Departamento:
Departamento de Ingeniería e Investigaciones Tecnológicas
Programa de acreditación:
CyTMA2

Programa de Investigación[1]:

Código del Proyecto:

Título del proyecto
Entorno de integración continua para validación de sistemas embebidos de tiempo real
PIDC: ☐
Elija un elemento.

PII: ☐
Elija un elemento.

Director:
Lic. Graciela Elisabeth De Luca
Director externo:

Codirector:
Ing. Waldo Adolfo Valiente
Integrantes:
Ing. Esteban Carnuccio; Ing. Mariano Volker
Investigador Externo, Asesor- Especialista, Graduado UNLaM:

Alumnos de grado: (Aclarar si tiene Beca UNLaM/CIN)
Sr. Raúl Villca; Sr. Matías Adagio (*Sin beca*)
Alumnos de posgrado:

Resolución Rectoral de acreditación: N°
249/20

Fecha de inicio:
01/01/2020
Fecha de finalización:
31/12/2021

[1] Los Programas de Investigación de la UNLaM están acreditados con resolución rectoral, según lo indica la Resolución HCS Nº 014/15 sobre **Lineamientos generales para el establecimiento, desarrollo y gestión de Programas de Investigación a desarrollarse en la Universidad Nacional de La Matanza**. Consultar en el departamento académico correspondiente la inscripción del proyecto en un Programa acreditado.

A. Desarrollo del proyecto (adjuntar el protocolo)

A.1. Grado de ejecución de los objetivos inicialmente planteados, modificaciones o ampliaciones u obstáculos encontrados para su realización (desarrolle en no más de dos (2) páginas)

Con respecto a los tres objetivos específicos que se desarrollaron en esta etapa. Uno de ellos estaba pendiente desde el año anterior y los otros dos corresponden al presente estadio de la investigación. De ellos se puede comentar lo siguiente:

A) Construcción de ambiente automatizado para realizar las pruebas y validaciones de optimización sobre los sistemas embebidos reales o virtuales.

Este objetivo se encontraba pendiente desde la primera parte del proyecto. Ahora se logró desarrollarlo, con el uso de contenedores Docker, permite generar un ambiente encapsulado de implementación y prueba de sistemas embebidos. El encapsulamiento admite que a partir de la misma imagen Docker, se puedan lanzar contenedores, con la emulación del sistema embebido, enfocado en diferentes ambientes de prueba, que son descartados una vez finalizada la ejecución. El único punto pendiente fue la ejecución automática del sistema embebido. Para ello se realizaron pruebas con la herramienta JENKINS. El contratiempo que nos encontramos fue que la herramienta requiere que la ejecución, nodo de Jenkins, brinde un resultado o finalice para validar su código de retorno. Pero los sistemas embebidos están preparados para una ejecución continua. Otro punto que hubiera servido como validación era que, sí en la ejecución el sistema embebido realizaba algún tipo de petición a un sistema externo, como invocar algún servicio web o transferir información. Se hubiera validado si la petición era realizada. Pero, esta clase de simulaciones estaba fuera del alcance del presente proyecto. Sin embargo, este tipo de situaciones presentó las bases para un próximo proyecto de investigación. –Objetivo ejecutado al 80%-

B) Seleccionar las mejores herramientas (para depuración, medición de performance, Virtualización o Simulación) con licencia GPL o similar. Que permita el uso de estas tanto en la investigación como en el aula.

Para este objetivo se estudiaron diferentes formas de realizar métricas sobre sistemas embebidos emulados. En donde se determinó que el mejor depurador para controlar la ejecución del código en estas plataformas es a través de la herramienta GDB. Para ello se emplea el programa específico del kit de desarrollo, llamado *arm-none-abi-gdb*. Dicha depuración debe ser realizada, fuera del contenedor, desde una terminal se accede al servidor de GDB, que se ejecuta dentro del contenedor junto con la imagen del sistema embebido emulado. Por otro lado, para medir el desempeño del sistema embebido mientras es emulado por la herramienta Qemu dentro del contenedor, se validaron diferentes herramientas con resultados variados. En primer lugar, se adaptó la herramienta de *profiling* GPROF. No obstante, después de su implementación, no se obtuvieron los resultados esperados,

por lo que, fue descartado. Otra alternativa, fue el mecanismo de medir los *Ticks* que brinda como información el procesador, a través de la captura de sus interrupciones internas *Systick*. De esta forma se puede determinar la cantidad de tiempo que demora en ejecutarse una porción del programa. Otra variante, que se utiliza en sistemas embebidos que emplean la biblioteca *FreeRTOS*, es utilizar la herramienta de *profiling* que brinda dicho Sistema Operativo. Para ello se debió realizar diferentes adaptaciones en el código del programa. No obstante, se observa que en el simulador esta clase de mediciones pueden presentar un tipo de desfasaje. Por lo que en algunas mediciones no se consigue obtener el resultado deseado. -Objetivo ejecutado al 70%-

C) Identificar técnicas de optimización sobre algoritmos RTOS que serán aplicadas y validadas.

Sobre este objetivo, los sistemas embebidos poseen ciertas particularidades en la optimización, que no lo tienen los programas tradicionales. Se pueden identificar tres campos de acción que infieren a la optimización. Ellos son el consumo de memoria, el consumo energético y eficiencia del programa. Por el lado de optimizar el tamaño que ocupa un programa que ejecuta en sistemas embebidos. Hay dos técnicas sencillas, una consiste en reducir las estructuras de datos ociosas o sobredimensionadas. También es posible sustituir llamadas recursivas por bucles, para reducir el consumo de la memoria del stack. Estas optimizaciones pueden detectarse con la información en la etapa de la compilación. Por el lado de las optimizaciones del consumo energético. Se puede reducir la frecuencia de trabajo del procesador o incluso poner al procesador en modo de dormir (inactivo), para que espere hasta que sea reactivado con el manejo de alguna interrupción. Esta opción se encuentra disponible en algunos procesadores. Estas técnicas se aplican en la etapa de ejecución. La primera aplica a cambiar la configuración por defecto, en el inicio del programa desde el *bootloader*. Mientras que la segunda, se debe implementar explícitamente en la lógica del programa. Para la optimización que falta, de la eficiencia del sistema embebido, es importante medir y conocer el desempeño del programa (objetivo anterior). De esta forma se logra un fino equilibrio de las partes del programa que merecen ser optimizadas. En este equilibrio hay que tener en cuenta las limitaciones físicas del *chip*. Por ejemplo, la interfaz de entrada y salida, dado que las limitaciones en el tiempo de respuesta están dadas por el propio dispositivo. –Objetivo ejecutado al 90%—

B. Principales resultados de la investigación

B.1. Publicaciones en revistas (informar cada producción por separado)

Artículo Revista ReDDI 2021	
Autores	*Esteban CARNUCCIO, Waldo VALIENTE, Mariano VOLKER*
Título del artículo	*CREACIÓN DE ENTORNO INTEGRADO PARA SISTEMAS EMBEBIDOS*
N° de fascículo	*1*
N° de Volumen	*6*
Revista	*Revista Digital del Departamento de Ingeniería*
Año	*2021*
Institución editora de la revista	*Departamento de Ingeniería e Investigaciones Tecnológicas. Universidad Nacional de La Matanza*
País de procedencia de institución editora	*Argentina*
Arbitraje	SI
ISSN:	*2525-1333*
URL de descarga del artículo	*Url*
N° DOI	*No tiene*

B.2. Libros

Sin publicaciones de libros.

B.3. Capítulos de libros

Sin publicaciones de capítulos de libros.

