, la plataforma fue diseñada en eCos, pero con el auge de IoT han surgido RTOS más orientados a esta tecnología. Este trabajo surge de la necesidad de portar el sistema de eCos a Zephyr, un RTOS libre más moderno.

Los servicios implementados, que definen el comportamiento de los dispositivos de Satlink, son escritos en el lenguaje Lua, por lo que el primer paso de la portación consiste en integrar Lua sobre Zephyr. Incluir un lenguaje de scripting en un RTOS es cada vez más habitual, y Zephyr no proporciona una solución sencilla. Por ello, se consideró una buena oportunidad para desarrollar una integración consistente, formalizarla en un módulo y presentar una propuesta de publicación a Zephyr Project.

Al tratarse de dispositivos satelitales, estudiar el hardware dedicado a este tipo de comunicaciones se convierte en un punto de interés para el desarrollo. Por esta razón, se investigaron algunas de las soluciones propuestas por Iridium (proveedor con el que trabaja Satlink) para ser utilizadas en sistemas embebidos.

Palabras clave

IoT, Zephyr, Zephyr Project, Lua, RTOS, CMake, libc, Iridium

ABSTRACT

INTEGRATION OF LUA ON TOP OF ZEPHYR FOR SATELLITE DEVICES

Satlink is a technology company that develops IoT solutions for the fishing industry in order to achieve a sustainable and efficient management of ocean resources. Since 2013, it has been working on a platform for the development of services on a satellite device in collaboration with the ArTeCS ("Architecture and Technology of Computing Systems") research group at UCM (Complutense University of Madrid). Initially, the platform was designed in eCos, but with the rise of IoT, more IoT-oriented RTOS have emerged. This project arises from the need to port the eCos system to Zephyr, a more modern and open-source RTOS.

The implemented services, which define the behaviour of Satlink devices, are written in the Lua language, so the first step of the porting is to integrate Lua on top of Zephyr. Including a scripting language in an RTOS is becoming more and more common, and Zephyr does not provide a simple solution. Therefore, it was considered a good opportunity to develop a consistent integration, formalise it in a module and submit a proposal for publication to the Zephyr Project.

As these are satellite devices, studying the hardware dedicated to this type of communications becomes a point of interest for development. For this reason, some of the solutions proposed by Iridium (the provider with which Satlink works) for use in embedded systems were investigated.

Keywords

IoT, Zephyr, Zephyr Project, Lua, RTOS, CMake, libc, Iridium

ÍNDICE DE CONTENIDOS

Resumen III

Abstract V

Índice de contenidos VII

Índice de figuras IX

Índice de tablas X

Capítulo 1 - Introducción 1

1.1 Motivación 1

1.2 Objetivos 2

1.3 Plan de trabajo 3

1.4 Estructura del documento 4

Capítulo 2 - Estado de la cuestión 5

2.1 Lua 5

2.2 CMake 6

2.3 *libc* 7

2.4 Zephyr 8

2.5 ESP32 DevKitC y NXP FRDM-K64F 17

2.6 Iridium 20

2.7 Versiones de las herramientas 21

Capítulo 3 - *newlib* en Zephyr 23

3.1 Posibles integraciones 23

3.2 Configuración 23

3.3 Ejemplo 24

Capítulo 4 - Lua en Zephyr....25
4.1 Importación del proyecto Lua....25
4.2 Integración desde CMake....26
4.3 Error y solución en *newlib*....29
4.4 Módulo *liblua*....31
4.5 Requisitos de un módulo para ser candidato....35
4.6 Proceso de publicación....37
Capítulo 5 - Iridium en sistemas embebidos....39
5.1 Integración de comunicaciones satelitales....39
5.2 Interfaz de comandos en un módem satelital....40
5.3 Iridium en los dispositivos de Satlink....42
5.4 Recursos para el desarrollo con Iridium....43
Capítulo 6 - Resultados....49
6.1 Funcionamiento del módulo *liblua*....49
6.2 Estado actual de la propuesta a Zephyr Project....51
6.3 Pruebas con el emulador Iridium 9602/9603....51
Capítulo 7 - Conclusiones y trabajo futuro....55
Chapter - Introduction....57
Chapter - Conclusions and future work....61
Bibliografía....63
Apéndices....67

ÍNDICE DE FIGURAS

Figura 1-1: Arquitectura del sistema diseñada por ArTeCS para Satlink..... 2

Figura 2-1: Arquitectura de la pila de protocolos de Zephyr 11

Figura 2-2: Arquitectura del sistema Zephyr 13

Figura 2-3: Proceso de compilación - Fase de configuración.....15

Figura 2-4: Periféricos soportados por Zephyr en microprocesadores ESP32 18

Figura 2-5: Esquema pinout del ESP32 DevKitC v4.....19

Figura 4-1: Proceso de publicación de un módulo en Zephyr Project 38

Figura 5-1: Iridium 9603 Module en comparación de tamaño con un lapicero 40

Figura 5-2: Módulo RockBLOCK Mk2 de Iridium 43

Figura 5-3: Kit de desarrollo 9603 de Iridium 45

Figura 5-4: Pines desoldados en el kit de desarrollo 9603 45

Figura 5-5: Arquitectura de las comunicaciones reales con Iridium.....47

Figura 5-6: Arquitectura de las comunicaciones con Iridium a través de un emulador..47

ÍNDICE DE TABLAS

Tabla 2-1: Comparativa Lua vs Python 6

Tabla 6-1: Características de los modelos del módulo RockBLOCK Mk2 de Iridium 44

Capítulo 1 - Introducción

1.1 Motivación

Satlink [1] es una empresa tecnológica española centrada en el desarrollo de soluciones de Internet de las Cosas (IoT). Como proveedor de soluciones tecnológicas para el segmento pesquero, Satlink ayuda a optimizar la toma de decisiones brindando acceso a datos de interés para llevar a cabo una gestión más sostenible y eficiente de los recursos oceánicos.

Desde finales del año 2013, el grupo de investigación ArTeCS ("Architecture and Technology of Computing Systems") perteneciente al departamento de Arquitectura de Computadores y Automática de la UCM (Universidad Complutense de Madrid) ha estado trabajando en una "Plataforma HW/SW para el desarrollo de servicios de un dispositivo satelital sobre un lenguaje de programación imperativo" [2] junto a Satlink. En la Figura 1-1 aparece la arquitectura que diseñó ArTeCS para dicha plataforma, en la cual se distingue: la parte hardware, compuesta por el microprocesador y los componentes dedicados a las comunicaciones, y la parte software, que se implementa en varias capas. Las capas superiores ("Common Services", "User Services" y "Service Framework + Extensions") se encargan de implementar la lógica de la aplicación y, por tanto, es donde se determina el comportamiento del equipo que Satlink va a desarrollar. Esta lógica se define a través de servicios, es decir, fragmentos de códigos que ejecutan una funcionalidad muy concreta. Los servicios son escritos en Lua, un lenguaje de scripting enfocado a sistemas embebidos, por lo que la siguiente capa ("Lua Compiler & Interpreter") corresponde al intérprete y al compilador que va a permitir su ejecución. Por último, la capa inferior ("Firmware + Operating System") alberga el sistema operativo y el firmware necesario para interactuar con el hardware de la plataforma y se trata del objeto de estudio de este trabajo.

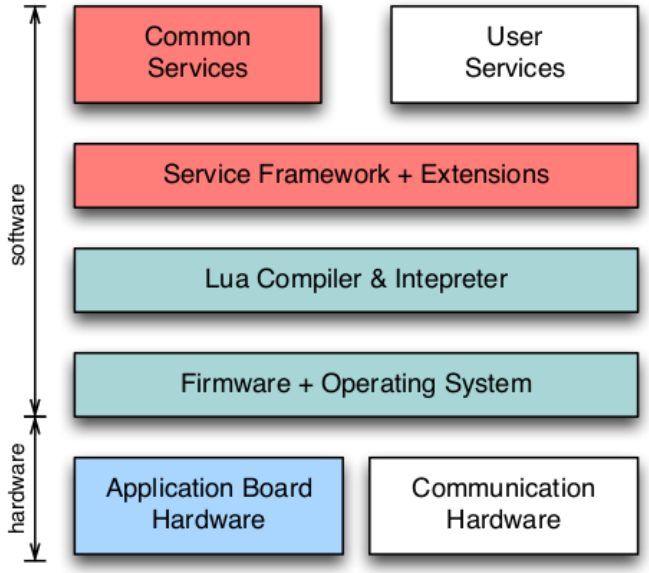


Figura 1-1: Arquitectura del sistema diseñada por ArTeCS para Satlink

1.2 Objetivos

La idea de ArTeCS y Satlink fue desarrollar el sistema sobre un RTOS así que decidieron comenzar con eCos, el cual puede ofrecer numerosas ventajas pero que, actualmente, mantiene una licencia propietaria y un bajo nivel de adaptación a las tecnologías IoT que han surgido en los últimos años. Por esta razón y con vistas a mejorar la robustez y flexibilidad del sistema, decidieron cambiar a una tecnología más reciente. Así, el objetivo que se propuso para este trabajo fue empezar a portar el sistema desde eCos a Zephyr, un RTOS moderno y libre enfocado en el desarrollo de dispositivos IoT.

En concreto, se plantearon los siguientes objetivos:

1. Alcanzar un alto nivel de entendimiento de Zephyr para poder manejarlo desde sus bases.
2. Realizar una integración funcional de Lua sobre Zephyr.
3. Proponer una solución estable, en forma de módulo, a Zephyr Project con el fin de que sea incluida en el proyecto.
4. Trabajar con el hardware dedicado a las comunicaciones satelitales.

1.3 Plan de trabajo

Para lograr estos objetivos, primero había que comprender la implementación del sistema actual de Satlink y conocer sus puntos de interés para empezar a portarlo de la mejor manera posible a la nueva tecnología. Esta fase de comprensión del sistema que Satlink está integrando en sus productos fue realizada durante las prácticas en empresa, periodo durante el cual se realizó un acercamiento desde su mayor nivel de abstracción para entender los servicios Lua que desarrollan y la solución final que ofrecen.

Con una idea ya establecida del sistema en su conjunto, se comenzó con este trabajo, consistiendo la primera parte en terminar de aprender el lenguaje Lua. A continuación, se lanzó el primer objetivo de aprender a manejar Zephyr. Este RTOS es en el que se iba a basar todo el sistema y era necesario conocer sus puntos fuertes y débiles para poder explotarlos más adelante. La dificultad de este objetivo se veía incrementada porque no solo era necesario saber utilizar el RTOS, sino también conocer aspectos que habitualmente pasan a un segundo plano, como es su proceso de compilación, su jerarquía de los recursos, sus componentes disponibles para las aplicaciones, etc.

Una vez se conocían lo suficiente las herramientas que se iban a utilizar, había que comenzar a portar el sistema. Para ello, lo esencial era conseguir integrar Lua sobre Zephyr, pues a partir de ese momento se podrían portar los servicios implementados por Satlink al nuevo sistema. Se empezó por una integración sencilla hasta entrar en el último de los objetivos: formalizar la solución. De aquí en adelante, se estudiaron los requisitos que debía de cumplir un módulo para que fuera considerado por Zephyr y se inició el proceso para realizar la propuesta.

Por último, se entró a estudiar las soluciones que ofrecía Iridium para conocer la forma en la que se podía trabajar con ellos en un sistema embebido. Hecho esto, se planteó realizar algunas pruebas con un kit de desarrollo de Iridium para poner en práctica las técnicas de desarrollo en este tipo de hardware.

1.4 Estructura del documento

Este documento se divide en distintos capítulos:

- En el Capítulo 1 se resume las ideas iniciales del origen del trabajo, como la motivación, los objetivos planteados, el plan que se seguirá y la estructura en la que se organiza este documento.
- En el Capítulo 2 se describen las principales herramientas que han sido utilizados durante el trabajo y que es necesario conocer bien para poder comprender las tareas que se han realizado. En concreto, se habla de: Lua, el lenguaje de scripting que integran los equipos de Satlink; CMake, la herramienta a la que se delega el proceso de compilación en proyectos complejos; *libc*, la biblioteca estándar de C que resulta imprescindible de emplear en la mayoría de sistemas embebidos; Zephyr, el RTOS al que se quiere portar el sistema de Satlink y ArTeCS; las plataformas con las que se han trabajado, un ESP32 y una NXP; Iridium, el servicio satelital que trabaja con Satlink; y las versiones de las herramientas empleadas en el trabajo.
- En el Capítulo 3 se explican las distintas alternativas que ofrece Zephyr para la integración de estándares de C en una aplicación, además de aportarse un ejemplo de uso.
- En el Capítulo 4 se trata el proceso de integración de Lua sobre Zephyr y cómo se puede contribuir a un proyecto de código abierto, como es Zephyr Project, con esta solución.
- En el Capítulo 5 se detalla el proceso para incorporar y desarrollar comunicaciones satelitales en un sistema embebido. Se hará un repaso de las soluciones que ofrece Iridium y de cómo trabajar con ellas.
- En el Capítulo 6 se comentan los resultados finales y el estado actual de la propuesta de contribución.
- En el Capítulo 7 se realizan una serie de conclusiones, así como la evolución con la que debería continuar el trabajo.

Capítulo 2 - Estado de la cuestión

Este capítulo está enfocado a describir los recursos empleados durante el trabajo que fueron fundamentales. De todos ellos se destacan sus aspectos más importantes.

2.1 Lua

Como se mencionaba, los servicios que conforman las aplicaciones están escritos en Lua. Este lenguaje se trata de un lenguaje de scripting cuyas características más destacables son [3]:

- Su robustez, ya que ha sido empleado en numerosas aplicaciones (e.g., Adobe's Photoshop Lightroom), sistemas embebidos (e.g., el middleware Ginga para la TV digital en Brazil) y videojuegos (e.g., World of Warcraft y Angry Birds).
- Su portabilidad, dado que se basa en un pequeño paquete de código que es compilado out-of-the-box en cualquier plataforma con un compilador de C.
- Su integración en sistema embebidos, porque además de ser pequeño en memoria, ofrece una API para integrarlo en cualquier aplicación implementada en otro lenguaje.
- Su licencia MIT de software libre, que hace que pueda ser distribuido con fines comerciales sin ninguna restricción.

Sin embargo, Lua es un lenguaje bytecode, lo que significa que no se trata ni de un lenguaje interpretado ni compilado, sino que hace uso de una máquina virtual donde se ejecuta el código. Este mecanismo otorga una gran velocidad de ejecución gracias a la traducción directa de bytes que realiza del código y que es ejecutado directamente en la máquina virtual.