B.4. Trabajos presentados a congresos y/o seminarios

Primer artículo - WICC 2020	
Autores	*Valiente, W.; Carnuccio, E.; Volker, M.; De Luca, G.; Villca, R. & Adagio, M.*
Título	*Entorno de integración continua para validación de sistemas embebidos de tiempo real*
Año	*2020*
Evento	*XXII Workshop de Investigadores en Ciencias de la Computación (WICC)*
Lugar de realización	*Universidad Nacional de la Patagonia Austral (UNPA), Santa Cruz, Argentina*
Fecha de presentación de la ponencia	*18 Junio*
Entidad que organiza	*Red de Universidades con Carreras en Informática*
URL de descarga del trabajo (especificar solo si es la descarga del trabajo; formatos pdf, e-pub, etc.)	*Url*

Segundo artículo - CACIC 2020	
Autores	*Carnuccio, E.; De Luca, G.; Valiente, W.; Volker, M.; Villca, R. & Adagio, M.*
Título	*Análisis de rendimiento y consumo para sistema embebido con requisitos de tiempo explícitos*
Año	*2020*
Evento	*XXVI Congreso Argentino de Ciencias de la Computación (CACIC)*
Lugar de realización	*Universidad Nacional de la Matanza (UNLaM), San Justo, Buenos Aires, Argentina*
Fecha de presentación de la ponencia	*8 de Octubre*
Entidad que organiza	*Red de Universidades con Carreras en Informática*
URL de descarga del trabajo (especificar solo si es la descarga del trabajo; formatos pdf, e-pub, etc.)	*Url*

Tercer articulo - WICC 2021	
Autores	*Waldo Valiente,* *Esteban Carnuccio,* *Mariano Volker,* *Graciela De Luca,* *Raúl Villca,* *Matías Adagio*
Título	*Entorno de Contenedores de Emuladores que contienen Sistemas Embebidos*
Año	*2021*
Evento	*XXIII Workshop de Investigadores en Ciencias de la Computación (WICC 2021)*
Lugar de realización	*Universidad Nacional de Chilecito (UNdeC), Chilecito, La Rioja (Modalidad Virtual)*
Fecha de presentación de la ponencia	*Abril 2021*
Entidad que organiza	*Red de Universidades con Carreras en Informática*
URL de descarga del trabajo (especificar solo si es la descarga del trabajo; formatos pdf, e-pub, etc.)	*Url*

Cuarto articulo – CACIC 2021	
Autores	*Esteban Carnuccio,* *Waldo Valiente,* *Mariano Volker,* *Raúl Villca,* *Matías Adagio*
Título	*Entorno de contenedores con emuladores de sistemas embebidos STM32*
Año	*2021*
Evento	*XXVII Congreso Argentino de Ciencias de la Computación (CACIC 2021)*
Lugar de realización	*Universidad Nacional de Salta (UNSa), Salta (Modalidad Virtual)*
Fecha de presentación de la ponencia	*Octubre 2021*
Entidad que organiza	*Red de Universidades con Carreras en Informática*
URL de descarga del trabajo (especificar solo si es la descarga del trabajo; formatos pdf, e-pub, etc.)	*Url*

B.5. Otras publicaciones

Sin otras publicaciones.

C. Otros resultados. Indicar aquellos resultados pasibles de ser protegidos a través de instrumentos de propiedad intelectual, como patentes, derechos de autor, derechos de obtentor, etc. y desarrollos que no pueden ser protegidos por instrumentos de propiedad intelectual, como las tecnologías organizacionales y otros. Complete un cuadro por cada uno de estos dos tipos de productos.

C.1. Títulos de propiedad intelectual. Indicar: Tipo (marcas, patentes, modelos y diseños, la transferencia tecnológica) de desarrollo o producto, Titular, Fecha de solicitud, Fecha de otorgamiento

No fue requerido tramitar títulos de propiedad intelectual.

C.2. Otros desarrollos no pasibles de ser protegidos por títulos de propiedad intelectual. Indicar: Producto y Descripción.

No fue requerido tramitar títulos de propiedad intelectual.

D. Formación de recursos humanos. Trabajos finales de graduación, tesis de grado y posgrado. Completar un cuadro por cada uno de los trabajos generados en el marco del proyecto.

D.1. Tesis de grado

Bajo esta investigación no se realizaron tesis de grado.

D.2 Trabajo Final de Especialización

Bajo esta investigación no se realizaron tesis de grado.

D.2. Tesis de posgrado: Maestría

Bajo esta investigación no se realizaron tesis de posgrado.

D.3. Tesis de posgrado: Doctorado

Bajo esta investigación no se realizaron tesis de doctorado.

D.4. Trabajos de Posdoctorado

Bajo esta investigación no se realizaron trabajos de posgrado.

E. Otros recursos humanos en formación: estudiantes/ investigadores (grado/posgrado/ posdoctorado)

Apellido y nombre del Recurso Humano	Tipo	Institución	Período (desde/hasta)	Actividad asignada[2]
Villca, Raúl David	Grado	UNLaM	01-2020 al 12-2021	El alumno realizo tareas de ampliación y mejora del Entorno construido de la imagen Docker
Adagio, Matías	Grado	UNLaM	01-2020 al 12-2021	El alumno realizo tareas de investigación de técnicas de *profiling*

F. Vinculación[3]: Indicar conformación de redes, intercambio científico, etc. con otros grupos de investigación; con el ámbito productivo o con entidades públicas. Desarrolle en no más de dos (2) páginas.

G. Otra información. Incluir toda otra información que se considere pertinente.

H. Cuerpo de anexos:

- Anexo I: Copia de cada uno de los trabajos mencionados en los puntos B, C y D, y certificaciones cuando corresponda.[4]
- Anexo II:
  - FPI-013: Evaluación de alumnos integrantes. (si corresponde)
  - FPI-014: Comprobante de liquidación y rendición de viáticos. (si corresponde)
  - FPI-015: Rendición de gastos del proyecto de investigación acompañado de las hojas foliadas con los comprobantes de gastos.
  - FPI-035: Formulario de reasignación de fondos en Presupuesto.
- Anexo III: Alta patrimonial de los bienes adquiridos con presupuesto del proyecto (FPI-017)

[2] Descripción de la/s actividad/es a cargo (máximo 30 palabras)

[3] Entendemos por acciones de "vinculación" aquellas que tienen por objetivo dar respuesta a problemas, generando la creación de productos o servicios innovadores y confeccionados "a medida" de sus contrapartes.

[4] En caso de libros, podrá presentarse una fotocopia de la primera hoja significativa o su equivalente y el índice.

- Nota justificando baja de integrantes del equipo de investigación.

.. San Justo, 24 de Febrero 2022 ..
Lugar y fecha

- Presentar una copia impresa firmada del presente documento junto con los Anexos, y enviar todo en archivo PDF por correo electrónico a la Secretaría de Investigación Departamental. **Límite de entrega: 28 de febrero de 2020**

ANEXO I: Publicaciones realizadas durante el proyecto de investigación.

En total se realizaron en total 5 publicaciones durante el periodo 2020 y 2021.

Código	FPI-009
Objeto	Guía de elaboración de Informe final de proyecto
Usuario	Director de proyecto de investigación
Autor	Secretaría de Ciencia y Tecnología de la UNLaM
Versión	5
Vigencia	03/9/2019

1. ReDDI –Artículo en Revista

Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza

ISSN: 2525-1333
Vol.: 6 - Nro. 1 (JULIO 2021)

Informe Técnico

CREACIÓN DE ENTORNO INTEGRADO PARA SISTEMAS EMBEBIDOS

BUILDING AN INTEGRATED ENVIRONMENT FOR EMBEDDED SYSTEMS

Esteban CARNUCCIO[(1)], *Waldo VALIENTE*[(2)], *Mariano VOLKER*[(3)]

[(1)] Departamento de Ingeniería e Investigaciones Tecnológicas – Universidad Nacional de La Matanza
ecarnuccio@unlam.edu.ar

[(2)] Departamento de Ingeniería e Investigaciones Tecnológicas – Universidad Nacional de La Matanza
https://orcid.org/0000-0003-1821-1554
wvaliente@unlam.edu.ar

[(3)] Departamento de Ingeniería e Investigaciones Tecnológicas – Universidad Nacional de La Matanza
mvolker@unlam.edu.ar

Resumen:

Internet de las Cosas emerge entre los años 2008 y 2009, porque la cantidad de dispositivos conectados a Internet empezaron a superar al número de habitantes en el planeta. En la actualidad se estiman que existen un total de 50 Billones de dispositivos interconectados. Dada esta enorme cantidad, en su mayoría Sistemas embebidos, gestionar la configuración en un proyecto se vuelve una tarea titánica. Esto se debe a la gran cantidad de fabricantes, diversos modelos existentes y herramientas específicas que se requieren. Con el fin de simplificarlo, buscamos la manera de generar un entorno de trabajo de integración multiplataforma. Para ello analizamos pros y contras de utilizar máquinas virtuales o contenedores Docker. De esta forma se pretende armar un entorno que permita generar programas de dispositivos Arduino, STM32 y ESP32.

Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

Abstract:

Internet of Things emerged between 2008 and 2009, as the number of devices connected to the Internet began to exceed the cant of people on the planet. Currently it is estimated that there are a total of 50 Billion interconnected devices. Given this huge amount, mostly Embedded Systems, managing the configuration in a project becomes a titanic task. This is due to the large number of manufacturers, diverse existing models and specific tools that are required. In order to simplify it, we are looking for a way to generate a multiplatform integration work environment. To do this, we analyze the pros and cons of using virtual machines or Docker containers. In this way, it is intended to build an environment that allows us to generate programs for Arduino, STM32 and ESP32 devices.