Si se compara Lua con otro lenguaje de programación (de la misma naturaleza) como por ejemplo Python, pueden obtenerse diferencias muy significativas, como las nombradas en la Tabla 2-1, a la hora de decantarse por uno de ellos.

Lua	Python
Lenguaje bytecode (más rápido)	Lenguaje interpretado (más lento)
Comunidad de usuarios pequeña	Comunidad de usuarios muy grande
Reducido soporte de bibliotecas	Amplio soporte de bibliotecas
Muy eficiente en memoria	Poco eficiente en memoria
Integración con C sencilla	Integración con C más complicada
No soporta manejo de excepciones	Soporta manejo de excepciones
Código abierto	Código abierto

Tabla 2-1: Comparativa Lua vs Python

2.2 CMake

CMake [4] es un sistema extensible y de código abierto que gestiona el proceso de compilación en un sistema operativo de manera aislada al compilador. A diferencia de muchos sistemas multiplataforma, CMake está diseñado para ser utilizado junto con el entorno de compilación nativo.

Esta herramienta funciona por medio de una serie de ficheros de configuración situados en cada directorio del proyecto a compilar. Estos ficheros reciben el nombre de "CMakeLists.txt" y son utilizados para generar scripts de compilación (por ejemplo, scripts Makefile en sistemas Unix). Como resultado de la ejecución de estos scripts, CMake genera un entorno de compilación para el código fuente del proyecto, creando bibliotecas (tanto estáticas como dinámicas) y generando los ejecutables.

Una característica muy útil de CMake es que genera un fichero en caché para emplear un editor gráfico. Así, cuando CMake se ejecuta, localiza ficheros, bibliotecas y ejecutables, y además encuentra directivas de compilación opcionales. Esta información es recogida en la caché y puede ser modificada por el usuario antes de la generación de los scripts de compilación a través de un editor gráfico.

Desde un principio, CMake ha sido diseñado para soportar jerarquías de directorios complejas y aplicaciones con dependencias de varias bibliotecas. Debido a su licencia de código abierto y su diseño simple y extensible, CMake puede ampliarse

con nuevas características para amoldarse a las necesidades del usuario. CMake es sencillo de utilizar, ya que cada uno de los ficheros CMakeLists.txt contiene un conjunto de comandos que son ejecutados para compilar el código que se encuentra en el mismo directorio. Se dice que CMake es extensible, entre otras cosas, porque el usuario no está obligado a utilizar los comandos proporcionados por CMake, sino que puede definir los suyos propios. Incluso los usuarios con más experiencia tienen la posibilidad de modificar los generadores de los scripts de compilación enfocándolos a una combinación particular de compilador y sistema operativo.

CMake es ampliamente utilizado en C y C++ y, por lo general, se suele emplear Makefile o Ninja como sistemas para ejecutar los scripts de compilación generados.

2.3 *libc*

libc [5] es como se conoce comúnmente a la biblioteca estándar de C ("standard C library"), la cual contiene un conjunto de funciones que pueden ser usadas por cualquier código C (y, ocasionalmente, también por códigos en otros lenguajes). Actualmente, *libc* es lo mismo que *glibc*, pero históricamente eran distintas implementaciones, ya que *libc* correspondía a una biblioteca más amplia pensada para el desarrollo de Linux y *glibc* para GNU.

La mayoría de las funciones de *libc* pueden ser llamadas directamente por el usuario, pero existen algunas que son internas y que el compilador usa para ejecutar operaciones no soportadas por el hardware (por ejemplo, operaciones en punto flotante).

Por otro lado, *newlib* [6] es una adaptación de la *libc* para sistemas embebidos. Esta implementación ha sido creada a partir de fragmentos de la *libc* original, algo que la hace más ligera (pero menos funcional) y, por consiguiente, apta para ser usada en plataformas bare-metal (sin sistema operativo) o en plataformas con un RTOS. Todavía existe una implementación más liviana que es conocida como *nanolib*, la cual aumenta el alcance de *libc* hasta las plataformas con las mayores limitaciones en memoria.

newlib puede ser compilada para multitud de procesadores siempre y cuando cuente con el BSP correspondiente. Un BSP ("Board Support Package") es un paquete de códigos propio de cada plataforma que implementa las funcionalidades básicas;

podría entenderse como los drivers de una plataforma embebida. De esta forma, *newlib* simplemente realiza llamadas al BSP que se encarga de llevar a cabo la tarea indicada.

Todas estas bibliotecas están sujetas a una licencia de software libre, por lo que su utilización no se ve restringida en ningún caso.

2.4 Zephyr

Zephyr es un sistema operativo de tiempo real basado en un ligero kernel que ha sido diseñado para ser usado en plataformas con recursos limitados, desde simples sensores ambientales hasta sofisticados controladores IoT. Además, el kernel de Zephyr soporta una gran variedad de arquitecturas, como pueden ser ARM, Intel x86, ARC, Nios II, Tensilica Xtensa, RISC-V, SPARC y MIPS [7]. Un aspecto interesante que es muy valorado en la actualidad es la licencia que rige el proyecto. En este caso, Zephyr se mantiene bajo una licencia Apache 2.0, lo que significa que está abierto a ser redistribuido con fines comerciales y a modificaciones del código original (en el Apéndice A aparece la descripción de la licencia) [8].

Zephyr es un RTOS que sigue todavía en desarrollo pero que ya cuenta con una base suficientemente consistente como para poder ser integrado en soluciones finales. Algunas de las características de las que dispone son [9]:

- Un amplio conjunto de servicios del kernel, donde destacan:
    - El soporte multitarea, que permite definir tareas de tipo cooperativo, basados en prioridades, preemptive y non-preemptive, integrando incluso un reparto del tiempo de CPU por mecanismos de time-slicing.
    - Los servicios de interrupción para registrar en tiempo de compilación los manejadores de las interrupciones que se consideren necesarios.
    - La gestión de la memoria, dando paso a la reserva y liberación de memoria en tiempo de ejecución.
    - Los mecanismos de sincronización de las tareas, pues ofrece semáforos binarios y no binarios y mutexes.

    - Los mecanismos de intercambio de información entre tareas, incluyendo colas de mensajes y flujos de bytes.
    - Los servicios de gestión de la energía para realizar optimizaciones en el consumo del dispositivo.
- Una selección de algoritmos de scheduling para las tareas, como:
    - El método cooperativo y preemptive, donde son las tareas quienes ceden el procesador cuando lo consideran oportuno o un scheduler se encarga de expulsarlas, respectivamente.
    - El algoritmo "Earliest Deadline First", que provoca que el scheduler priorice las tareas cuyo deadline es más cercano en el tiempo.
    - El mecanismo previamente mencionado time-slicing, que determina bloques de tiempo de CPU que son repartidos entre tareas preemptive con misma prioridad.
    - Distintas estrategias de colas, como son una cola simple enlazada, una árbol rojo-negro o múltiples colas agrupando tareas con necesidades de scheduling similares.
- Un entorno altamente flexible y configurable, ya que permite que cada aplicación incorpore únicamente los módulos y componentes que necesita, permitiendo también indicar los recursos disponibles para cada uno de ellos.
- Un amplio soporte de arquitectura cruzada, dado que es plenamente compatible con multitud de arquitecturas y herramientas de desarrollo. Además, las contribuciones de los usuarios están ayudando en el crecimiento de SoCs y plataformas de desarrollo compatibles con Zephyr.
- Un mecanismo muy completo de protección de la memoria. Un aspecto muy importante en muchos sistemas embebidos es la protección de la memoria, ya sea por temas de seguridad o de funcionamiento, por lo que Zephyr ofrece técnicas de detección de desbordamiento de pila, traqueo de permisos para el acceso a objetos del kernel y drivers de dispositivos y

un entorno de ejecución aislado para las tareas. En el Apéndice A, se proporciona más información acerca de la seguridad en Zephyr.

- La posibilidad de definir en tiempo de compilación los recursos del sistema, algo que reduce significativamente el tamaño de la aplicación y mejora el rendimiento de la plataforma.
- Un modelo optimizado para la gestión de los drivers de los dispositivos que facilita la inicialización y la configuración los dispositivos de la plataforma. Este modelo también permite reusar los drivers entre plataformas con mismos dispositivos.
- El uso de un devicetree encargado de describir el hardware de la plataforma, una información utilizada posteriormente en la creación de la imagen de la aplicación. El devicetree es una estructura jerárquica de datos que describe el hardware de cada una de las plataformas soportadas así como su configuración inicial.
- Una pila de protocolos de red muy amplia, una característica que hace destacar a Zephyr para sistemas IoT respecto a otros RTOS. En la Figura 2-1 aparece la arquitectura de la pila implementada.
- Una pila BLE 5.0 que incluye Bluetooth Mesh y un controlador Bluetooth basado en GAP, GATT y con interfaz HCI, que sumado a la pila de protocolos anterior, Zephyr gana todavía más fuerza en entornos IoT.
- Una CLI compatible con los sistemas operativos principales (Windows, Linux y macOS).
- Soporte a sistemas de ficheros virtuales LittleFS y FATFS.

- Un framework de gestión de logging, que optimiza la creación y almacenamiento de mensajes de log a distintos niveles, como puede ser a nivel de memoria, de red, de sistema de ficheros o de consola.

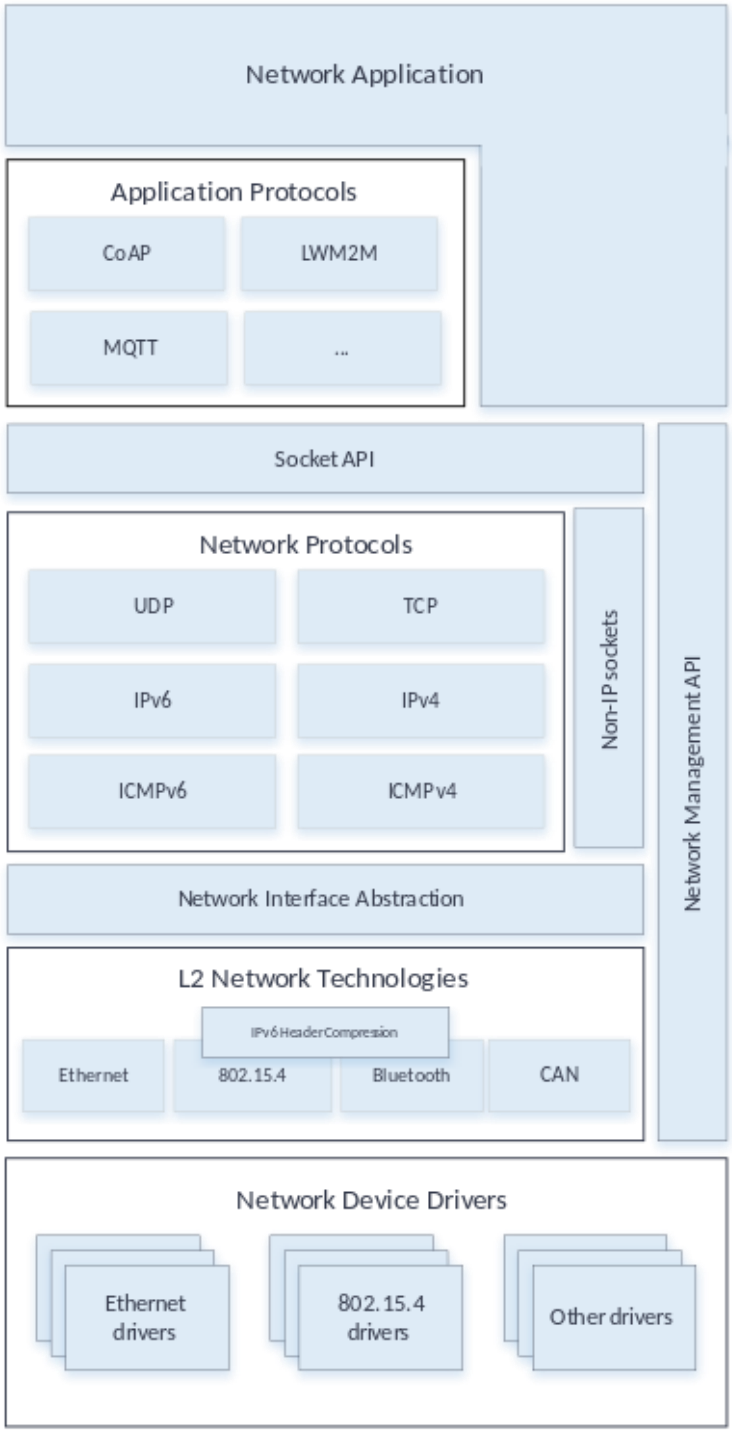


Figura 2-1: Arquitectura de la pila de protocolos de Zephyr [10]

- Un subsistema para la integración de una shell en la aplicación del usuario que ofrece una solución con autocompletado, wildcards, colores, historial y metakeys (barra espaciadora, flechas, backspace, CTRL+Z, etc.). Esta funcionalidad es muy útil en sistemas embebidos IoT porque habitualmente es necesario algún tipo de acceso desde el exterior para poder interactuar con el dispositivo.
- Un subsistema para el almacenamiento no volátil de los ajustes, de forma que cualquier módulo puede mantener de forma persistente su configuración o incluso la propia aplicación puede guardar distintos estados de ejecución.
- Un sistema NVS (Non-Volatile Storage), que al igual que el subsistema anterior, esta memoria permite almacenar variables o constantes de interés para la aplicación.
- Un puerto nativo de POSIX, lo que implica que Zephyr puede ejecutarse como un proceso más en un sistema Linux. Principalmente, esto suele ser interesante en entornos de validación.

Todas estas características se organizan en un arquitectura muy modulada y bien definida, que puede apreciarse en la Figura 2-2. En ella, se puede ver una separación en dos partes: el sistema operativo, compuesto por la capa del kernel ("kernel") y de los servicios del sistema operativo ("OS Services"), y el espacio de usuario, que lo conforma la capa de los servicios de aplicación ("Application services"). La primera de las partes contiene los drivers específicos de la plataforma e implementaciones para las distintas interfaces de E/S, funciones propias del kernel y la librería de criptografía, mientras que en la parte del usuario se encuentran servicios a más alto nivel, como implementaciones de los protocolos de red y modelos de datos para el desarrollo.