Palabras Clave: Integración, Docker, Compilador Cruzado, Sistema Embebido

Key Words: Integration, Docker, Cross Compiler, Embedded System

Colaboradores: *Matías Adagio, Raúl Villca.*

Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

I. CONTEXTO

El término internet de las cosas "Internet of Things" (IoT) fue utilizado por primera vez en 1999 por Kevin Asthon, en una conferencia de RFIDs[1] para llamar la atención de la audiencia sobre estos componentes y su conexión a Internet [1]. Por otro lado, el informe técnico de Cisco Internet Business Solutions Group [2], establece la fecha del nacimiento de IoT entre los años 2008 y 2009. Porque en ese período es la primera vez que se superó el número de las "cosas" o los dispositivos conectados a Internet, que individuos viviendo en el planeta. Ese volumen de dispositivos sigue aumentando año a año, hasta alcanzar en la actualidad un total de 50 Billones. En este marco desarrollar software para los dispositivos se vuelve una tarea compleja, por la gran cantidad de fabricantes y la multiplicidad de modelos, que pueden adecuarse a las necesidades del producto a desarrollar. Por ese motivo iniciamos la línea de investigación, bajo el proyecto CyTMA2, llamado "Entorno de integración continua para validación de sistemas embebidos de tiempo real". En la que buscamos integrar las etapas de construcción del software, desarrollo y prueba, para diferentes Sistemas Embebidos (SE) en un único entorno común.

II. INTRODUCCIÓN

Para lograr un entorno integrado, que reúna las diferentes herramientas para compilar los múltiples embebidos, se necesita de un entorno que soporte y que simplifique las necesidades de las personas que intervienen en el proyecto. Con la premisa de que sea multiplataforma, para ser utilizada en diferentes Sistemas Operativos, como Linux o Windows. Por lo que mantener una gestión de configuración estos requerimientos, es una tarea compleja. En este sentido, una de las posibilidades analizadas, consistió en armar una máquina virtual con el software ya instalado. Pero su tamaño final, en el orden de los Gigabytes, resultaba complejo compartirla entre los participantes del proyecto. Sin contar con el mantenimiento que este requiere. Como solución a esta problemática nos encontramos con Docker y su alternativa basada en contenedores. Por ese motivo en este artículo, se explicarán las nociones básicas de contenedores, de cómo construir el software necesario para trabajar con los SE y la facilidad de compartir el entorno integrado.

Para explicar el concepto principal de Docker, expondremos la analogía que se explica en [3]: "Antes los estibadores requerían habilidades, ellos son los trabajadores encargados de mover mercancías comerciales dentro y fuera de los barcos en el puerto. Las mercancías se componían de cajas y artículos de diferentes tamaños y formas. Los experimentados estibadores eran apreciados por su capacidad para acomodar los distintos tipos de mercancías dentro de los barcos. Contratar a estas personas no era barato, pero hacían un trabajo realmente eficiente. Como una mejora surgieron los contenedores marítimos, que son cubos rectangulares de iguales proporciones, que permiten simplificar la carga y descarga de los barcos". Ahora esas mercancías, de formas irregulares, se guardan desde el origen dentro del contenedor antes de llegar al puerto. Esto permite que los barcos sean cargados y descargados mucho más rápido, incluso con sistemas automatizados. Ya que los contenedores poseen un tamaño estándar. Esta metáfora es conocida en el ámbito de proyectos de software, porque se invierte mucho tiempo y energía en conseguir software heterogéneo, de forma análoga a las

[1] RFID: (Radio Frequency Identification) Sistema remoto de almacenamiento y recuperación de datos de identificación.

Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

mercancías. Estos se integran de formas complejas en un sistema, siendo este equivalente al barco. En este sentido, Docker ha cambiado todo esto, permitiendo que los diferentes involucrados en el proceso de desarrollo, hablen un mismo idioma de forma eficaz. Haciendo de esta manera, que trabajar en equipo sea más sencillo. Ya que no es necesario seguir manteniendo una variedad desconcertante de configuraciones de cada software integrado. Gracias a que ahora pueden coexistir en contenedores y en forma independientes entre sí.

III. DESARROLLO

Para explicar cómo se realizó la investigación. Comenzaremos analizando el paradigma de Máquina virtual versus Docker. Luego explicaremos cuáles son sus componentes. Finalmente evaluaremos las herramientas necesarias para generar programas dentro del contenedor Docker, que podrá ejecutarse en los SE.

a. Paradigma Máquina Virtual versus Docker

Para entender las principales diferencias entre Docker y máquinas virtuales, se explicará con la analogía de las cualidades que tiene una casa y un edificio [3]. Una casa, tiene sus propias conexiones de tuberías, electricidad, calefacción y su entrada principal. Mientras que un edificio tiene los mismos recursos que una casa, pero estos son compartidos entre todas sus unidades habitacionales. Dentro de los departamentos, existen diferentes cantidades de habitaciones y comodidades. Solo se adquiere el departamento que tiene lo estrictamente necesario, no todo el complejo. Los departamentos del edificio serían los contenedores, y los recursos compartidos son el anfitrión del contenedor. Por otro lado, las casas, con sus instalaciones completas, serían la máquina virtual.

Una máquina virtual es un software que emula una computadora, generalmente para ejecutar un sistema operativo y sus aplicaciones. El usuario final utiliza a los programas como si estuviera en una máquina física, pero aquellos que administran el hardware pueden centrarse en la asignación de los recursos a mayor escala [4]. Mientras que Docker es un conjunto de todos los archivos que componen el software. Esta colección incluye a la aplicación más todas las bibliotecas, binarios y otras dependencias, todo lo necesario para ejecutar el programa que se almacena en forma encapsulada.

b. La estructura interna de Docker

De la misma manera que un software puede verse como un programa en ejecución, una imagen de Docker puede verse como un contenedor en ejecución. Docker se construye con imágenes, que son de solo lectura. Por lo tanto, su contenido no se puede alterar. Como consecuencia, se pueden crear varios contenedores a partir de una sola imagen, donde cada una de las instancias están aisladas entre sí. Cualquier cambio realizado en el contenedor, no afectará la definición de la imagen.

b.1. Imagen de Docker

Una imagen de Docker se compone internamente de capas. De esta forma la arquitectura de composición de la imagen aprovecha eficazmente el contenido de cada una de ellas. Esto permite que se pueda ir agregando nuevas funcionalidades adicionales a la imagen, con el fin de satisfacer las diferentes necesidades y aumentar la reutilización entre sus capas. En otras palabras, las funcionalidades se van sumando a la imagen, apilándose una sobre otra, agregando así niveles adicionales de capas. Además, cada una de ellas tiene una relación entre

Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

padres e hijos. En donde, la capa de la parte inferior es llamada capa base [5]. La cual es un nivel especial, que no tiene ningún padre.

En la fig. 1, se puede observar un ejemplo de la composición de las capas que constituye una imagen. En este caso la capa base se conforma de una versión adaptada del Sistema Operativo *Ubuntu*. Esta ofrece la posibilidad utilizar funcionalidades y bibliotecas de ese Sistema Operativo. Así como también, manejo de archivos, conexión HTTP a internet y la utilización de algunos comandos básicos de Linux. En el siguiente nivel, se agrega la capa 2. En ella se instala el software de versionado "*git*", que permite manejar repositorios del tipo GitHub[2]. Por otra parte, en la capa 3, se usa el comando git para descargar archivos de repositorios web. Como este software de versionado necesita conectarse a internet para funcionar, debe hacer referencia a la capa base de Ubuntu, para poder así hacer uso de sus servicios de conexión. De esta manera utiliza los recursos de acceso a internet que brinda el Sistema Operativo. Por lo tanto, la imagen de este ejemplo está compuesto por 3 capas no modificables. Las cuales se pueden instanciar y utilizar a través de un contenedor, que conformará una cuarta capa que se puede alterar. De esta manera, dentro del contenedor, al ser una instancia de la imagen, se pueden copiar archivos fuentes, scripts o de configuración, desde diferentes repositorios. Como se mencionó, lo que se realice en el contenedor, existirá como una capa modificable. Mientras que las capas inferiores que componen a la imagen no sufrirán alteraciones. Gracias a estas cualidades, a partir de una imagen se pueden generar tantos contenedores que funcionan independientemente.

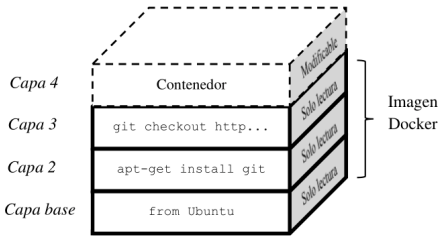


Fig. 1. Capas que forman la imagen en Docker.

Por otro lado, al generar una nueva imagen se reutilizan capas, si es que requiere utilizar algún componente que pertenezca a otra imagen. De esta manera no se agregarían como capas nuevas, sino que serían reutilizadas por el motor Docker. Ahorrando así recursos y tiempo de generación.

b.2. Dockerfile

Los archivos *Dockerfiles* se utilizan para generar imágenes personalizadas por los usuarios. Son archivos de configuración en donde se especifican los comandos, en forma de meta instrucciones, que Docker utilizará para construir una imagen deseada. *Dockerfiles* proporciona un lenguaje común, simple y universal para el aprovisionamiento de Imágenes en Docker. Dentro de ellos, se puede usar cualquier cosa que se desee para alcanzar el fin deseado. Para ello se debe hacer uso de los comandos provistos por Docker (RUN, FROM, COPY, etc.). A su vez, todos los programas agregados dentro de una capa posteriormente pueden ser utilizados para generar nuevas capas.

c. Implementación de los Sistemas Embebidos

[2] **GitHub**: Servicio basado en la nube que aloja un sistema de control de versiones.

Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

Con la finalidad de armar un entorno de compilación para los SE, que permita generar binarios que ejecuten en el dispositivo embebido, resulta necesario llevar a cabo un proceso denominado Compilación Cruzada. Este consiste en convertir el código fuente del programa en un archivo binario, que es independiente a la plataforma que lo compila, para que ejecute en el SE. De esta manera tendrá un set de instrucciones distinto al equipo donde se genera el archivo ejecutable. Para este trabajo se utiliza los compiladores cruzados para las arquitecturas de Arduino, STM32 y ESP32. Las cuales funcionan dentro de la imagen de Docker.

c.1 Implementación en Arduino

Se utiliza el proyecto *Ino* [6], para convertir el código fuente de Arduino en un archivo binario, que ejecuta en dicha arquitectura. La cual es una herramienta que, desde línea de comandos, permite compilar, implementar y depurar programas para Arduino. Además, permite trabajar con los modelos conocidos; tales como Arduino Uno, Nano, Mega, etc. Para que esta herramienta pueda funcionar, tiene como prerrequisitos: el programa *git*, el intérprete de Python versión 3, el módulo Pyserial (para hacer uso de la conexión por terminal serial), entre otros. Por lo que estos, deben estar instalados como capas previas en la imagen de Docker.

c.2 Implementación en STM32

Este tipo de plataforma utiliza el conjunto de herramientas embebidas para las arquitecturas ARM, denominado *arm-none-eabi* [7]. Por ese motivo se debe instalar su compilador cruzado *gcc-arm-none-eabi*, que permite generar ejecutables que funcionan en STM32. Esta debe ser incluida en una capa de la imagen de Docker, junto a sus dependencias y a otras herramientas. Como dependencia se encuentra la biblioteca *libnewlib-arm-none-eabi* que brinda funcionalidades de inicio y configuración a los ejecutables. Por otra parte, se debe instalar el depurador *arm-none-eabi-gdb*, que se utiliza para depurar programas que ejecutan en este tipo de SE.

c.3 Implementación en ESP32

Al querer desarrollar programas que funcionen en las arquitecturas ESP32 desde Docker, resulta necesario instalar el ambiente de desarrollo de Software, denominado ESP-IDF [8]. El cual es un producto de la empresa Espressif System CO. LTD. Además, se necesita tener instaladas en las capas previas, las dependencias de ESP-IDF, que son: los programas git, *cmake*, el intérprete de *python3*, la biblioteca *libusb*, entre otras. Al ejecutar el instalador de ESP-IDF, automáticamente se instalan las distintas herramientas necesarias para generar y depurar los programas en las plataformas ESP32. Entre ellas se encuentra el compilador llamado *xtensa-ESP32-elf -gcc* y su depurador *xtensa-ESP32-elf-gdb.*

d. Generación de imagen integrada de herramientas

Como puede verse las dependencias de los compiladores cruzados tienen paquetes en común: Tales como el intérprete de *Python*, el manejador de repositorios *git*, y el programa de dependencias *make*, entre otros. Esto permite tener una capa de dependencias compacta, que provea todo lo necesario para que las herramientas sean generadas correctamente dentro de una sola imagen.

Las primeras versiones de las imágenes Docker, que fueron generadas en esta investigación, existían en forma separada. No obstante, con el avance del trabajo, se detectó que las imágenes resultantes ocupaban mucho espacio en el disco. Esto resultaba llamativo, ya que a pesar de limpiar los archivos temporales al finalizar la

Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

instalación, aun así no se reducía el tamaño de la imagen. Luego de analizar en detalle se detectó la causante, esta se produce al generar la imagen. Ya que cada invocación al comando *RUN*, configurado en el archivo *Dockerfile*, genera nuevas capas. Entonces si en una capa inferior, se cargan archivos, que luego son eliminados desde un nivel superior, estos no resultan visibles en las capas siguientes, ni tampoco por el contenedor. Sin embargo, seguían perteneciendo a la historia de la imagen, como capas inferiores, ocupando el espacio en la misma. Esto se visualiza en el siguiente ejemplo, que se muestra en la fig. 2. En la parte Izquierda de la ilustración, se ejecuta a cada uno de los comandos *RUN* en capas separadas, para formar así la imagen de Docker. En detalle la Capa 3, se descarga el código fuente del repositorio X. Luego este es compilado e instalado en las capas siguientes. En la capa 6, se eliminan los archivos fuentes y temporales. Ya que solamente se requiere del archivo binario ejecutable, no los recursos para generarlo. Posteriormente, al instanciar esta imagen en un contenedor, se verifica que no existen los archivos eliminados previamente. Pero como puede verse en la figura, las capas 3, 4 y 5 son parte de la historia de la imagen resultante, por lo que siguen perteneciendo a ella.

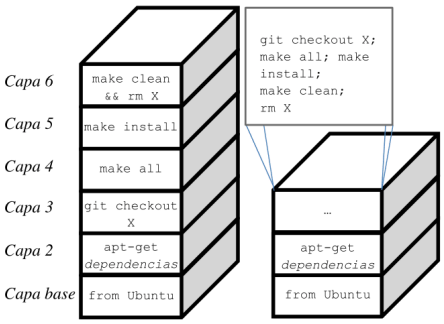


Fig. 2. Izquierda imagen de Docker compacta; Derecha imagen de Docker sin optimizar.

La solución a este comportamiento se muestra en la parte derecha de la fig. 2. Esta consiste en ejecutar la mayor cantidad de comandos de creación, compilación y eliminación de archivos temporales dentro de una misma capa. De esta forma la capa resultante solamente contendrá el archivo ejecutable buscado. El conjunto de capas creadas de esta forma, permite que la imagen sea mucho más liviana que la anterior. En nuestro proyecto la diferencia fue 8 Gigabytes para la imagen original y 1,5 Gigabytes para la imagen compacta. Este tipo de creación de capas, trae un pequeño sacrificio. La capa 3 en la imagen compacta, es muy específica, por lo que es muy raro que sea compartida en otra imagen.

IV. CONCLUSIONES

El entorno de integración propuesto en este trabajo, se basa en la generación de las herramientas de compilación cruzada y no en utilizar solamente los recursos del sistema operativo. Por ende, se determinó conveniente utilizar Docker, debido a que resulta ser más liviano que las máquinas virtuales. No solo por el tamaño que ocupa la imagen. Sino que también simplifica la forma de compartirlo entre los integrantes del proyecto, ya que el archivo de configuración *Dockerfile*, tiene todo lo necesario para recrear la imagen y su tamaño es de apenas unos pocos Kilobytes. Este archivo indica la secuencia de comandos requerida para la creación de la imagen, que implícitamente documenta los paquetes y las dependencias necesarias para la creación del entorno. Por lo tanto, la imagen resultante puede ser instanciada en distintos contenedores, que no modifican a la imagen original. Estos dos puntos son vitales para mantener una gestión de configuración limpia. Al mismo tiempo, Docker es multiplataforma, por lo que la misma imagen puede instanciarse desde distintos Sistemas Operativos,

Revista Digital del Departamento de Ingeniería e Investigaciones Tecnológicas de la Universidad Nacional de La Matanza

ISSN: 2525-1333. Vol.: 6 - Nro. 1 (JULIO 2021)

en contenedores que poseen el mismo entorno de trabajo. Además, los contenedores proporcionan un entorno aislado, que puede ser utilizado para generar imágenes de prueba, con actualizaciones o parches de seguridad. Los cuales se pueden aplicar sobre la imagen original. Permitiendo así realizar todas las pruebas necesarias, antes de publicar la nueva versión de la imagen del entorno integrado.

V. REFERENCIAS Y BIBLIOGRAFÍA

[1] M. Cañon y A. David, «Seguridad de la información en la internet de las cosas,» Universidad Piloto de Colombia, Bogotá, 2016.

[2] E. Dave, «The Internet of Things - How the Next Evolution of the Internet Is Changing Everything,» Cisco, United States, 2011.

[3] G. Sébastien, Docker Cookbook: Solutions and Examples for Building Distributed Applications, O'Reilly, 2015.

[4] M. Ian y S. Aidan Hobson, Docker in practice, Shelter Island, NY: Manning Publications Co., 2019.

[5] C. Jeeva S., V. S. y R. Pethuru, Learning Docker - Second Edition: Build, ship, and scale faster, Birmingham, Reino Unido: Packt Publishing, 2017.