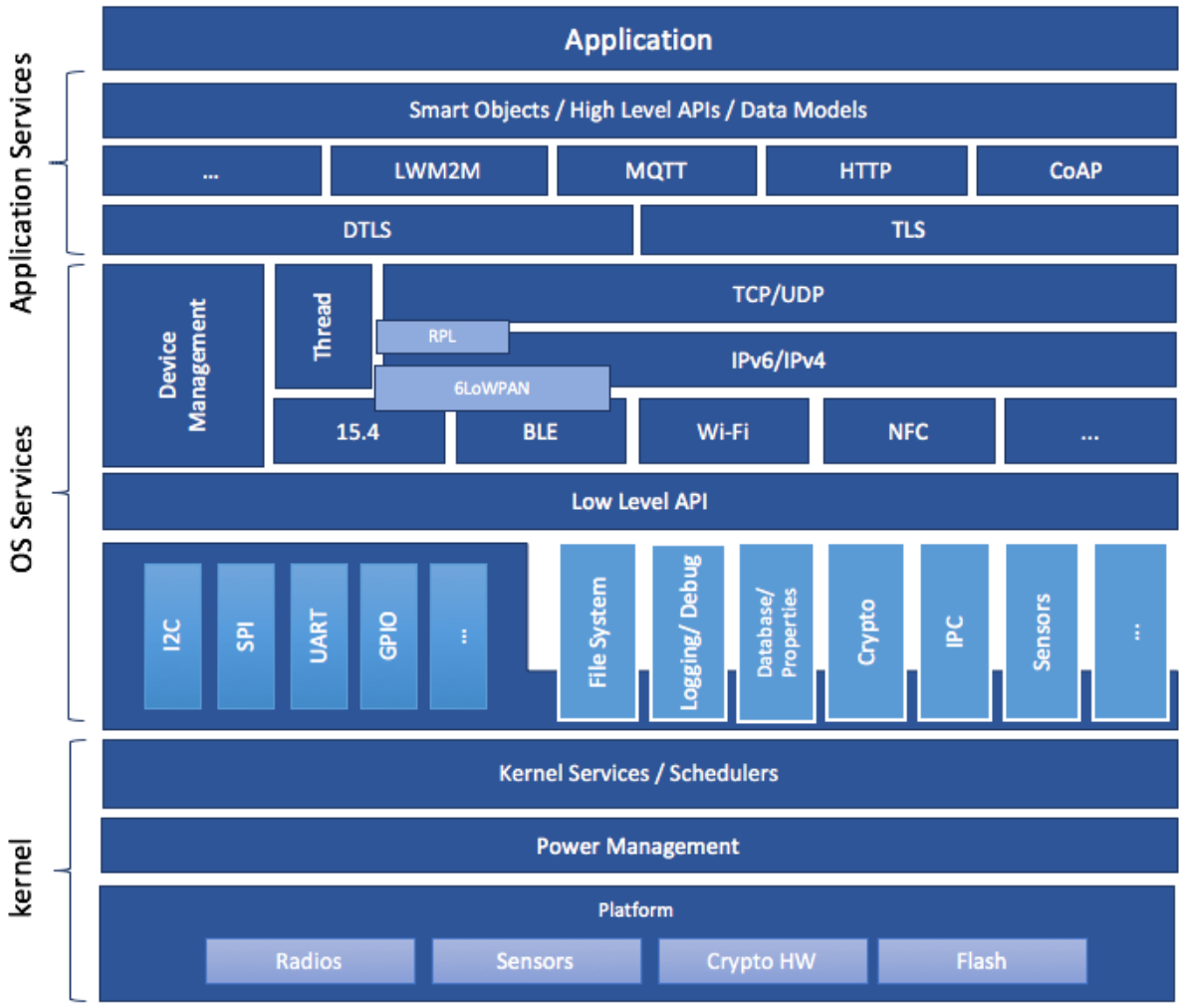


Figura 2-2: Arquitectura del sistema Zephyr [11]

En cuanto al proceso de compilación, Zephyr permite generar una única imagen con la aplicación y el kernel. El proceso se realiza en dos fases [12]: la de configuración y la de compilación.

La fase de configuración, dirigida por CMake, comienza cuando el usuario manda la orden de generar un sistema de compilación, para lo cual necesita indicar el directorio donde se encuentra el código de su aplicación y la plataforma hardware para la que quiere compilar dicha aplicación. En Figura 2-3, se muestra el proceso que sigue CMake en esta fase:

- Primero, CMake ejecuta todos los `CMakeLists.txt` del proyecto, comenzando por el de la aplicación que, de manera recursiva, va referenciando al resto que se encuentran distribuidos por el árbol de directorios de Zephyr. Como resultado de esta ejecución, se generan un conjunto de ficheros Makefile o Ninja (en función del sistema de compilación empleado en la siguiente fase) que van a guiar el proceso de compilación más adelante.
- Segundo, los ficheros con extensiones `.dts` y `.dtsi` ("devicetree source" y "devicetree source include", respectivamente) son tomados del código de la plataforma destino que contiene Zephyr y fusionados en un único fichero con el devicetree resultante (`zephyr.dts.pre`). Esta tarea es llevada a cabo por el preprocesador de C.
- Tercero, el devicetree preprocesado es parseado para generar la cabecera `devicetree_unfixed.h` con las macros de preprocesador. Este fichero es incluido en una cabecera general (`devicetree.h`) que, a su vez, debe ser incluida en el código fuente de la aplicación para que el usuario pueda acceder al hardware de la plataforma. Además, durante el parseado del devicetree, se escribe el devicetree final en el fichero `zephyr.dts`, que es utilizado por el dtc ("devicetree compiler") para la depuración.
- Cuarto, los ficheros `dts_fixup.h` del código de la plataforma destino son concatenados para generar un único fichero `devicetree_fixups.h`, el cual también será incluido en la cabecera general `devicetree.h` para complementar a `devicetree_unfixed.h`.

- Quinto, todas las opciones de configuración del sistema (definidas en ficheros `Kconfig`) y de la aplicación (definidas en el fichero `prj.conf`) son fusionadas para generar los ficheros `autoconf.h` y `.conf`, que son usados más tarde por CMake.

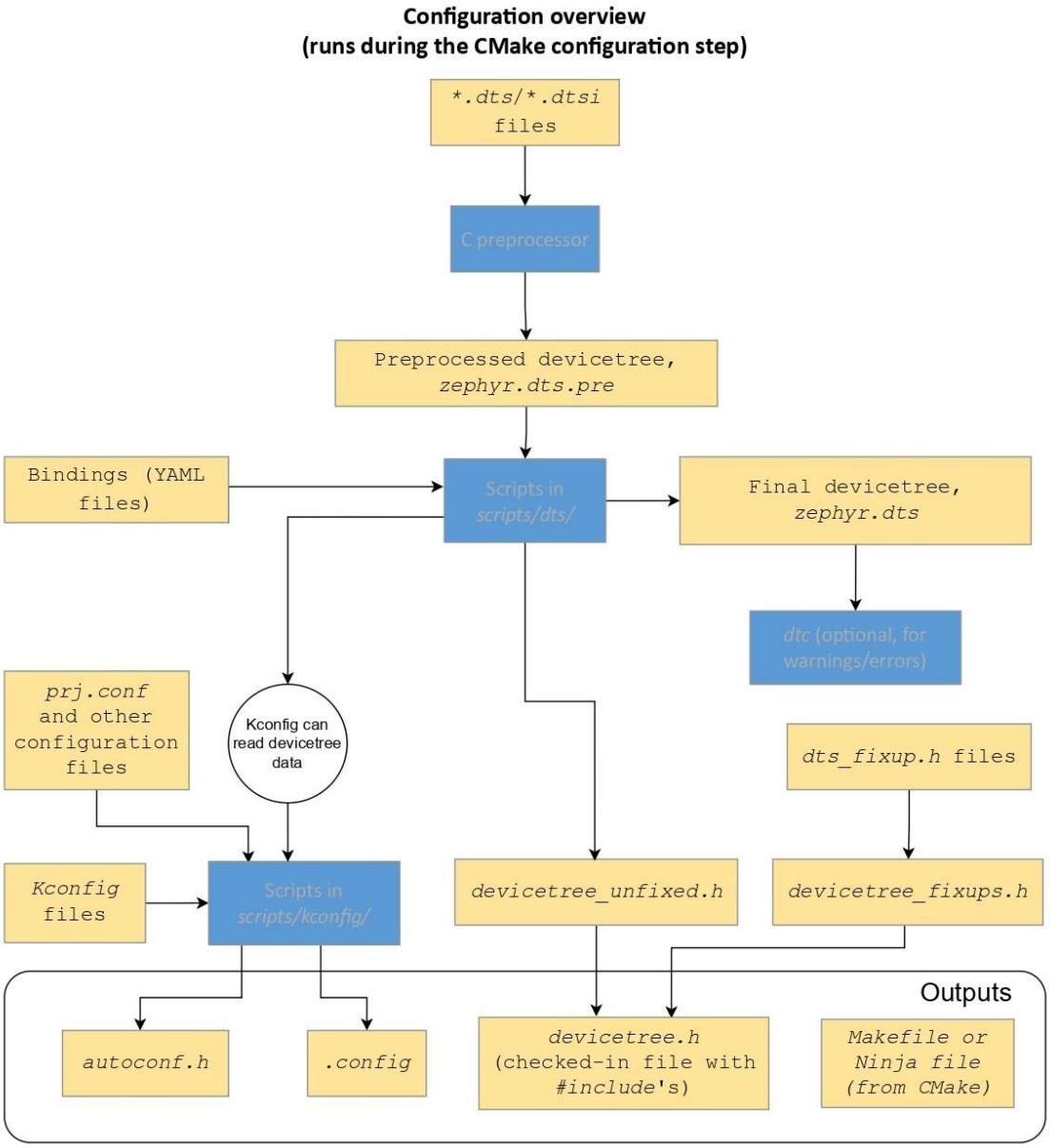


Figura 2-3: Proceso de compilación - Fase de configuración [12]

La fase de compilación, dirigida por Ninja o CMake, comienza cuando el usuario ejecuta el comando "ninja" o "make", es decir, cuando invoca al sistema de compilación. En ese momento, se lanzan los scripts de compilación que fueron generados en la fase de configuración. Como resultado de esta fase, se obtiene el binario con el kernel y la aplicación del usuario que será finalmente cargado en la plataforma para la que fue compilada.

Uno de los puntos fuertes de Zephyr es su CLI, una herramienta que facilita en gran medida el desarrollo debido a que provee un conjunto de comandos para poder trabajar con el workspace del proyecto. La CLI recibe el nombre de "west" y dispone de su propio repositorio independiente [13].

En Zephyr, como en la mayoría de tecnologías, el usuario necesita un workspace o entorno de trabajo para desarrollar su proyecto. La organización de un workspace de Zephyr sigue la siguiente estructura [14]:

```
zephyrproject/
      ├── .west/
      │     └── config
      ├── zephyr/
      │     ├── [... other files ...]
      │     └── west.yml
      ├── modules/
      │     ├── [... other projects ...]
      │     └── lib/
      │           ├── [... other projects ...]
      │           └── tinycbor/
      ├── tools/
      │     └── [ ... other projects ...]
      └── bootloader/
            └── [ ... other files ...]
```

Como se puede ver en el esquema, existen varios directorios principales: `.west`, que contiene un fichero de configuración con los ajustes locales del workspace (principalmente, la ruta del directorio raíz y del fichero manifest del workspace); `zephyr`,

con el código base de Zephyr (kernel, drivers, devicetrees, bibliotecas, scripts, etc.) y donde se encuentra el fichero manifest del workspace; `modules`, donde se exportan los proyectos externos que han sido integrados por Zephyr, estos son, repositorios de Git que han sido anexados al proyecto principal (un ejemplo de proyecto externo es TinyCBOR, una implementación concisa de representación de objetos binarios); `tools`, que engloba herramientas útiles de desarrollo (por ejemplo, para comunicarse desde el host con una máquina virtual de Zephyr corriendo en QEMU); `bootloader`, con el código del bootloader que utiliza Zephyr.

El fichero manifest `zephyr/west.yml` contiene la definición de los proyectos externos que son traídos al workspace de Zephyr, es decir, que cuando la herramienta "west" inicialice o actualice un nuevo workspace (con el comando "west init" o, si ya había sido creado el workspace, con "west update"), lo primero que va a hacer es obtener de este fichero una lista con todos los repositorios de Git que tendrá que clonar localmente. Así, Zephyr no solo contará con la funcionalidad que implementa de forma nativa, sino también con la funcionalidad integrada de cada uno de estos repositorios. Esta idea de integración es muy importante, tal y como se verá más adelante.

2.5 ESP32 DevKitC y NXP FRDM-K64F

Durante la realización de este trabajo, se han empleado dos plataformas de desarrollo: un ESP32 DevKitC v4 y una NXP FRDM-K64F.

Inicialmente, dado que en las prácticas de laboratorio del máster se había trabajado con el ESP32 DevKitC v4, se decidió mantenerlo por temas de familiaridad y sencillez. Sin embargo, se supo que llegaría un punto que debería sustituirse ya que, como se ha dejado ver anteriormente, Zephyr es un RTOS moderno y por tanto existen soportes de algunas plataformas que todavía no han sido desarrollados por completo. Es el caso de ESP32 que, como se puede ver en la Figura 2-4, tiene periféricos que en Zephyr todavía no son ni siquiera compatibles.

Peripheral	ESP32	ESP32-S2	ESP32-C3
CPU	Yes	Yes	Yes
IRQ	Yes	Yes	Yes
Timers	Yes	Yes	Yes
UART	Yes	Yes	Yes
I2C	Yes	Yes	Yes
SPI Flash	Yes	Yes	Yes
SPI Flash (Octal)	N/A	N/A	N/A
SPI RAM	Yes	Yes	N/A
SPI RAM (Octal)	N/A	N/A	N/A
SPI (general)	Yes	Yes	Yes
Cryptography	No	No	No
Wi-Fi	Yes	Yes	Yes
Bluetooth	Yes	N/A	Yes
GPIO	Yes	Yes	Yes
TWAI	Community	No	No
E-FUSE	Yes	No	No
ADC	Community	No	No
DAC	No	No	N/A
MCPWM	No	No	No
LEDPWM	Yes	No	No
TRNG	Yes	Yes	Yes
DMA	No	No	No
Watchdog	Yes	Yes	Yes
Low Power	WIP	WIP	WIP
RTC	No	No	No
USB OTG	N/A	No	N/A
USB CDC	N/A	No	Community
ETH MAC	No	N/A	N/A
SDIO	No	N/A	N/A
I2S	No	No	No
ULP	No	No	No
SMP	Conditional	N/A	N/A
AMP	Beta	N/A	N/A
Flash Encrypt.	Beta	Beta	Beta
DFS	No	No	No
MCUboot	Yes	Yes	Yes
OpenOCD	Yes	Yes	Yes
Secure Boot	Yes	Yes	Yes

Figura 2-4: Periféricos soportados por Zephyr en microprocesadores ESP32 [15]

Aun así, gran parte del trabajo se llevó a cabo con este microprocesador, el cual cuenta con un pinout muy completo (véase en la Figura 2-5) y un framework muy intuitivo, como es ESP-IDF con el que también se trabajó en las prácticas de laboratorio del máster.