[6] C. David, B. Jared, E. Lars, R. Alberto, S. Michael, P.-L. Marc y K. Fabian, «GtiHub - amperka/ino:,» 2014. [En línea]. Available: https://github.com/amperka/ino. [Último acceso: 18 06 2021].

[7] A. Limited, «Arm Developer,» 06 2021. [En línea]. Available: https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads. [Último acceso: 27 06 2021].

[8] E. Systems, «GitHub - Espressif IoT Development Framework,» 2021. [En línea]. Available: https://github.com/espressif/esp-idf. [Último acceso: 18 06 2021].

Recibido: 2021-06-28
Aprobado: 2021-07-19
Hipervínculo Permanente https://reddi.unlam.edu.ar/index.php/ReDDi
Datos de edición: Vol. 6 - Nro. 1 - Art. 2
Fecha de edición: 2021-07-27

2. ReDDI –Certificado

3. WICC 2020 – Poster

Entorno de integración continua para validación de sistemas embebidos de tiempo real

Waldo Valiente, Esteban Carnuccio, Mariano Volker, Graciela De Luca, Raúl Villca, Matías Adagio

Departamento de Ingeniería e Investigaciones Tecnológicas
Universidad Nacional de La Matanza

ARSO – Arquitectura, Redes y Sistemas Operativos

SODIUM

Resumen

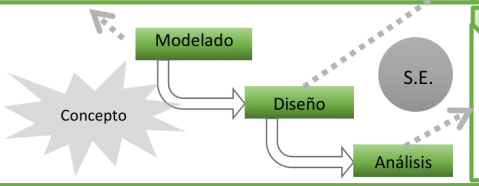
El proceso de selección de **Sistemas Embebidos** (SE) a utilizar se vuelve una tarea compleja. En parte por la cantidad de fabricantes y características en sus modelos, sumado a la necesidad del cumplimiento de requisitos no funcionales. Este trabajo se centrará en que esta tarea, como así también del proceso de selección/elaboración de un sistema embebido de **tiempo real** para lograr, incluso, su óptima replicación. Por ese motivo se pretende desarrollar un entorno de **mejora continua** que facilite el desarrollo, pruebas y optimización de los sistemas embebidos. Además se pretende que el conjunto pueda ejecutar sobre distintos **dispositivos**, tanto **real** como **virtual**. Permitiendo así la correcta selección del dispositivo, incluso antes de ser adquirido.

Modelado

Se centra en la definición de la dinámica del comportamiento de las partes que formarán el S.E., como los eventos físicos serán medidos y convertidos en pulsos eléctricos por los sensores que van registrando eventos que accionan funcionalidades del S.E. para dar una salida por sus actuadores.

Diseño

Se define cómo se realiza la integración de las partes del S.E. y como obtener datos precisos de los sensores. También se define que funcionalidad aplicará (sobre cómo el sistema interactúa con su entorno). En esta etapa se plantean requisitos específicos y no funcionales sobre el S.E.



Análisis

Es utilizada para garantizar que el S.E. resultante, cumpla con la funcionalidad correcta, como con los requisitos planteados en las etapas de Modelado y Diseño.

Resultados Esperados

Se realizará un análisis sobre S. E. y los tipos de Sistemas Operativos de Tiempo Real (RTOS) que se pueden ejecutar. Para lograr comparar sus características, ventajas y desventajas; Se desarrollarán algoritmos y casos de prueba. Que ejecutarán automáticamente dentro del entorno de integración continua, que permitirá la comparación de los resultados de las distintas ejecuciones sobre S.E. reales y virtuales.

líneas de investigación

-Uso de virtualización para ejecutar diferentes S.E.
- Seleccionar algoritmos RTOS, para utilizarlos como caso de prueba.
- Construir un entorno de integración continua, para las pruebas.
- Aplicar indicadores de optimización en RTOS, que serán los indicadores de aceptación de las pruebas.

Formación de Recursos Humanos

- La presente línea de investigación dentro del departamento de Ingeniería e Investigaciones Tecnológicas, uno de los investigadores se encuentra realizando para su maestría.

- Completan el grupo de investigación dos de docentes de categoría V y dos ingenieros en formación de investigador.

4. WICC 2020 – Artículo

Entorno de integración continua para validación de sistemas embebidos de tiempo real

Waldo Valiente, Esteban Carnuccio, Mariano Volker, Graciela De Luca, Raúl Villca, Matías Adagio
Departamento de Ingeniería e Investigaciones Tecnológicas
Universidad Nacional de La Matanza
Dirección: Florencio Varela 1703 – CP 1754 – {wvaliente, ecarnuccio, mvolker, gdeluca}@unlam.edu.ar, {raul.villcasd, mati.adagio}@gmail.com

RESUMEN

Cada año se desarrollan billones de sistemas embebidos. Por lo que el proceso de selección del dispositivo a utilizar se vuelve una tarea compleja. En parte por la cantidad de fabricantes y características en sus modelos, sumado a la necesidad del cumplimiento de requisitos no funcionales, tales como la limitación en el consumo energético y la criticidad de los tiempos de respuesta a eventos externos. Este trabajo se centrará en que esta tarea sea factible en tiempos razonables desde el punto de vista del negocio, como así también del proceso de selección/elaboración de un sistema embebido de tiempo real para lograr, incluso, su óptima replicación y fabricación. Por ese motivo se pretende desarrollar un entorno de mejora continua que facilite el desarrollo, pruebas y optimización de los sistemas embebidos. La mejora continua permite generar ciclos de medición y retroalimentación de las tres partes que forman el proceso de desarrollo. Además se pretende que el conjunto pueda ejecutar sobre distintos dispositivos, tanto real como virtual. Permitiendo así la correcta selección del dispositivo, incluso antes de ser adquirido.

Palabras clave: *Sistemas Embebidos, Tiempo Real, Optimización, Virtual, Mejora Continua.*

CONTEXTO

Nuestra Línea de Investigación es parte del proyecto "*Entorno de integración continua para validación de sistemas embebidos de tiempo real"*, dependiente de la Unidad Académica del *Departamento de Ingeniería e Investigaciones Tecnológicas*, perteneciente al programa de Investigaciones CyTMA2 de la Universidad Nacional de La Matanza, el cual es formado por docentes e investigadores de la carrera de ingeniería en informática. Este proyecto es continuación de los trabajos que viene realizando el grupo de investigación en sistemas operativos, computación de alto rendimiento y en el área de Internet de las cosas.

1. INTRODUCCIÓN

En sus inicio los Sistemas Embebidos (SE), se requerían para funcionalidades sencillas, con el fin de obtener programas de reducido tamaño, debido a las limitaciones en la memoria que disponía el hardware en esa época. Con el paso del tiempo y gracias a los avances tecnológicos, los microcontroladores incrementaron la

cantidad de memoria. Por ende se volvieron más sofisticados, por lo que requirió mayor complejidad en la programación, hasta el punto de agregar funcionalidades complejas que pertenecen al ámbito de los Sistemas Operativos de Tiempo Real (RTOS). Con ello aparecieron múltiples modelos de dispositivos, confeccionados por diferentes fabricantes, cada uno con su propia configuración y conjunto de herramientas de uso. Con el auge de Internet de las cosas apareció un creciente desarrollo de los SE, por año se crean 6 billones de SE nuevos [1], mientras que computadoras de escritorio, solo rondan los 100 millones [2]. Existen proyectos científicos que acompañan esta evolución. Por un lado la *National Science Foundation*, el año pasado finalizó un proyecto de optimización semántica para RTOS que utilizan SE [3]. Mientras que la Comunidad Europea, planteó un proyecto con el objetivo de bajar el consumo energético para RTOS con altas prestaciones que ejecutan en SE [4]. Así mismo desarrollar software para Sistemas Embebidos de Tiempo Real (RTSE) es una tarea compleja. Está afirmación es justificada por la experiencia adquirida de parte del equipo de investigación en los proyectos "Sistema de monitoreo y alarma para personas adultas mayores" y "Dispositivo de asistencia de personas mediante monitoreo y análisis de datos en la nube".