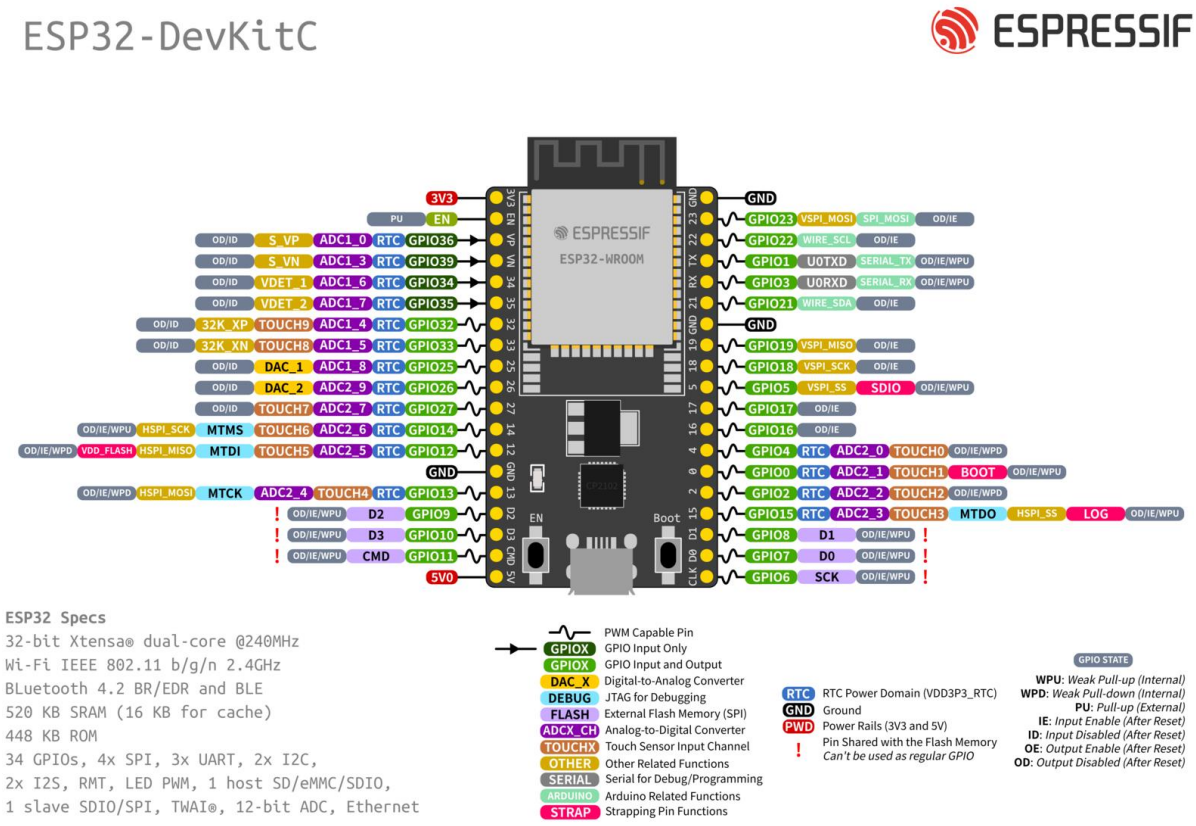


Figura 2-5: Esquema pinout del ESP32 DevKitC v4 [16]

Más adelante, debido a un error por recursos en el ESP32, se pasó a trabajar con la plataforma NXP FRDM-K64F. Este microprocesador es completamente compatible con Zephyr [17], por lo que se consiguió evitar problemas futuros derivados de compatibilidad y soporte de la plataforma.

2.6 Iridium

Habitualmente, las instituciones y las empresas privadas deciden implantar una constelación de satélites con el fin de gestionar ellos mismos, sin terceros involucrados, el servicio satelital que quieran proveer. Una constelación de satélites consiste en un grupo de satélites que trabajan juntos formando un único sistema, consiguiendo así una alta disponibilidad del servicio, ya que ofrecen cobertura global o cuasiglobal permanente gracias a que al menos un satélite es visible en cualquier momento y desde cualquier punto sobre la Tierra.

Iridium [18] es el nombre de una de estas constelaciones. En concreto, se trata de una constelación formada por 66 satélites que giran alrededor de la Tierra siguiendo órbitas LEO ("Low Earth Orbit"), las cuales se encuentran a una altura aproximada de 780 kilómetros, que fue diseñada por Motorola en el año 1997. Su objetivo fue el de proveer un servicio de comunicación por voz y datos mediante el uso de dispositivos portátiles específicos en áreas situadas fuera del rango de los sistemas de comunicación tradicional como la celular. A continuación, se listan algunos de sus casos de uso [19]:

- En deportes extremos se utiliza para aumentar la seguridad del usuario. Por ejemplo, a los pilotos de la aviación o navegantes de embarcaciones les permite mantenerse comunicados en todo momento, por muy remota que sea su localización.
- En expediciones, tanto en los polos como en la selva o en el desierto, los participantes pueden comunicarse con cualquier otro usuario, ya sea del equipo que le acompaña o de su familia.
- En el seguimiento de mercancías por cualquier medio de transporte puede realizarse una monitorización de su trayectoria y de su estado actual.
- En vehículos autónomos porque pueden ser controlados remotamente gracias a la comunicación bidireccional que ofrece Iridium.
- En todo tipo de comunicación aérea y marítima, sobre todo a niveles industriales, es fundamental para poder comunicarse con el exterior.

2.7 Versiones de las herramientas

El desarrollo del trabajo se ha llevado a cabo en una máquina Ubuntu donde se han instalado todas las herramientas necesarias. Por parte de Zephyr, se instaló su versión 3.1.99, la versión 0.13.1 de su SDK y la versión 0.13.1 de la CLI "west". Además, fue imprescindible incluir el compilador DTC ("devicetree compiler") en su versión 1.6.0 y el compilador GNU de C versión 10.3.0. También se utilizó la versión 3.8.10 de Python. Durante el trabajo se emplearon dos plataformas diferentes: primero el ESP DevKitC v4, para la cual se utilizó la versión 4.3.0 de su toolchain ESP-IDF, y segundo la NXP FRDM-K64F. Por último, para el desarrollo del módulo *liblua* se decidió tomar la última versión de Lua, que corresponde a la 5.4.4.

Por la parte satelital, se trabajó con un emulador del módem Iridium 9602/9603 [20] debido a que el hardware físico que se iba a utilizar, el kit de desarrollo 9603, dejó de funcionar. La implementación de este emulador se llevó a cabo con Python2, por lo que para ejecutarlo fue necesario retornar a la versión 2.7.18 de Python, con la que se instalaron también los módulos `pyserial` y `sendmail`. Con el fin de poder lanzar el emulador localmente, se empleó la herramienta `socat` de Linux en su versión 1.7.3.3 para crear puertos serie virtuales.

Capítulo 3 - *newlib* en Zephyr

En este capítulo se explican los pasos seguidos para añadir la blibioteca *newlib* a una aplicación en Zephyr.

3.1 Posibles integraciones

Zephyr ofrece dos formas de integrar estándares de C:

1. La biblioteca *minimal libc* [21], una implementación de *libc* con la funcionalidad más básica que necesita Zephyr para sus subsistemas. Esta biblioteca forma parte del núcleo de Zephyr y contiene principalmente funciones para la manipulación de cadenas de caracteres y visualización de información.
2. La biblioteca newlib [22], que no la incluye el núcleo de Zephyr sino que se encuentra precompilada en su SDK para cada una de las arquitecturas soportadas. También puede incluirse la versión *nanolib*.

Para este caso, es necesario integrar una versión completa de *newlib*.

3.2 Configuración

Se comentaba en el punto 2.4 que Zephyr cuenta con ficheros, nombrados `prj.conf`, donde se definen las opciones de configuración de la aplicación. En ellos, se pueden habilitar o deshabilitar los módulos y componentes de Zephyr que se consideren necesarios. *newlib* es uno de estos módulos, así que para poder habilitarlo hay que indicarlo en este fichero con la siguiente línea: `CONFIG_NEWLIB_LIBC=y`. De esta forma, cuando se compile la aplicación junto al kernel de Zephyr, se incluirá el móculo *newlib* permitiendo que la aplicación pueda realizar llamadas a sus funciones.

Hasta este punto del trabajo se había estado desarrollando con la plataforma de ESP32, pero durante las pruebas con *newlib* fue necesario cambiar a la NXP debido a un error en la compatibilidad de Zephyr con esta arquitectura.

3.3 Ejemplo

Aplicados estos cambios, se decidió desarrollar una pequeña aplicación de ejemplo para validar el funcionamiento de la *newlib*. Su código era muy simple, únicamente se realizaba una llamada a la función `malloc()`. Su ejecución fue exitosa, dado que se conseguía mostrar por pantalla un mensaje con la dirección de memoria reservada. Fue interesante observar como si se volvía a ejecutar el mismo código recompilando la aplicación con la opción de configuración de *newlib* deshabilitada, el error se producía en tiempo de ejecución en vez de en tiempo de compilación, algo que lleva a pensar que implícitamente Zephyr siempre va a incluir su interfaz.

Capítulo 4 - Lua en Zephyr

Tal y como se ha explicado al inicio de este documento, el sistema que ArTeCS ha desarrollado junto a Satlink en eCos ha integrado el lenguaje Lua para la implementación de los servicios en los equipos. Por tanto, es fundamental poder integrar Lua sobre Zephyr.

Incluir un lenguaje de scripting en un sistema embebido es una tarea muy habitual que resulta interesante en multitud de escenarios. Dado que Zephyr es una tecnología en pleno crecimiento, se pensó que era una buena oportunidad para aportar valor al proyecto de Zephyr Project con la integración de Lua.

En este capítulo, se explica la integración que se realizó, así como las condiciones que debía cumplir la aportación para que fuera considerada y el proceso que debía seguir para su publicación en Zephyr Project.

4.1 Importación del proyecto Lua

Lo primero de todo, es traer el código fuente de Lua al workspace de Zephyr donde se está trabajando. Este código se encuentra alojado en un repositorio en Github [23] y ha sido escrito en C.

La tarea de importación puede hacerse fácilmente desde el fichero manifest `west.yml` que, como se explicaba en el último párrafo del punto 2.4, es donde se definen los proyectos externos que serán traídos al workspace. En concreto, debe añadirse la siguiente definición a la sección "projects" de este fichero:

```
        - name: lua
          url: https://github.com/lua/lua
          revision: v5.4.4
          path: <path>
```

En ella, se indica lo mínimo indispensable para que la herramienta "west" de Zephyr sepa cómo traer el nuevo proyecto. El campo: "name" sirve como referencia al proyecto dentro del proceso de ejecución de "west"; "url" contiene la dirección web donde se localiza el repositorio del proyecto; "revision" está reservado para metadatos

del proyecto, en este caso contiene la versión de Lua, pero podría contener por ejemplo el hash del commit concreto que quiere importarse; "path" es la ruta relativa al workspace donde se alojará el código (<path> debería sustituirse por el valor que se considere oportuno).

Hecho esto, en el momento en el que se inicialice o actualice el workspace, se importará el nuevo proyecto Lua.

4.2 Integración desde CMake

Inicialmente, se optó por integrar el proyecto Lua en el proceso de compilación con CMake dentro la aplicación de Zephyr. Así, se conseguiría incluir la funcionalidad de Lua localmente a la aplicación de una forma rápida.

Para ejemplificar los pasos, vamos a suponer la siguiente estructura de aplicación de Zephyr (en este caso, al campo "path" del fichero `west.yml` explicado en el punto anterior se le asignaría el valor "app/lua"):

```
<workspace>/app
    ├── prj.conf
    ├── CMakeLists.txt
    ├── lua
    │     └── [Lua source code]
    └── src
          ├── CMakeLists.txt
          └── main.c
```

Como se puede apreciar, la aplicación se compone del código fuente en el fichero `main.c` y del código Lua, algo que habrá que reflejar en los ficheros `CMakeLists.txt`. Para compilar la aplicación, la idea es generar una librería a partir del código de Lua que posteriormente será enlazada al objeto de compilación principal, es decir, el generado a partir del `main.c`.

Para ello, el fichero `<workspace>/app/CMakeLists.txt` contendrá los siguientes comandos:

```
cmake_minimum_required(VERSION 3.20.0)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})
project(app)

add_library(liblua
      STATIC [corresponding to lua/*.c]
      )
target_include_directories(liblua
      INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}/lua
      PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/lua
      )

target_link_libraries(liblua zephyr_interface)
install(TARGETS liblua DESTINATION lib)
add_subdirectory(src)
```

Una explicación de cada uno de ellos aparece a continuación:

- `cmake_minimum_required`: indica la versión mínima necesaria de CMake para la compilación del proyecto.
- `find_package`: localiza el entorno de instalación de Zephyr.
- `project`: fija el nombre del proyecto CMake que se está compilando.
- `add_library`: crea una biblioteca, en este caso estática, a partir de los ficheros fuente de Lua. Le asigna el nombre "liblua".
- `target_include_directories`: asocia a `liblua` unos ficheros a modo de interfaz que serán visibles desde cualquier objeto que se enlace a la biblioteca. Estos ficheros son todos los que conforman el código de Lua y permitirá que el `main.c` de la aplicación pueda realizar los `#include` necesarios para trabajar con Lua.

- `target_link_libraries`: establece una dependencia con la biblioteca `zephyr_interface` para que, cuando se enlace `liblua` con un objeto, también lo haga esta biblioteca de Zephyr, la cual encapsula todas las opciones de compilación necesarias para Zephyr.
- `install`: permite mantener un orden en la compilación del proyecto.
- `add_subdirectory`: añade un nuevo directorio en la jerarquía de CMake. En este caso, informa de que dentro del directorio `src/` también existe un `CMakeLists.txt` que hay que tener en cuenta para la compilación del resto del proyecto.

En este punto, CMake ya sabe cómo generar la biblioteca de Lua. Ahora es necesario enlazarla al código principal de la aplicación desde el fichero `<workspace>/app/src/CMakeLists.txt`:

```
target_sources(app PRIVATE main.c)
target_link_libraries(app PRIVATE liblua)
```

Donde:

- `target_sources`: crea el objeto principal `app` a partir del fichero `main.c`.
- `target_link_libraries`: enlaza `app` con la biblioteca `liblua`.

Desde este momento, la aplicación ya puede hacer uso de las llamadas Lua.

4.3 Error y solución en *newlib*

Dado que el código de Lua realizaba llamadas a *newlib*, fue necesario integrar también esta biblioteca siguiendo los pasos del Capítulo 4. Inesperadamente, apareció un error durante la compilación. Resulta que el compilador no conseguía encontrar la definición de las funciones `_times()`, `_unlink()` y `_link()`. Investigando acerca de su origen, se encontró que correspondían a tres stubs que empleaba *newlib*, es decir, tres funciones que deberían ser implementadas por el propio módulo *newlib* ofrecido por Zephyr.

Este error era un problema importante de funcionalidad en Zephyr, ya que se trata de falta de implementación en un módulo ya integrado. Las soluciones ante este problema eran claras, o informar del error a la comunidad de Zephyr para que se arreglara, o aportar desde la aplicación una implementación a cada uno de los stubs.

Con el fin de evitar depender de terceros y encontrar una solución rápida, se optó por la segunda opción (aunque también se dejó constancia del problema en la comunidad de Zephyr [24]). Para ello, se investigó sobre las funciones que estos stubs debían realizar, concluyendo que: `_times()` se encargaba de obtener una medición del tiempo de ejecución y `_unlink()` y `_link()` se llamaban durante el borrado y la creación de ficheros, respectivamente. De los tres stubs, solo `_unlink()` pudo implementarse con una funcionalidad real gracias a que Zephyr cuenta con una llamada muy similar, `fs_unlink()`, en su componente encargado de los sistema de ficheros. Para el resto de stubs se aportó una implementación vacía en forma de "operación no soportada".

Para llevar a cabo esta implementación, se creó un nuevo fichero, `missing_stubs.c`, con el fin de aislar en él los tres stubs. Se situó en `<workspace>/app/src` y su contenido era el siguiente:

```
#include <zephyr.h>
#include <stdio.h>

clock_t _times(struct tms* tms) {
     errno = ENOSYS;
     return (clock_t) -1;
}

int _unlink(const char *pathname) {
     return fs_unlink(pathname);
}

int _link(const char *oldpath, const char *newpath) {
     errno = ENOSYS;
     return -1;
}
```

Por último, `missing_stubs.c` debía incluirse en el fichero `CMakeLists.txt` de `<workspace>/app/src/`, por lo que se cambió únicamente el comando `target_sources(app PRIVATE main.c)` por:

```
target_sources(app PRIVATE main.c missing_stubs.c).
```

4.4 Módulo *liblua*

Después de conseguir integrar Lua localmente a la aplicación, se pensó que se trataba de una solución que debía solventarse de una manera más formal, por lo que se planteó la idea de crear un módulo *liblua* para una integración nativa en Zephyr.

Recordando la definición: un módulo en Zephyr consiste en un código fuente proveniente de un proyecto externo el cual es añadido a Zephyr con el objetivo de integrar una nueva funcionalidad. Un ejemplo de módulo es OpenAMP, el cual ha permitido a Zephyr trabajar con sistemas embebidos heterogéneos.

Los pasos para crear el nuevo módulo de Lua se explican a continuación:

De nuevo, se va a suponer que se trabaja sobre el siguiente workspace de Zephyr:

```
zephyrproject/
      ├── .west/
      │     └── config
      ├── zephyr/
      │     ├── [... other files ...]
      │     └── west.yml
      ├── modules/
      │     ├── [... other projects ...]
      │     └── lib/
      │           └── [... other projects ...]
      ├── tools/
      │     └── [ ... other projects ...]
      └── bootloader/
            └── [ ... other files ...]
```

1. Importar el proyecto de Lua como se describió en el punto 4.1, aunque en este caso al campo "path" del fichero manifest se le asigna el valor "modules/lib/liblua".

2. Tras actualizar el workspace con el comando "west update", se habrá importado correctamente el proyecto Lua. Ahora hay que establecer la estructura de ficheros necesaria para formar un módulo.

```
zephyrproject/
└── modules/
    └── lib/
        └── liblua/
            ├── [Lua source code]
            ├── README
            ├── CMakeLists.txt
            ├── Kconfig
            └── zephyr
                └── module.yml
```

Estos ficheros son:

- `README`: en lenguaje natural, explica el origen, la utilidad y cómo funciona el módulo, es decir, aporta una visión general.
- `Kconfig`: define las opciones de configuración del módulo para CMake. En este caso, se ha decidido crear una opción que deberá ser activada por el usuario para integrar Lua en su aplicación. El contenido del fichero queda como sigue:

```
config MODULE_LIBLUA
    bool "Lua support"
    depends on FILE_SYSTEM && NEWLIB_LIBC
    help
      Use the Lua implementation.
```

Se puede ver que se ha definido de tal forma que dependa del componente `FILE_SYSTEM` y `NEWLIB_LIBC`, ya que el primero de ellos incluye la llamada para la implementación del stub `_unlink()` y el segundo es utilizado por el código fuente de Lua. Estas dependencias

implican que el usuario deberá también activar esas opciones de configuración o el compilador devolverá un error.

- `CMakeLists.txt`: determina cómo debe compilarse el módulo en CMake. Su contenido cambia ligeramente respecto al que aparece en el punto 4.2, ya que se utilizan comandos CMake definidos por Zephyr:

```
if(CONFIG_MODULE_LIBLUA)

add_library(liblua INTERFACE)
zephyr_include_directories(
     ${CMAKE_CURRENT_SOURCE_DIR}
     )
zephyr_library()
zephyr_library_sources_ifdef(
     CONFIG_MODULE_LIBLUA
     [corresponding to *.c]
     )

zephyr_library_sources_ifndef(
     CONFIG_BOARD_NATIVE_POSIX
     missing_stubs.c
     )

endif()
```

Destacar que los comandos se encuentran bajo una estructura condicional, en la que se comprueba si está habilitada la opción de configuración definida en el `Kconfig`. De esta forma, se compilará Lua solamente si el usuario lo requiere. Hay que mencionar también que el fichero `missing_stubs.c` es compilado solo si la plataforma destino no es POSIX, ya que en caso contrario incluirá una implementación válida de los stubs que faltan.

- `zephyr/module.yml`: indica dónde se encuentran el `CMakeLists.txt` y el `Kconfig` del módulo. En este caso, son las rutas por defecto.

```
build:
  cmake: .
  kconfig: Kconfig
```

3. Hecho esto, el usuario puede hacer uso de Lua en su aplicación habilitando los módulos necesarios en su fichero `prj.conf` (además debe

```
# liblua
CONFIG_MODULE_LIBLUA=y
# liblua dependencies
CONFIG_FILE_SYSTEM=y
CONFIG_NEWLIB_LIBC=y
CONFIG_MAIN_STACK_SIZE=4096
```

aumentarse el tamaño de la pila para Lua):

4.5 Requisitos de un módulo para ser candidato

Con el fin de conseguir una mejor organización en el flujo de trabajo de los módulos que forman parte de Zephyr, existen una serie de políticas y buenas prácticas que deberían llevarse a cabo. En la siguiente lista se resumen varias de ellas [25]:

- El proyecto externo asociado al módulo debe tener su propio ciclo de vida fuera de Zephyr, es decir, residir en su propio repositorio y tener su propio "contribution and maintenance workflow".
- El módulo no debe contener código que haya sido escrito exclusivamente para Zephyr.
- El módulo debe ofrecer una funcionalidad que haya sido avalada y aprobada por el TSC ("Technical Steering Committee") de Zephyr.
- El módulo debe cumplir con:
    - Los requisitos de licencias:
        - Todos los ficheros fuente del módulo deben incluir una cabecera donde se indique la licencia bajo la que se encuentran, a menos que el propio repositorio tenga un fichero con su licencia que englobe a todos estos ficheros.
        - La licencia debe ser documentada en el fichero `README` o en el manifest `.yml` del módulo.
        - Si el proyecto externo tiene una licencia distinta a la Apache 2.0 pero es OSI ("Open Source Initiative", es decir, abierta), deberá ser aprobada por Zephyr.
    - Los requisitos de usuarios, contando con los siguientes roles:
        - El administrador, que es responsable de gestionar el acceso al repositorio del módulo.
        - El propietario, que se encarga de gestionar el módulo a más alto nivel que el administrador.

        - El merger, cuya función es la de publicar resultados y actualizaciones en el repositorio de Zephyr.
    - Los requisitos de la documentación, indicando en un fichero:
        - El enfoque y el objetivo del módulo.
        - La manera de integrar el módulo en Zephyr.
        - El propietario del módulo.
        - La información correspondiente a la sincronización con el repositorio externo, como la referencia al commit, el SHA, la versión, etc.
        - La información correspondiente a la licencia como se ha explicado anteriormente.
    - Los requisitos de validación:
        - Debe incluirse en el árbol principal de Zephyr un conjunto sencillo de ejemplos y pruebas de test.
        - Debe verificarse el funcionamiento básico del módulo, como configuración, API, etc.
    - Los requisitos del mérito:
        - El TSC debe valorar si es el módulo candidato lleva a cabo su funcionalidad de manera óptima.
        - El TSC debe confirmar que el mantenimiento del proyecto externo sigue en actividad.
        - El TSC debe plantear si en el pasado ya se han considerado alternativas para la solución que integra el módulo, así como si existen otros proyectos externos que implementen la misma funcionalidad.
- El módulo debería tener un desarrollador comprometido con su mantenimiento.

- A través de un pull-request, el repositorio externo del módulo debe ser incluido y/o actualizado en el repositorio de Zephyr Project.

En el módulo *liblua* creado en el punto 4.4, no se han formalizado todos estos aspectos, ya que como se verá en el siguiente punto, antes de enviar el módulo candidato debe realizarse una propuesta a los miembros de Zephyr.

4.6 Proceso de publicación

Publicar una solución en un proyecto de código abierto como es Zephyr Project implica seguir un proceso para verificar su viabilidad. En el esquema de la Figura 4-1 se ilustra paso a paso el proceso que debe seguirse.

En concreto, el flujo teórico para el módulo *liblua* quedaría de la siguiente forma:

1. Lua cuenta con licencia MIT, por lo tanto sí que es englobada en OSI.
2. Se publica una issue con la propuesta del módulo *liblua* en el repositorio de Zephyr Project (redactada en el Apéndice B) [26].
3. El TSC estudia y aprueba/rechaza la propuesta.
4. La licencia MIT es distinta que la licencia Apache 2.0. Sin embargo, su política es más abierta que la mínima exigida, por lo que debería ser aprobada por la junta de Zephyr.

Finalizado este proceso, se realizaría un pull-request a Zephyr Project con el módulo *liblua*, siendo integrado oficialmente en el proyecto.

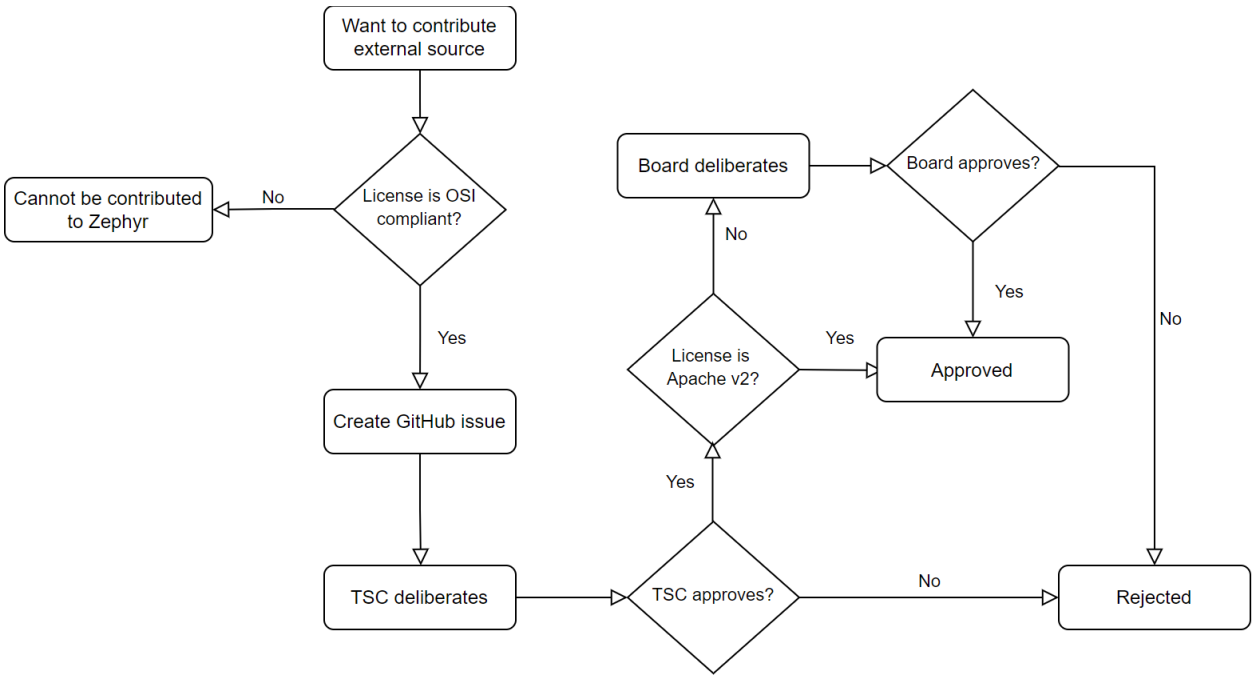


Figura 4-1: Proceso de publicación de un módulo en Zephyr Project [27]

Capítulo 5 - Iridium en sistemas embebidos

Las características de un sistema IoT vienen definidas por los requerimientos del caso de uso que se quiera solventar. A menudo, esos requisitos se centran en ofrecer comunicaciones desde lugares remotos que ni siquiera las comunicaciones celulares alcanzan. Por esta razón, surge la necesidad de emplear comunicaciones satelitales, pues aportan una visibilidad en las comunicaciones casi permanente.

En este capítulo, se abordan distintos temas relacionados con la implantación de las comunicaciones por satélite en un sistema embebido.

5.1 Integración de comunicaciones satelitales

Cualquier sistema embebido diseñado para realizar comunicaciones satelitales necesita un módem. Hasta cierto punto, estos módems permiten aislar al sistema de los aspectos relacionados con este tipo de comunicaciones. Los módems ofrecen una interfaz por comandos que permite interactuar y sacar el máximo provecho de su funcionalidad. Así, el sistema embebido únicamente se dedica a manejar la interfaz predefinida por el fabricante para llevar a cabo sus tareas de comunicación.