Problemática a investigar:

Los dispositivos utilizados en SE poseen múltiples diferencias, ya que cada uno cuenta con sus herramientas propias por lo que trabajar con ellos requiere de reunir constantemente habilidades y conocimientos técnicos para desenvolverse en dicha área. También existen dos complicaciones en la selección y el uso de cada microcontrolador, ya que cada uno posee una tecnología propia. La primera de ellas surge debido a la correcta elección del dispositivo, más allá de las especificaciones básicas, junto con el costo y la facilidad de adquisición en el mercado local. Cuando se plantean requisitos no funcionales estrictos, como el tiempo de respuesta ante distintos eventos, o que el consumo energético permita que el dispositivo funcione con batería asegurando la disponibilidad, existe una amplia variedad de modelos, que pueden estar sobre dimensionados o no llegar a satisfacer estos requerimientos. No es económicamente viable adquirir tantos modelos de dispositivos para elegir el que mejor se adapte a los requisitos. Por lo que se propone de utilizar un SE virtual para realizar las pruebas de rendimiento, verificando distintas optimizaciones, antes de llegar a adquirir el hardware del dispositivo. La segunda complicación, ya con el dispositivo seleccionado, surge por el amplio conjunto de herramientas suministrado por los fabricantes de SE. Estas se enfocan en el modelado, desarrollo, depuración e implementación del programa en el SE. Por lo que se busca armar un entorno de trabajo que permita realizar la ejecución de SE virtuales y reales, y medir las características de optimización que aseguren los requisitos no funcionales, en el uso de los recursos del SE.

Por otra parte se encontraron problemas comunes inherentes a la funcionalidad de los RTSE, estos se describen a continuación.

Problemas Comunes

Según [5], "En sistemas embebidos el objetivo principal de la programación del proceso es garantizar una respuesta rápida a eventos externos y cumplir el tiempo de ejecución del mismo". Para lo cual las funciones de código específicas pueden dividirse en tareas. Cada una puede tener una prioridad, esta dependerá de su periodicidad o si son activados como respuesta a eventos externos. En los SE suele haber procesadores de un solo núcleo, por eso lo que se suele multiplexar uso del procesador. Esto se

efectúa para alternar el uso del procesador entre tareas, generando concurrencia. Para los SE complejos pueden tener procesadores de varios núcleos, logrando verdadera paralelización de tareas o realizando distintas tareas concurrentemente, lo que puede ocasionar retardos no buscados por problemas de sincronización debido al uso de semáforos, memoria compartida, tuberías, entre otros.

Los sistemas operativos que al principio se ejecutaban en un SE, eran programas con funciones muy simples de control, que esperaban eventos desde el exterior para funcionar. Actualmente como la demanda de los sistemas multifuncionales aumenta, también lo hace la infraestructura donde se ejecutan. Por lo que se dispone de sistemas operativos cada vez más complejos, como los Smartphone que tienen casi tantas características computacionales como los de una computadora de escritorio. Algunos poseen memoria en el orden de los Gigabytes, procesadores con varios núcleos y pantallas de alta definición. Por lo que ya no se suele realizar diseños propios del sistema operativo para cada SE. Lo que se realiza es adaptar el sistema operativo tradicional, mediante configuraciones a estas arquitecturas, por ejemplo Android baso su versión de GNU/Linux o Microsoft Windows® para Microsoft Surface Go®.

En el último tiempo hubo un incremento exponencial de Sistemas Embebidos de Tiempo Real (RTOS) [6]. El requisito de Tiempo Real es necesario para completar su trabajo y entregar servicios regularmente, ya que tienen requisitos de tiempo estricto que deben cumplirse. Todos los días estos sistemas proporcionan tareas importantes, por ejemplo al conducir, al volar, monitoreo de pacientes, etc. La diferencia de estos sistemas con el de PC tradicional, es que los últimos ejecutan aplicaciones complejas que no requieren de tiempo real, como navegadores, procesadores de texto, entre otros. La vertiente actual busca utilizar los recursos de SE junto con funciones específicas de concurrencia que poseen los Sistemas Operativos tradicionales. RTOS es la opción que logra combinar ambos mundos, ya que están preparados para ejecutar tareas que poseen requerimiento de tiempos estrictos. Un RTOS debe responder inmediatamente a eventos externos con la menor latencia de interrupción. Además, debe completar los pedidos de servicio antes que se cumpla el tiempo límite. Para lograr cumplir con estos requisitos tienen las siguientes capacidades [5]: Menor latencia de interrupción, regiones críticas cortas, administrador de tareas apropiativo y algoritmos de planificación dinámicos. Los dos primeros requisitos tienen que ver con acortar los tiempos de ejecución de RTOS y que las tareas tengan mayor tiempo de procesador. Los otros dos requisitos tienen que ver como son administrados los tiempos de procesador, permitiendo a tareas de mayor prioridad cumplir su tiempo límite. Algunos de los RTOS que cumplen con estos requisitos son FreeRTOS, MicroC/OS, NuttX y QNX.

2. LINEAS DE INVESTIGACIÓN Y DESARROLLO

Por esta creciente necesidad de programas más robustos se tuvo que establecer un proceso de desarrollo para SE, este se compone de tres etapas [7], representadas en (Fig. 1):

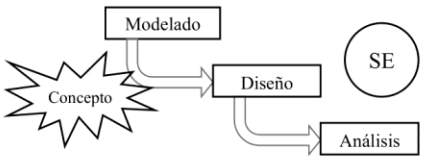


Fig. 1 Etapas del proceso de construcción de SE.

La concepción del SE parte desde una idea o concepto que se debe plasmar en el hardware.

1°) La primera etapa, la del Modelado, se centra

en la definición de la dinámica del comportamiento de las partes que formarán el SE, como los eventos físicos serán medidos y convertidos en pulsos eléctricos por los sensores. El paradigma que representa mejor esta lógica es a través de máquinas de estados concurrentes, donde los sensores van registrando eventos que accionan funcionalidades del SE para dar una salida por sus actuadores. Las herramientas que suelen utilizarse para el modelado son *Simulink*® y *LabVIEW*®. Estas son herramientas que permiten realizar pruebas, mediciones, controles y depuración en un entono simulado o real y embebido. Posibilita realizar todas las verificaciones antes de construir el hardware. Este programa puede conectarse al software de diseño electrónico llamado Proteus, a través de la interfaz emulador virtual serial [8], permitiendo así la simulación del microcontrolador. Cabe aclarar que estos programas son de licenciamiento pago.

2°) La etapa correspondiente al Diseño se centra en cómo se realiza la integración de las partes del SE y como obtener datos precisos de los sensores, ya sea por calibración o filtrado. Así como la funcionalidad que debe realizar el microcontrolador con esa información (sobre cómo el sistema interactúa con su entorno). En esta etapa se plantean las técnicas que debe emplearse para cumplir con requisitos específicos sobre el SE. Estas técnicas pueden requerir atender a varios sensores simultáneamente, como así también enfocar el diseño para lograr un menor consumo energético, para que funcione con baterías. Otro requisito, que también se diseña, tiene que ver con la forma de realizar cálculos complejos en el propio SE. Las herramientas que se utilizan en esta etapa son provistas por el fabricante de SE, ya que permite programarlo, muchas de ellas se basan en la interfaz de programación Eclipse. Para arquitecturas Arduino: Eclipse C++, para procesadores ARM: posee un conjunto de herramientas con licenciamiento pago *KIEL*® y la versión libre *System Workbench*, entre otros.

3°) La etapa de Análisis. Es utilizada para garantizar que el SE resultante, cumpla con la funcionalidad correcta, como con los requisitos planteados en las etapas de Modelado y Diseño. Debe incluir especificaciones, tales como seguridad, consumo, fiabilidad, entre otras, que deben ser definidas como propiedades en forma precisas, expresadas técnicamente, para evitar ambigüedad, facilitando la recolección y comparación de resultados. El análisis debe ser realizado cuantitativamente para lograr validar las definiciones de características o alcance del SE. En consecuencia existen técnicas para analizar tiempos dinámicamente [6], tales como: **Circuito Emulador:** Es un dispositivo de propósito especial que se comporta como un procesador particular, pero con mejores capacidades para depuración e inspección del programa que ejecuta. **Osciloscopios:** Son herramientas destinadas a medir con exactitud atributos de las señales eléctricas que recorren circuitos, por lo que pueden ser utilizados para medir tiempos de respuesta de las entradas y salidas de SE. **Analizadores lógicos:** Son dispositivos que permiten leer la memoria o el bus de direccionamiento y analizar las instrucciones que son leídas. **Dispositivo de rastreo:** Los puertos de rastreo son incorporados a los microcontroladores actuales. Estos permiten depurar el programa en ejecución e inspeccionar sus datos. **Temporizadores en el código:** Se puede agregar dentro del programa, funciones específicas que miden el intervalo de tiempo transcurrido en ejecutar partes del código. **Contadores de procesador:** Procesadores como X86, PowerPC o MIPs poseen contadores

dentro del procesador, que se incrementan a medida que ejecutan instrucciones. **Herramientas de inspección de desempeño:** Obtienen el rendimiento del hardware, a través de código agregado por el compilador. **Simuladores:** Programa que trata de reproducir la conducta de un microcontrolador real [2], a los fines de validar la funcionalidad, medir tiempos y atributos de la ejecución. **Emulador software:** Programa que permite ejecutar el programa SE como si ejecutara sobre la arquitectura real. Son de fácil acceso en las primeras etapas del desarrollo, aun sin contar con el hardware real.