Un ejemplo de módem de Iridium puede verse en la Figura 5-1, se trata del módulo 9603. Este dispositivo está pensado para ser integrado en sistemas embebidos debido al pequeño tamaño, ligereza y bajo consumo que supone. En realidad, este ejemplo no se trata de un módem sino de un módulo Iridium. La diferencia es que un módem corresponde únicamente al hardware que trabaja junto a la antena y el módulo integra tanto el módem como el resto del hardware necesario para que sea funcional (control lógico basado en un microprocesador, E/S por pines digitales y analógicos, suministro de energía y la propia antena con el módem). Además, la mayoría de los dispositivos de Iridium hacen uso del SBD ("Short Burst Data"), un servicio de datos de Iridium que optimiza el envío y la recepción de pequeños flujos de datos (menores a 2 KB) en intervalos periódicos de tiempo. Este servicio es ideal para aplicaciones de monitorización remota.



Figura 5-1: Iridium 9603 Module en comparación de tamaño con un lapicero [28]

Habitualmente, tanto los módems como los módulos se conectan al sistema embebido a través de UART o de I2C, por donde pueden acceder a la interfaz de comandos expuesta por el módem.

5.2 Interfaz de comandos en un módem satelital

Como se ha comentado en el punto anterior, los fabricantes ofrecen una interfaz de comandos por medio de la cual los sistemas en los que se integran pueden enviar las órdenes que consideren oportunas.

Actualmente, todos los módems utilizan los comandos conocidos como AT. Esta familia de comandos surge de los antiguos comandos Hayes, que eran utilizados con el mismo propósito y siempre comenzaban con las siglas AT para solicitar "ATtention" por parte del módem. De ahí, evolucionaron a los comandos AT, lo cuales incorporan todos los comandos Hayes entre otros.

Existen cuatro tipos de comandos AT:

- *Test*: comprueban si un comando es soportado por el módem.

  Sintaxis – `AT<comando>=?`

- *Read*: obtienen los ajustes del módem para una operación.

  Sintaxis – `AT<comando>?`

- *Set*: modifican los ajustes del módem para una operación.

  Sintaxis – `AT<comando>=<valor0>…<valorN>`

- *Execution*: ejecutan una operación.

  Sintaxis – `AT<comando>=<param0>…<paramN>`

Algunos de los comandos soportados por los módulos Iridium son [29]:

- Para obtener información del módem:
  - `+GMI` consulta el distribuidor.
  - `+GMM` consulta el modelo.
  - `+GMR` consulta la versión.
  - `+GSN` consulta el número de serie o IMEI.
  - `+CSQ` consulta la calidad de la señal o RSSI.
  - `+CULK` lo desbloquea después de que haya sido bloqueado por un Gateway o consulta el estado de bloqueo en el que se encuentra.
- Para utilizar el servicio SBD:
  - `+SBDI` inicia una sesión SBD.
  - `+SBDIX` inicia una sesión SBD en su versión extendida, la cual incluye la función de "Automatic Notification" que notifica al sistema embebido ante la llegada de eventos registrados.
  - `+SBDREG` inicia una sesión SBD pero realiza un registro manual de los parámetros de la red Iridium.

- `+SBDWB` envía un mensaje binario vía SBD.
- `+SBDRB` recibe un mensaje binario vía SBD.
- `+SBDWT` envía un mensaje de texto vía SBD.
- `+SBDRT` recibe un mensaje de texto vía SBD.
- `+SBDD` vacía los buffers de mensajes SBD.

Con esta información, ya pueden generarse una gran variedad de comandos. Por ejemplo, para enviar un mensaje de texto a través del servicio SBD, se puede ejecutar el comando `AT+SBDWT=Hello World` para enviarlo al buffer del módem y, a continuación, el comando `AT+SBDIX` para llevar a cabo el envío.

5.3 Iridium en los dispositivos de Satlink

Satlink, tal y como se ha explicado, desarrolla dispositivos IoT enfocados a la industria pesquera. Sin embargo, no solo interviene en esta industria, sino que también ofrece soluciones para la industria aérea y terrestre. En todas ellas, busca proporcionar equipos que sean capaces de mantener una comunicación permanente independientemente del punto del planeta en el que se encuentren.

Por esta razón, Iridium es una de las tecnologías que emplean por la extraordinaria conectividad que ofrecen. En todos estos casos, el usuario se va a encontrar en escenarios inalcanzables por las comunicaciones celulares: los dispositivos para la industria pesquera se van a localizar en cualquier punto del océano, los dispositivos para la industria aérea van a estar a una altitud de varios kilómetros respecto al nivel del mar y los dispositivos para la industria terrestre pueden estar en mitad de la selva. Todo ello hace que las comunicaciones satelitales, en concreto las ofrecidas por Iridium, sean imprescindibles para el desarrollo de las soluciones de Satlink.

5.4 Recursos para el desarrollo con Iridium

Para desarrollar los dispositivos de Satlink, se propuso trabajar con el módulo RockBLOCK Mk2 (véase en la Figura 5-2). Este módulo se trata de un dispositivo Plug&Play, por lo que cuenta con todo lo necesario para funcionar una vez es instalado en el sistema. Se basa en el módem Iridium 9602 y tiene una antena RF built-in para establecer las comunicaciones. También expone una interfaz física de conexiones para datos y alimentación, siendo compatible con casi cualquier dispositivo gracias a su interfaz serie. La interacción entre el dispositivo y el módulo se realiza por medio de comandos AT, como se explicaba en el punto anterior. Debido a que en las comunicaciones satelitales se llevan a cabo intercambios de pequeñas cantidades de datos (en el caso de Iridium, 340 B para el envío y 270 B para la recepción) de manera periódica, el módulo RockBLOCK ofrece un modo de bajo consumo para ahorrar energía mientras no se produzca ninguna de estas comunicaciones.



Figura 5-2: Módulo RockBLOCK Mk2 de Iridium [30]

Existen tres modelos diferentes de este módulo que incorporan características propias, tal y como se resumen en la Tabla 6-1.

	Naked Std	Naked w/SMA	RB Plus
Antena integrada	✓	✗	✓
Puerto SMA para antena externa	✗	✓	✗
Requisitos de alimentación	5 V	5 V	9 – 30 V
Carcasa impermeable	✗	✗	✓
Modo de bajo consumo	✓	✓	✓
Puerto de notificación de mensajes	✓	✓	✓
Interfaz UART	✓	✓	✗
Interfaz RS232	✗	✗	✓
Estructura de montaje	✗	✗	✓

Tabla 5-1: Características de los modelos del módulo RockBLOCK Mk2 de Iridium

En el momento en el que se iba a comenzar a trabajar con el kit de desarrollo 9603 (mostrado en la Figura 5-3), surgió un problema inesperado: el módulo no funcionaba. Analizando el hardware, se dio con la causa y resulta que su conectore con la PCB se había desoldado (en la Figura 5-4 puede verse en detalle).

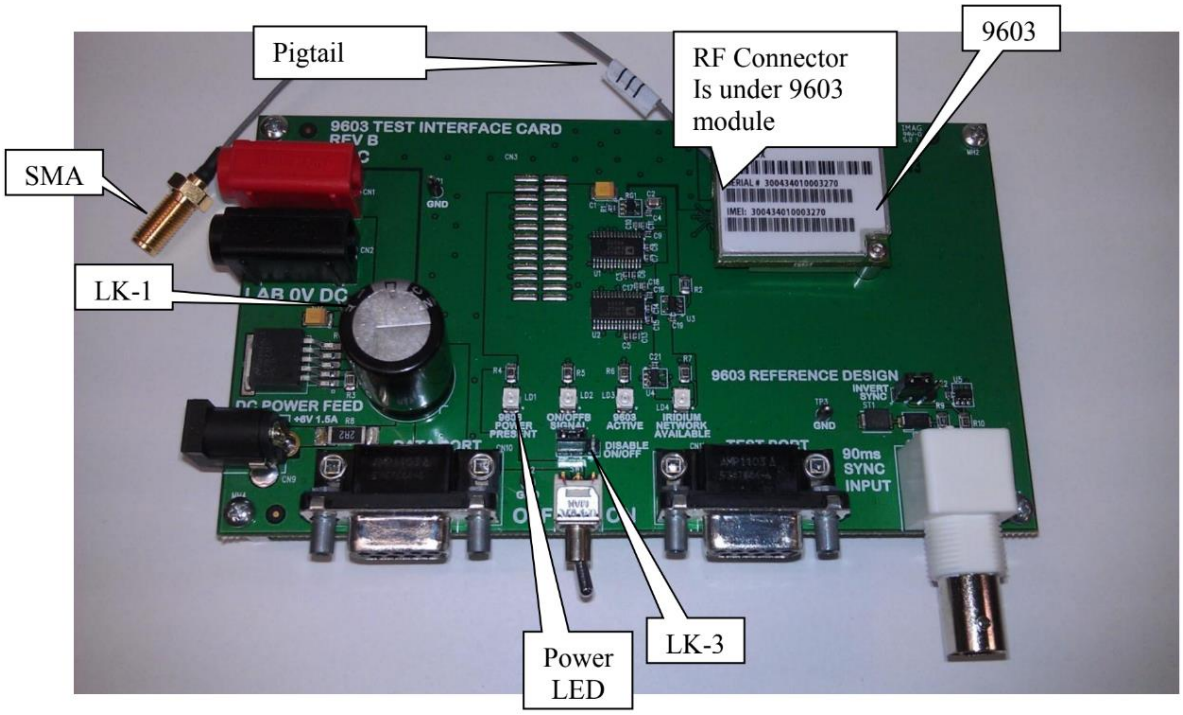


Figura 5-3: Kit de desarrollo 9603 de Iridium [31]

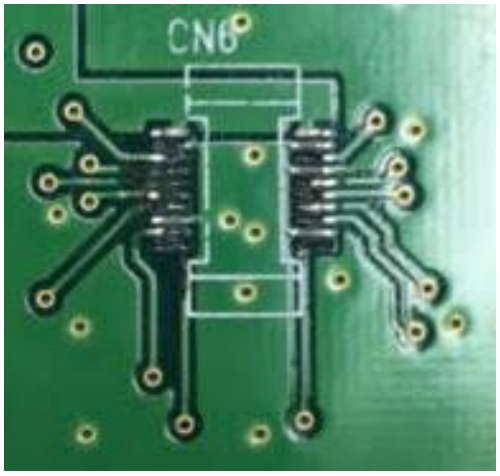


Figura 5-4: Pines desoldados en el kit de desarrollo 9603

Ante esta situación, se comunicó a Satlink el problema y optaron por realizar el pedido de un nuevo módulo. Sin embargo, el pedido no llegó a tiempo para el trabajo, por lo que se propuso una alternativa más inmediata: utilizar un emulador.

A partir de GitHub, se dio con la opción que se estaba buscando, un emulador del módem Iridium 9602/9603 [20], estos son, los módems que se iban a emplear para el desarrollo. El código del emulador se compone de varios scripts Python, los cuales van a

ser lanzados sobre puertos serie virtuales (con la herramienta `socat` de Linux) en el host, aunque el sistema embebido deberá conectarse a través de un cable USB-TTL para poder interactuar con el emulador. Para poder lanzarlo a ejecución, fue necesario aplicar algunas modificaciones sobre el código original [32].

Concretamente, la arquitectura en las comunicaciones con Iridium, que aparece ilustrada en la Figura 5-5, consta de tres partes diferenciadas:

1. El end-device, que está formado por el payload, el software que lo procesa y el modem Iridium junto a sus controladores.
2. La constelación de satélites Iridium, que cuenta con comunicaciones inter-satelitales y que redirecciona el payload hacia su destino.
3. La estación en tierra, que actúa como Gateway y que se encarga de administrar el payload.

El payload en cuestión puede ser transmitido de forma indirecta (a través de e-mail) o de forma directa (a través de UDP indicando la dirección IP del destino).

Con este escenario, se puede comprender el objetivo del emulador: eliminar la necesidad de disponer de un módem Iridium físico y de tener que acceder a la constelación de satélites. En la figura 5-6, puede verse como se simplifica esta arquitectura con el emulador. Ahora, solamente se necesita el software para procesar el payload y la estación destino para que actúe como extremo de la comunicación.

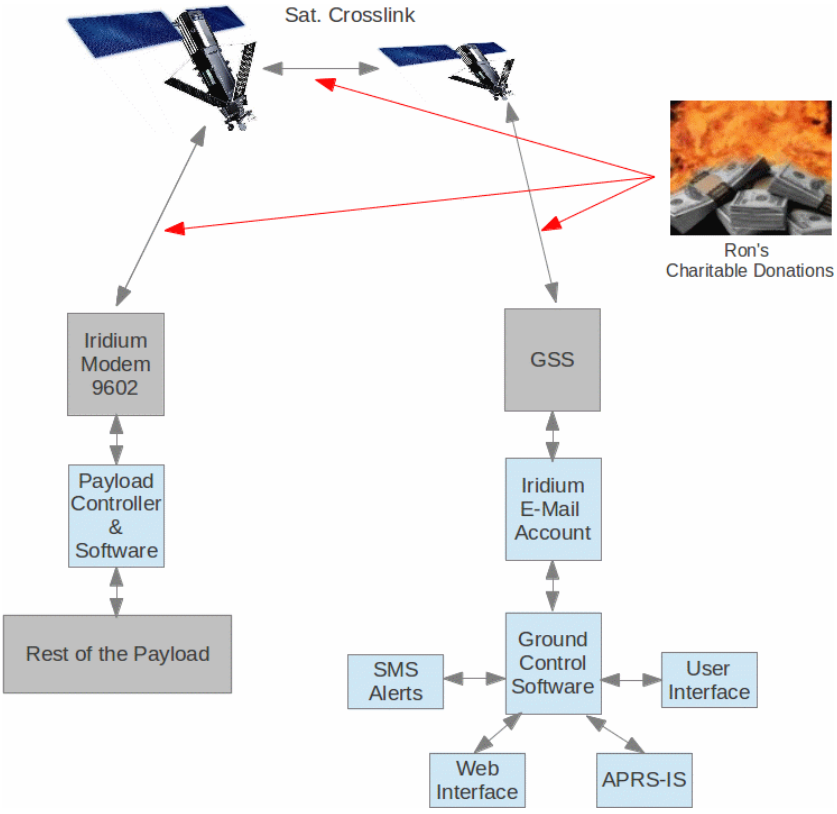


Figura 5-5: Arquitectura de las comunicaciones reales con Iridium [33]

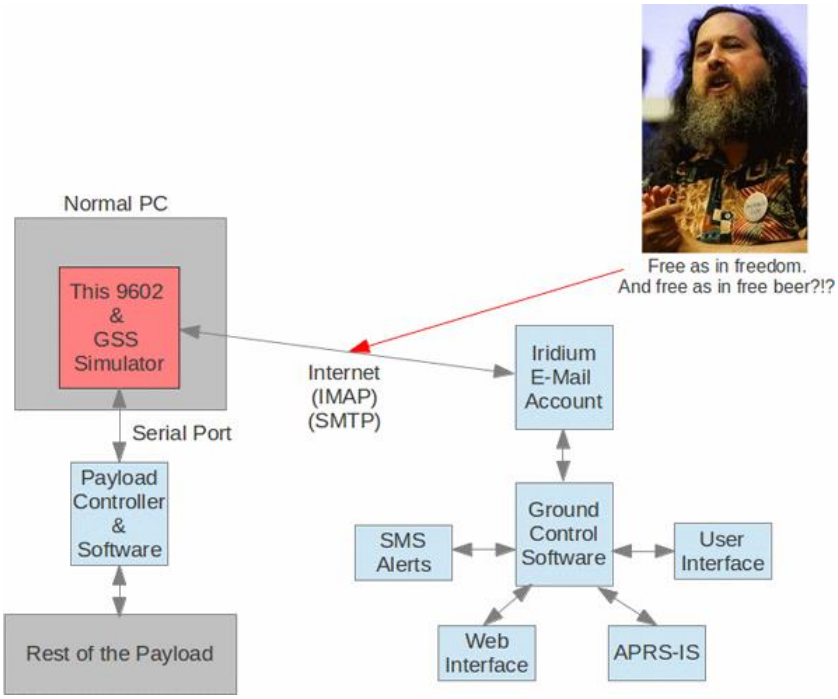


Figura 5-6: Arquitectura de las comunicaciones con Iridium a través de un emulador [33]

Capítulo 6 - Resultados

Este capítulo recoge los resultados que se obtuvieron del trabajo. Primero, la validación de la integración del módulo *liblua*, después, el estado actual en el que se encuentra la propuesta presentada a Zephyr Project, y por último, las pruebas realizadas con el emulador Iridium 9603.

6.1 Funcionamiento del módulo *liblua*

Para validar el funcionamiento del módulo *liblua*, se ha implementado una pequeña aplicación en Zephyr que ha sido ejecutada sobre una MV x86 en QEMU.

El código de la aplicación realiza llamadas a distintas funciones de Lua:

```
#include <zephyr.h>
#include <fs/fs.h>

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

void main(void) {
      lua_State *L;
      printk("Starting Lua code\n");

      L = luaL_newstate();
      luaL_openlibs(L);
      luaL_dostring(L, "print('Hello from Lua')");
      /* Expected output: Hello from Lua */

      lua_newtable(L); /* Push table */
      int ref = luaL_ref(L, LUA_REGISTRYINDEX);
```

```
        lua_rawgeti(L, LUA_REGISTRYINDEX, ref);
        lua_pushstring(L, "hello"); /* Push key */
        lua_pushstring(L, "hi"); /* Push value */
        lua_settable(L, -3);
        lua_pushstring(L, "bye");
        lua_pushinteger(L, 45);
        lua_settable(L, -3);
        lua_pop(L, 1); /* Pop table */

        lua_rawgeti(L, LUA_REGISTRYINDEX, ref);
        lua_pushstring(L, "bye");
        lua_gettable(L, -2);
        printk("table[\"bye\"] = %d\n",(int)lua_tointeger(L, -1));
        /* Expected output: table[“bye”] = 45 */
        lua_pop(L, 1); /* Pop table */

        luaL_dostring(L, "a = {\"15\", \"38\"}");
        luaL_dostring(L, "print(a[1])");
        /* Expected output: 15 */

        lua_close(L);
        printk("Test Done!\n");
    }
```

Generando la salida esperada en QEMU:

```
        Starting Lua code
        Hello from Lua
        table[“bye”] = 45
        15
        Test Done!
```

6.2 Estado actual de la propuesta a Zephyr Project

En el punto 4.6, se ha descrito el flujo teórico que debería de seguir la propuesta del módulo *liblua*. Durante el desarrollo de este trabajo, se llegó a publicar la issue describiendo cuidadosamente el módulo pero, a día de hoy, no se ha obtenido todavía respuesta del TSC con su aprobación.

6.3 Pruebas con el emulador Iridium 9602/9603

Las comunicaciones satelitales han sido planteadas a través de un emulador del módem Iridium 9602/9603, como se comentaba en el punto 6.4. Para poder ejecutar este emulador en el host, se ha utilizado la herramienta `socat` de Linux. Gracias a ella, se ha conseguido interactuar con el emulador a través de una shell local siguiendo los siguientes pasos:

1. Crear los puertos series virtuales (shell A).

   ```
   sudo socat -d -d pty,link=$HOME/ttyS0,raw,echo=0
   pty,link=$HOME/ttyS1,raw,echo=0
   ```

2. Habilitar los permisos sobre los puertos virtuales.

   ```
   sudo chmod o+rw $HOME/ttyS*
   ```

3. Acceder a un extremo del puerto (shell B).

   ```
   minicom -D $HOME/ttyS1 19200
   ```

4. Lanzar el emulador (shell C).

   ```
   python2 Iridium9602.py -d $HOME/ttyS0 -u <email> -p
   <pass> -i imap.gmail.com -o smtp.gmail.com -r
   <dest_email> -m EMAIL
   ```

   En este caso, se está indicando al emulador que lance el módem en modo `EMAIL`, lo que significa que será capaz de mandar mensajes por correo electrónico. Para ello, en el campo `<email>` y `<pass>` se indican las credenciales de la cuenta de correo electrónico que va a enviar el mensaje, y en el campo `<dest_email>` la dirección de correo electrónico que lo va a recibir. También se especifican los servidores IMAP y SMTP para realizar el envío del mensaje.

Hecho esto, se puede interactuar con el emulador enviando los comandos AT a través de la shell B. Por ejemplo:

```
> at+gmi
< +Iridium


< OK
```

```
> at+gmm
< IRIDIUM 9600 Family SBD Transceiver


< OK
```

```
> at+gsn
< 300234060604220


< OK
```

Estos comandos, generan en la shell C la siguiente salida:

```
Here is what I received:at+gmi
Sending OK
Here is what I received:at+gmm
Sending OK
Here is what I received:at+gsn
Sending OK
```

Como se puede ver, en la shell B se está recibiendo la respuesta que se obtendría de un módem real.

En el siguiente ejemplo, se muestra un caso un poco más complejo del envío de un mensaje a través del correo electrónico:

```
> at+sbdwt=hello world
< OK
> at+sbdix
< +SBDIX: 0, 1, 0, 0, 0, 25

< OK
```

Como resultado, en la cuenta de correo electrónico destino se recibe lo siguiente:

```
From: <email>
To: <dest_email>
Subject: SBD Msg From Unit: 300234060379270
Content:
      MOMSN: 0
      MTMSN: 0
      Time of Session (UTC): Mon Sep  5 17:23:53 2022
      Session Status: TRANSFER OK
      Message Size: 5

      Unit Location: Lat = 0.000000 Long = 0.000000
      CEPRadius = 3

      Message is Attached.
```

Este correo viene con un fichero adjunto (`text.sbd`) que almacena el contenido del mensaje, en este caso, "hello world".

Capítulo 7 - Conclusiones y trabajo futuro

En los últimos años, el Internet de las Cosas se ha asentado en el campo de la informática. Esto ha fomentado la aparición de nuevos proyectos enfocados a esta tecnología que han crecido gracias a la implicación de la comunidad. Zephyr es un claro ejemplo.

Inicialmente, se planteó este trabajo con el único objetivo de portar el sistema desarrollado por ArTeCS de eCos a Zephyr. Conforme se avanzaba, el alcance del trabajo se reajustó hacia un objetivo menos ambicioso: proponer una solución estable para la integración de Lua. Seguramente, las dificultades que surgieron podrían haberse esquivado si Zephyr se encontrase en un estado de desarrollo más avanzado.

Me gustaría destacar el resultado conseguido, ya que ha sido fruto de un proceso de desarrollo muy flexible y que puede llegar a aportar valor real a la comunidad. Por un lado, la propuesta del módulo *liblua* todavía se encuentra en la primera fase para ser aceptado, pero contribuir de manera activa en un proyecto de código abierto representa una motivación adicional del trabajo. Por otro lado, alcanzar una integración robusta de Lua en Zephyr y retomar el proyecto del emulador Iridium para trabajos posteriores podrá formar parte del desarrollo de muchos proyectos futuros.

La evolución futura de este trabajo se divide en dos vertientes. Por un lado, continuar con la propuesta del módulo *liblua*, ya que se encuentra en fases tempranas. Este proceso puede ser relativamente largo al depender de la aprobación de terceros. Por otro lado, desde el punto de vista de Satlink, continuar con la portación hacia Zephyr, para la cual este trabajo habrá sido de utilidad. Como se ha comentado, es un objetivo ambicioso por la complejidad del sistema desarrollado por ArTeCS, pero con el paso del tiempo, Zephyr irá ganando la consistencia necesaria para poder soportar el sistema en su totalidad.

Personalmente, termino este trabajo con pensamientos enfrentados. El alcance en tiempo y recursos no ha permitido lograr el estado de desarrollo que se esperaba en un principio. Sin embargo, creo que me ha aportado a nivel de experiencia y familiaridad el haber trabajado: primero, con un proyecto de RTOS, porque me ha

permitido entender cuál es su organización, qué aspectos son importantes, cómo funcionan sus bases, etc.; segundo, con una herramienta tan utilizada como CMake, que hasta el momento no había trabajado con ella en un proyecto complejo; y tercero, con hardware para comunicaciones satelitales, pues me ha resultado realmente interesante aprender un poco acerca de esta industria. Considero que estos tres campos de conocimiento van a ser útiles en mi trayectoria, puesto que aportan muy positivamente en el trabajo con sistemas embebidos para IoT. Además, acercarme a entender e intentar participar en un proyecto de código abierto pienso que es una tarea menos habitual de lo que debería y haber conocido el proceso y los requisitos para ello también me resulta muy atractivo.

Chapter - Introduction

Motivation

Satlink [1] is a Spanish technology company focused on the development of Internet of Things (IoT) solutions. As a provider of technological solutions for the fishing segment, Satlink helps to optimize decision-making by providing access to relevant data for a more sustainable and efficient management of ocean resources.

Since the end of 2013, the research group ArTeCS ("Architecture and Technology of Computing Systems") belonging to the Department of Computer Architecture and Automatics of the UCM (Complutense University of Madrid) has been working on a "HW/SW platform for the development of services of a satellite device on an imperative programming language" [2] together with Satlink. Figure 1-1 shows the architecture designed by ArTeCS for this platform, which distinguishes: the hardware part, composed by the microprocessor and the components dedicated to communications, and the software part, which is implemented in several layers. The upper layers ("Common Services", "User Services" and "Service Framework + Extensions") are in charge of implementing the application logic and, therefore, it is where the behaviour of the equipment that Satlink is going to develop is determined. This logic is defined through services, that is, code fragments that execute a very specific functionality. The services are written in Lua, a scripting language focused on embedded systems, so the next layer ("Lua Compiler & Interpreter") corresponds to the interpreter and compiler that will allow its execution. Finally, the lower layer ("Firmware + Operating System") houses the operating system and the firmware necessary to interact with the platform's hardware and is the object of study of this project.

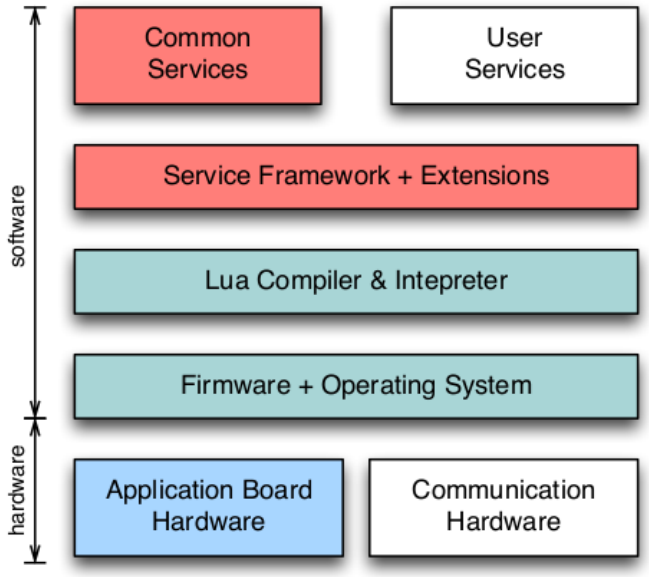


Figure 1-1: System architecture designed by ArTeCS for Satlink

Objectives

ArTeCS and Satlink's idea was to develop the system on top of an RTOS, so they decided to start with eCos, an RTOS that can offer many advantages but currently maintains a proprietary licence and a low level of adaptability to the IoT technologies that have emerged in recent years. For this reason, and with a view to improving the robustness and flexibility of the system, they decided to switch to a newer technology. Thus, the objective proposed for this project was to start porting the system from eCos to Zephyr, a modern and free RTOS focused on the development of IoT devices.

Specifically, the following objectives were proposed:

1. To reach a high level of understanding of Zephyr to be able to manage it from its basis.
2. To make a functional integration of Lua on Zephyr.
3. To propose a stable solution, in the form of a module, to Zephyr Project so as to be included in the project.

Work plan

To achieve these objectives, it is first necessary to understand the implementation of Satlink's current system and to know its points of interest in order to start porting it in the best possible way to the new technology. This phase of understanding the system that Satlink is integrating on its products was carried out during the internship, a period during which an approach was made from the highest level of abstraction to understand the Lua services they develop and the final solution they offer.

With an already established idea of the system as a whole, the project began and its first part consisted of finishing learning the Lua language. Then, the first objective of learning how to use Zephyr was started. This RTOS is what the whole system was going to be based on and it was necessary to know its strengths and weaknesses to exploit them later. The difficulty of this objective is increased because it is not only necessary to know how to use the RTOS, but also to know aspects that usually are in the background, such as its compilation process, its resource hierarchy, its components that are available for the applications, etc.