3. RESULTADOS OBTENIDOS/ESPERADOS

Con el propósito de armar el entorno de trabajo de SE sé planifica el proyecto dividiéndolo en dos etapas. En la primera etapa se realizarán análisis sobre sistemas embebidos disponibles en el mercado, además de los tipos de RTOS que se pueden ejecutar sobre los mismos. Para lograr comparar sus características, ventajas y desventajas, se desarrollarán algoritmos y casos de prueba, que se almacenarán en repositorios para su utilización por parte de la comunidad. Estos se comparará sobre SE reales y virtuales, que serán configurados en las distintas pruebas. Se realizarán las compilaciones en el sistema embebido y se ejecutarán automáticamente con alguna herramienta de optimización de desarrollo, que serán evaluadas. Para el segundo año del proyecto se concentrara en el desarrollo de las pruebas automatizadas con herramientas disponibles con licencia GPL o similar. Estas herramientas están destinadas a mejorar el desarrollo de sistemas embebidos de tiempo real, depurando o midiendo rendimiento. Luego se realizarán análisis y pruebas sobre las optimizaciones que se puedan utilizar. De esta manera con la integración de las herramientas y las pruebas automáticas se podrá verificar y comparar el impacto de las optimizaciones sobre distintos SE. Permitiendo la elección del SE que se adecue a las necesidades.

4. FORMACIÓN DE RECURSOS HUMANOS

La presente línea de investigación dentro del departamento de Ingeniería e Investigaciones Tecnológicas forma parte del trabajo que uno de los investigadores se encuentra realizando para su maestría. Completan el grupo de investigación dos de docentes de categoría V y dos ingenieros en formación de investigador.

5. BIBLIOGRAFÍA

[1] X. Fan, Real-Time Embedded Systems: Design Principles and Engineering Practices, 1st ed., Newton, MA: Newnes, 2015.

[2] M. Barr y A. Massa, Programming Embedded Systems: With C and GNU Development Tools, 2 ed., Mumbai, India: O'Reilly Media, Inc., 2006.

[3] F. Eric, "Semantics of Optimization for Real Time Intelligent Embedded Systems (SORTIES)," National Science Foundation, Alexandria, Virginia 22314, USA , 2018.

[4] M. Bertogna, «High-Performance Real-time Architectures for Low-Power Embedded Systems,» 31 12 2018. [En línea]. Available: https://cordis.europa.eu/project/rcn/199161/factsheet/es. [Último acceso: 09 2019].

[5] K. C. Wang, Embedded and Real-Time Operating Systems, 1st ed., vol. 130, Springer Publishing Company, Incorporated, 2017, pp. 24, 36.

[6] I. Lee, J. Y.-T. Leung y S. H. Son, Handbook of Real-Time and Embedded Systems, 1st ed., Chapman & Hall/CRC, 2007.

[7] E. A. Lee y S. A. Seshia, Introduction to Embedded Systems: A Cyber-Physical Systems Approach, 2nd ed., The MIT Press, 2016.

[8] R. Singh, A. Gehlot, B. Singh y S. Choidhury, Arduino-Based Embedded Systems: Interfacing, Simulation, and LabVIEW GUI, CRC Press, 2017.

5. WICC 2020 – Certificados

Se certifica que **WALDO VALIENTE (UNLAM)** ha participado en calidad de autor del artículo **ENTORNO DE INTEGRACIÓN CONTINUA PARA VALIDACIÓN DE SISTEMAS EMBEBIDOS DE TIEMPO REAL (12830 - ARSO)** aceptado en el XXII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2020, organizado por la Universidad Nacional de la Patagonia Austral - Junio 2020.

CERTIFICADO Nº 1264 /2020 /UNPA

Se certifica que **ESTEBAN CARNUCCIO (UNLAM)** ha participado en calidad de autor del artículo **ENTORNO DE INTEGRACIÓN CONTINUA PARA VALIDACIÓN DE SISTEMAS EMBEBIDOS DE TIEMPO REAL (12830 - ARSO)** aceptado en el XXII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2020, organizado por la Universidad Nacional de la Patagonia Austral - Junio 2020.

CERTIFICADO Nº 1265 /2020 /UNPA

Se certifica que **MARIANO VOLKER (UNLAM)** ha participado en calidad de autor del artículo **ENTORNO DE INTEGRACIÓN CONTINUA PARA VALIDACIÓN DE SISTEMAS EMBEBIDOS DE TIEMPO REAL (12830 - ARSO)** aceptado en el XXII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2020, organizado por la Universidad Nacional de la Patagonia Austral - Junio 2020.

Lic. Patricia Pesado
Coordinadora
RedUNCi

Ing. Hugo Santos ROJAS
Rector
UNPA

CERTIFICADO Nº 1266 /2020 /UNPA

Se certifica que **GRACIELA DE LUCA (UNLAM)** ha participado en calidad de autor del artículo **ENTORNO DE INTEGRACIÓN CONTINUA PARA VALIDACIÓN DE SISTEMAS EMBEBIDOS DE TIEMPO REAL (12830 - ARSO)** aceptado en el XXII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2020, organizado por la Universidad Nacional de la Patagonia Austral - Junio 2020.

Lic. Patricia Pesado
Coordinadora
RedUNCi

Ing. Hugo Santos ROJAS
Rector
UNPA

CERTIFICADO Nº 1267 /2020 /UNPA

Se certifica que **RAÚL VILLCA (UNLAM)** ha participado en calidad de autor del artículo **ENTORNO DE INTEGRACIÓN CONTINUA PARA VALIDACIÓN DE SISTEMAS EMBEBIDOS DE TIEMPO REAL (12830 - ARSO)** aceptado en el XXII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2020, organizado por la Universidad Nacional de la Patagonia Austral - Junio 2020.

Lic. Patricia Pesado
Coordinadora
RedUNCi

Ing. Hugo Santos ROJAS
Rector
UNPA

CERTIFICADO Nº 1268 /2020 /UNPA

Se certifica que **MATÍAS ADAGIO (UNLAM)** ha participado en calidad de autor del artículo **ENTORNO DE INTEGRACIÓN CONTINUA PARA VALIDACIÓN DE SISTEMAS EMBEBIDOS DE TIEMPO REAL (12830 - ARSO)** aceptado en el XXII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2020, organizado por la Universidad Nacional de la Patagonia Austral - Junio 2020.

Lic. Patricia Pesado
Coordinadora
RedUNCi

Ing. Hugo Santos ROJAS
Rector
UNPA

CERTIFICADO Nº 1269 /2020 /UNPA

6. CACIC 2020 – Artículo

Análisis de rendimiento y consumo para sistema embebido con requisitos de tiempo explícitos

Esteban Carnuccio, Graciela De Luca, Waldo Valiente, Mariano Volker, Raúl Villca, Matías Adagio

Universidad Nacional de La Matanza,
Departamento de Ingeniería e Investigaciones Tecnológicas,
Florencio Varela 1903 - San Justo, Argentina

{ecarnuccio, gdeluca, wvaliente, mvolker}@unlam.edu.ar
{raul.villcasd, mati.adagio}@gmail.com
www.unlam.edu.ar

Resumen. El propósito de este trabajo consiste en el estudio y análisis del rendimiento de un sistema embebido que emplea el microcontrolador STM32F103C8T6, con el objetivo de reducir el consumo de energía originado desde una batería. Para ello se basó en patrones de diseño de máquinas de estados, que permiten mejorar los tiempos de ejecución y respuesta del sistema. Esto también se logra a través de la realización de distintas ejecuciones variando la velocidad del microcontrolador, al modificar la frecuencia del reloj. El desarrollo ejecuta módulos que poseen características de tiempo real, debido a que fueron diseñados para que un sistema de monitoreo de personas que detecta caídas, realiza geolocalización y ofrece control de pánico, entre otras funcionalidades más. Por ese motivo debe cumplir con un tiempo límite de ejecución estricto. Para garantizar estos requisitos, se aplicaron distintas técnicas de medición del rendimiento utilizando pulsos de reloj y señales de temporizador.