Once the tools to be used were sufficiently known, it was necessary to start porting the system. To do this, the essential thing was to integrate Lua on Zephyr, because from that moment, the services implemented by Satlink could be ported to the new system. We started with a simple integration until we reached the last objective: to formalise the solution. From this point, the requirements that a module had to meet to be considered by Zephyr were studied and the process of making the proposal began.

Document structure

This document is divided into different chapters:

- Chapter 1 summarises the initial ideas of the origin of the project, such as the motivation, the stated objectives, the work plan to be followed and the structure in which this document is organised.
- Chapter 2 describes the main tools that have been used during the project and which are necessary to know well in order to understand the tasks that have been made. In particular, it discusses: Lua, the scripting language integrated by the Satlink teams; CMake, the tool to which the compilation process is delegated in complex projects; *libc* and *newlib*, the standard C libraries that are essential to use in most embedded systems; Zephyr, the RTOS to which the Satlink and ArTeCS system is to be ported; the platforms that have been worked with, an ESP32 and an NXP; Iridium, the satellite service working with Satlink; and the versions of the tools used in the work.
- Chapter 3 explains the different alternatives offered by Zephyr for the integration of C standards in an application, as well as providing an example of use.
- Chapter 4 discusses the process of integrating Lua on Zephyr how to contribute to an open-source project, such as the Zephyr Project, with this solution.
- Chapter 5 details the process of incorporating and developing satellite communications into an embedded system. It will review the solutions offered by Iridium and how to work with them.
- Chapter 6 discusses the results and the current status of the proposed contribution.
- Chapter 7 draws a series of conclusions, as well as the evolution with which the project should continue.

Chapter - Conclusions and future work

In recent years, the Internet of Things has established itself in the field of computing. This has encouraged the emergence of new projects focused on this technology that have grown thanks to the involvement of the community. Zephyr is a clear example.

Initially, this project was approached with the sole objective of porting the system developed by ArTeCS from eCos to Zephyr. As the project progressed, the scope of the work was readjusted towards a less ambitious objective: to propose a stable solution for Lua integration. Certainly, the difficulties that arose could have been avoided if Zephyr was at a more advanced stage of development.

I would like to note the result achieved, as it has been the result of a very flexible development process that can bring real value to the community. On the one hand, the *liblua* module proposal is still in the first phase to be accepted, but actively contributing to an open-source project represents an additional motivation for the work. On the other hand, achieving a robust integration of Lua in Zephyr and taking up the Iridium emulator project for further work can be part of the development of many future projects.

The future evolution of this project is twofold. On the one hand, to continue with the *liblua* module proposal, as it is in early stages. This process may be relatively lengthy as it depends on the approval of third parties. On the other hand, from Satlink's point of view, to continue with the porting to Zephyr, for which this work will have been useful. As mentioned, this is an ambitious goal due to the complexity of the system developed by ArTeCS, but over time, Zephyr will gain the necessary consistency to be able to support the system as a whole.

Personally, I end this project with mixed thoughts. The scope in time and resources has not allowed me to reach the state of development that was originally expected. However, I believe that it has given me experience and familiarity to have worked: first, with an RTOS project, because it has allowed me to understand its organization, what aspects are important, how its bases work, etc.; second, with a tool as widely used as CMake, which until now I had not worked with it in a complex project; and third, with a

hardware for satellite communications, as learning a bit more about this industry has been really interesting. I consider that all of these fields of knowledge are going to be useful in my career, since they contribute very positively in the work with IoT embedded systems. Furthermore, I think that getting to understand and try to participate in an open-source project is a less common task than it should be, and having learned about the process and the requirements for this is also very attractive to me.

BIBLIOGRAFÍA

[1] Satlink, S.L., «Sitio Web Oficial de Satlink, S.L.» https://www.satlink.es/.

[2] ArTeCS, «Plataforma HW/SW para desarrollo de servicios de un dispositivo satelital sobre un lenguaje de programación imperativo» https://artecs.dacya.ucm.es/projects/technology-transfer-projects/dispositivo-satelital-imperativo/, 2013.

[3] Lua, «About» https://www.lua.org/about.html.

[4] CMake, «About CMake» https://cmake.org/overview/.

[5] Linux Programmer Guide's, «LIBC» https://man7.org/linux/man-pages/man7/libc.7.html.

[6] Wikipedia, «Newlib» https://en.wikipedia.org/wiki/Newlib.

[7] Zephyr Project, «Supported Boards» https://docs.zephyrproject.org/latest/boards/index.html#boards.

[8] Zephyr Project, «Introduction» https://docs.zephyrproject.org/latest/introduction/index.html.

[9] Zephyr Project, «Distinguishing Features» https://docs.zephyrproject.org/latest/introduction/index.html.

[10] Zephyr Project, «Network Stack Architecture» https://docs.zephyrproject.org/2.7.0/guides/networking/net-stack-architecture.html.

[11] Zephyr Project, «Zephyr Security Overview» https://docs.zephyrproject.org/3.0.0/security/security-overview.html.

[12] Zephyr Project, «Build and Configuration Systems» https://docs.zephyrproject.org/3.0.0/guides/build/index.html.

[13] Zephyr Project, «West» https://github.com/zephyrproject-rtos/west.

[14] Zephyr Project, «Basics» https://docs.zephyrproject.org/latest/develop/west/basics.html#west-basics.

[15] Espressif, «Update on Espressif's Support for Zephyr» https://www.espressif.com/en/news/Zephyr_updates, 2022.

[16] Espressif, «ESP32-DevKitC V4 Getting Started Guide» https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html.

[17] Zephyr Project, «NXP FRDM-K64F» https://docs.zephyrproject.org/2.6.0/boards/arm/frdm_k64f/doc/index.html.

[18] Wikipedia, «Iridium» https://es.wikipedia.org/wiki/Iridium.

[19] Iridium, «Sitio Web Oficial de Iridium» https://www.iridium.com/.

[20] J. Malsbury, «Iridium Modem (9602/9603) emulator» https://github.com/GLO-VT/Iridium-Uplink.

[21] Zephyr Project, «Minimal libc» https://docs.zephyrproject.org/latest/develop/languages/c/minimal_libc.html.

[22] Zephyr Project, «Newlib» https://docs.zephyrproject.org/latest/develop/languages/c/newlib.html.

[23] Lua, «Repositorio de Lua en Github» https://github.com/lua/lua.

[24] Ó. Baselga, «Adding, building and linking Lua in a project #45485» https://github.com/zephyrproject-rtos/zephyr/discussions/45485.

[25] Zephyr Project, «Modules (External projects)» https://docs.zephyrproject.org/latest/develop/modules.html#modules.

[26] Óscar Baselga, «Module request: Lua» https://github.com/zephyrproject-rtos/zephyr/issues/47301.

[27] Zephyr Project, «Contributing External Components» https://docs.zephyrproject.org/latest/contribute/external.html#external-src-process.

[28] Iridium, «Iridium 9603 Module» https://www.iridium.com/products/iridium-9603/.

[29] NAL Research Corporation, «AT COMMANDS FOR MODELS 9602-I/9602-N/9603-I V1.1» https://www.nalresearch.com/wp-content/uploads/2020/03/AT-Commands-for-Models-9603-TN2012-51-V1.0.pdf.

[30] SparkFun, «RockBLOCK Mk2 - Iridium SatComm Module» https://www.sparkfun.com/products/13745.

[31] Iridium, «Iridium 9603 SBD Transceiver Developer's Kit» Revision 1.0.

[32] Óscar Baselga, «Repositorio con las modificaciones en el emulador Iridium» https://github.com/oscarbaselga/virtual_iridium.

[33] J. Malsbury, «Little Python Saves Big on Iridium SBD» http://www.cnsp-inc.com/a-little-python-saves-big-on-iridium-sbd/.

APÉNDICES

Apéndice A - Información detallada sobre Zephyr

Licencia

```
Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction,
and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by
the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all
other entities that control, are controlled by, or are under common
control with that entity. For the purposes of this definition,
"control" means (i) the power, direct or indirect, to cause the
direction or management of such entity, whether by contract or
otherwise, or (ii) ownership of fifty percent (50%) or more of the
outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity
exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications,
including but not limited to software source code, documentation
source, and configuration files.
```

"Object" form shall mean any form resulting from mechanical
transformation or translation of a Source form, including but
not limited to compiled object code, generated documentation,
and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or
Object form, made available under the License, as indicated by a
copyright notice that is included in or attached to the work
(an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object
form, that is based on (or derived from) the Work and for which the
editorial revisions, annotations, elaborations, or other modifications
represent, as a whole, an original work of authorship. For the purposes
of this License, Derivative Works shall not include works that remain
separable from, or merely link (or bind by name) to the interfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity
on behalf of whom a Contribution has been received by Licensor and
subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work,

```
excluding those notices that do not pertain to any part of
the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its
distribution, then any Derivative Works that You distribute must
include a readable copy of the attribution notices contained
within such NOTICE file, excluding those notices that do not
pertain to any part of the Derivative Works, in at least one
of the following places: within a NOTICE text file distributed
as part of the Derivative Works; within the Source form or
documentation, if provided along with the Derivative Works; or,
within a display generated by the Derivative Works, if and
wherever such third-party notices normally appear. The contents
of the NOTICE file are for informational purposes only and
do not modify the License. You may add Your own attribution
notices within Derivative Works that You distribute, alongside
or as an addendum to the NOTICE text from the Work, provided
that such additional attribution notices cannot be construed
as modifying the License.

You may add Your own copyright statement to Your modifications and
may provide additional or different license terms and conditions
for use, reproduction, or distribution of Your modifications, or
for any such Derivative Works as a whole, provided Your use,
reproduction, and distribution of the Work otherwise complies with
the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
any Contribution intentionally submitted for inclusion in the Work
by You to the Licensor shall be under the terms and conditions of
this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify
the terms of any separate license agreement you may have executed
with Licensor regarding such Contributions.
```

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

```
END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following
boilerplate notice, with the fields enclosed by brackets "{}"
replaced with your own identifying information. (Don't include
the brackets!)  The text should be enclosed in the appropriate
comment syntax for the file format. We also recommend that a
file or class name and description of purpose be included on the
same "printed page" as the copyright notice for easier
identification within third-party archives.

Copyright {yyyy} {name of copyright owner}

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

Aspectos de seguridad

Hasta el momento, Zephyr ha centrado su implementación de seguridad en tres medidas fundamentales:

- La que denominan "security functionality", que se enfoca en algoritmos y protocolos criptográficos. Las funcionalidades criptográficas están disponibles por medio de bibliotecas. Actualmente, el usuario puede escoger entre dos en función de sus necesidades: TinyCrypt2 y mbedTLS. La primera de ellas soporta el uso de algoritmos de clave criptográfica para ser utilizados en las pilas de protocolos, sin embargo, el conjunto de algoritmos que ofrece es limitado. La segunda biblioteca proporciona un conjunto de algoritmos más amplio, a cambio de un coste adicional para ciertos requisitos, como el soporte a la función `malloc()`.

  Zephyr Project prevé incluir soporte hardware y software para soluciones de protección de claves basadas en SAM ("Secure Access Module"), TPM ("Trusted Platform Module") y TEE ("Trusted Execution Environment").

  Además, Zephyr Project comenta la importancia de compilar su kernel y la aplicación del usuario en un único binario estático, ya que el enlazado estático que emplean evita la posibilidad de que un atacante cargue un código malicioso de forma dinámica en el sistema.

- La conocida como "quality assurance", que es un proceso automático que se realiza antes de publicar nuevo código en el repositorio para comprobar la ausencia de vulnerabilidades, es decir, aplicar un análisis exhaustivo en el código. Este proceso se encarga de: verificar el correcto funcionamiento de la implementación, incrementar la legibilidad y mantenibilidad del código, asegurar un uso apropiado de las funciones de memoria y de cadenas de caracteres (que históricamente han sido las más vulnerables), validar los canales de entrada accesibles desde el exterior y comprobar el código más crítico para mantenerlo fuera de riesgos.

Todas estas tareas son llevadas a cabo por los miembros pertenecientes al comité de seguridad de Zephyr, que utilizan herramientas como Coverity Scan para mejorar la robustez del proceso.

- La nombrada "execution protection", que se centra en los aspectos de seguridad en tiempo de ejecución. En concreto, trata: la partición de la memoria en regiones independientes, que solo pueden ser accedidas por las tareas que tengan permiso explícito; la protección de la pila, detectando y procesando los desbordamientos; y el aislamiento de las tareas, haciendo que cada una tenga acceso únicamente a sus recursos de memoria (pila, variables, espacios reservados, etc.).

Apéndice B - Issue publicada en Zephyr Project con la propuesta del módulo *liblua*

```
## Origin

Lua - [https://github.com/lua/lua](https://github.com/lua/lua)

## Purpose

Lua is a scripting language that can be integrated into embedded systems
thanks to its lightweight. It is perfect for complex applications where a
two-parts separated implementation is needed: firmware and services.

Lua provides a C API that allows a bidirectional C-Lua interaction.

## Mode of integration

I would like to integrate Lua as an external module in order to give users
the posibility to include Lua in their applications whenever they want. I
suggest making an integration of Lua as a library rather than as an
interpreter, so an appropriate name is `liblua` and could be located in
`modules/lib/`.

The goal is to add the Lua repository in the west manifest and define a
configuration option which enable the module in the user application.

## Maintainership

I present myself as a single maintainer (@oscarbaselga).

## Pull Request

TBD

## Description

As previously mentioned, Lua is a lighweight scripting language that can be
integrated in embedded systems. A scripting language will allow users to
```

```
implement their own services over their firmwares and manage their
applications from a higher level view.


Currently, Lua is not supported in Zephyr applications, so users would need
to include all its source files to use it, something that also implies to
re-write the building process of their projects.


Lua is a proper solution due to its lightweight and license, as the
repository is under an OSI-compliant license.


However, there is a support problem with stubs. There are three functions
in newlib that are not implemented: `_times()`, `_unlink()` and `_link()`.
All of them are necessary to build Lua, so I decided to declare them in an
auxiliary C file and throw a warning if they are called. I consider this
problem out of the module flow, so my solution is a temporary approach
while waiting for a newlib update.


## Dependencies


Lua module depends on file system and newlib components, which are enabled
with `CONFIG_FILE_SYSTEM` and `CONFIG_NEWLIB_LIBC` respectively. Moreover,
the main stack size must be set at 4096 (`CONFIG_MAIN_STACK_SIZE=4096`).


## Revision


Lua v5.4.4


## License


MIT License -
[http://www.lua.org/license.html](http://www.lua.org/license.html)
```