Palabras Clave. Análisis de Rendimiento, Consumo de Energía, STM32, Tiempo Real

1 Introducción

Los sistemas embebidos (SE) actuales requieren cada vez más conexiones, tanto para sistemas externos, como también sensores y actuadores. Además ejecutan algoritmos cada vez más complejos, cuya completitud se ve aumentada si se le suma como requerimiento no funcional, la premisa del bajo consumo energético. Esto se debe a que por lo general su fuente de energía puede ser una batería. Por ese motivo se planteó aplicar patrones de diseño de máquina de estado, que proporciona un enfoque práctico para atender a distintos estímulos externos con un único hilo de ejecución [11]. De esta forma, además de tener impacto sobre el rendimiento y los tiempos de respuesta [5], los patrones de diseño ayudan en la mantenibilidad del código y seguridad del dispositivo. En este sentido la presente investigación se basa en algoritmos de tiempo

real, donde se tiene requisitos estrictos sobre los tiempos de respuesta. Por ejemplo, para que funcione correctamente un algoritmo que se encarga de detectar caídas, mínimamente debe leer mediciones del acelerómetro cada 24 milisegundos. De lo contrario, no se conseguirá detectar los picos de aceleración característicos que generan las caídas, el algoritmo de detección empleado fue analizado en profundidad en [7]. Además se necesitaba que la batería tenga una duración de su carga activa mayor a 24hs, para poder así asegurar su disponibilidad y comodidad de uso del dispositivo por parte del usuario monitoreado. Los requisitos no funcionales antes comentados, se completaron como parte de un sistema de tiempo real. En el cuál, la tarea crítica debe culminar correctamente antes de cumplirse su tiempo límite de respuesta. Esto es como consecuencia de que el ciclo completo de funcionamiento incluye el tiempo desde que solicita los datos, su normalización y finalmente su procesamiento, junto con las actividades necesarias a realizar sobre los otros sensores y actuadores.

En el análisis se optó por una arquitectura que no utiliza Sistema Operativo, de modo que el programa se ejecuta siempre sobre un único hilo de ejecución. Cabe aclarar que se podría haber utilizado bibliotecas que dan funcionalidades de múltiples tareas concurrentes, como *FreeRTOS* [12]. Sin embargo se determinó conveniente emplear patrones de diseño de máquina de estado, dado que la opción de utilizar bibliotecas externas aumenta el tamaño del binario ejecutable, debido a que agregan muchas líneas de código adicionales. De esta forma terminan consumiendo los recursos acotados del espacio disponible de memoria, dado que el embebido está limitado a solo 64 Kbyte para poder ejecutar el programa.

Por consiguiente en este documento, a los fines de garantizar el cumplimiento crítico de los requisitos de tiempo de ejecución mencionados, se detalla el análisis de rendimiento del sistema. De forma tal, que permita ajustar las configuraciones de la arquitectura para lograr las reducciones del consumo energético y validar los tiempos de ejecución en los algoritmos desarrollados.

2 Desarrollo

2.1 Máquina de estados

Los algoritmos utilizados basan su funcionamiento en un conjunto de máquinas de estados, que se encuentran interconectadas entre sí. Se planifico usar esta metodología, ya que al ejecutar sobre un único procesador, es indispensable que el sistema no caiga en esperas activas sobre los distintos componentes de software, controladores y algoritmos. Por eso se planteó como objetivo, que el sistema pueda leer correctamente los valores obtenidos por el acelerómetro, luego ejecute satisfactoriamente el algoritmo de caídas, lea las coordenadas de ubicación del GPS[1] y envíe notificaciones vía SMS[2] y por mensajes REST[3] al servidor sin retrasos. Para asegurar que se pueda satisfacer dicho comportamiento, se ha realizado un estudio exhaustivo del rendimien-

[1] GPS: *Global Positioning System*, Sistema de posicionamiento global.
[2] SMS: *Short Message Service*, Servicio de mensaje corto de la telefonía móvil.
[3] REST: *REpresentational State Transfer*, Interfaz de servicios de Internet.

to global del sistema. A continuación se expone gráficamente el diagrama del sistema (Fig. 1), en donde se detalla la máquina de estados central, llamado *Controlador_Principal*.

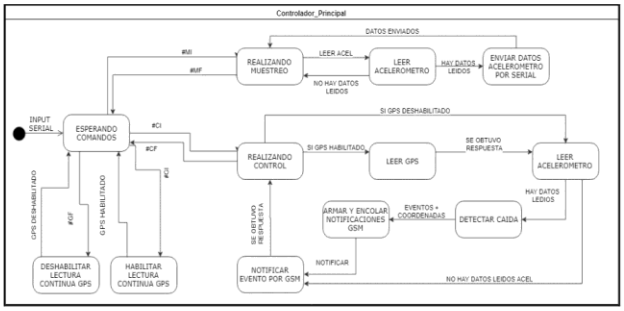


Fig. 1. Máquinas de estados del Controlador Principal.

El funcionamiento del SE es controlado por comandos a través de *bluetooth*[4], y dependiendo de la instrucción recibida ejecutará determinadas operaciones. La función principal, comienza cuando se recibe el comando "#CI", la cual iniciará la secuencia de ejecución de estados encargados de realizar el control principal del dispositivo embebido. En este punto es importante mencionar, que al ejecutar determinados estados de la (Fig. 1), se produce la invocación a las funcionalidades que internamente contienen otras máquinas de estados diferentes a la figura anterior. En consecuencia, al iniciar el ciclo *Realizando Control,* que se encarga de monitorear a la persona que lleva el dispositivo, primeramente se comprueba si se encuentra habilitada la lectura continua del GPS. Si esto es verdadero, se ejecuta el estado *LeerGPS, que es* la responsable de obtener las coordenadas de ubicación del SE. *L*uego pasa al estado *LeerAcelerómetro,* responsable de leer los valores del acelerómetro a través de su propia máquina de estados, que se encarga de estabilizar y normalizar las mediciones. Posteriormente se continúa con el estado *DetectarCaidas*. En él, se ejecuta el algoritmo responsable de detectar las caídas, que se encuentra analizado en detalle en [4]. Finalmente, en caso de detectase una caída, se cambia al estado *NotificarEventoPorGSM,* con la finalidad de enviar mensajes al servidor. El cuál distribuye la notificación a los usuarios responsables, informándoles de los eventos que hayan ocurrido en el sistema. Estas notificaciones se realizan a través de mensajes SMS y protocolo REST.

[4] Bluetooth: Protocolo inalámbrico de comunicación.

Es importante mencionar que en cada una de las etapas mencionadas, poseen su propia máquina de estados para lograr realizan tareas cortas y asincrónicas. De este modo no se generan esperas activas que bloqueen el flujo de ejecución del programa.

2.2 Configurar la frecuencia del reloj del microcontrolador

El SE se construyó utilizando el microcontrolador STM32F103C8T6. Una de las ventajas que tiene, es poder configurar la frecuencia de oscilación de su reloj principal (HCLK), ver (Fig. 2). De esta manera se puede configurar el sistema para que este componente funcione a distintas a frecuencias, desde 72 MHZ como máximo a 8 MHz de mínimo [1]. En este contexto se realizaron distintas pruebas modificando HCLK, para luego medir el desempeño, donde se analizaron las variaciones de tiempo en los algoritmos detallados anteriormente. De esta forma se llevaron a cabo mediciones de tiempo en distintos puntos específicos de esos algoritmos, junto con las evaluaciones de la variación del consumo energético. Por ende se realizaron configuraciones para que HCLK funcione con frecuencias de 8MHZ, 16MHZ y 72 MHZ. Como caso particular para la frecuencia de 8 MHZ, se seleccionó al oscilador interno HSI como origen del pulso. Mientras que en los casos de 16 MHZ y 72 MHZ, se utilizó el oscilador externo HSE.

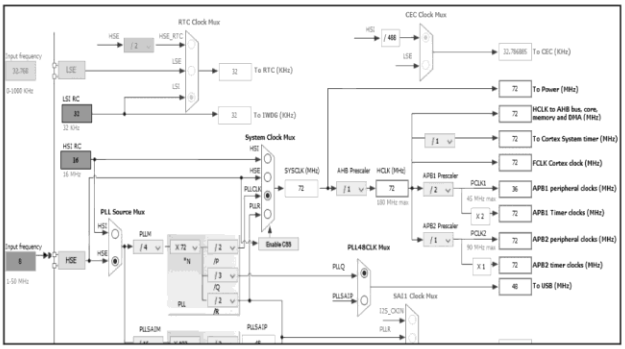


Fig. 2. Ventana de configuración frecuencia de microcontrolador SMT32.

En las pruebas se observó que con distintos valores del HCLK, se afectan indirectamente a la comunicación con los sensores y actuadores externos que se encuentren conectados. Esto se debe a que los periféricos trabajan a distinta velocidad que el microcontrolador, por eso existen dos buses independiente de periféricos (APB1 y APB2)[5], que realizan la comunicación con dispositivos que posean frecuencias diferentes a las del HCLK. No obstante estos son dependientes del mismo, debido a que

[5] ABP1 y ABP2: *Advanced Peripherals Bus 1 and 2: Bus dedicado a periféricos 1 y 2.*

se configuran utilizando divisores de frecuencia con respecto al valor del HCLK. En este sentido, al realizar las pruebas a distintas frecuencias del HCLK, el funcionamiento de los periféricos conectados al sistema se vio afectado. Por ese motivo al establecer el HCLK a 72 MHz, para un correcto funcionamiento de bus I2C[6] (que utiliza el acelerómetro), se configuró a una frecuencia de 400 KHZ. Mientras que al utilizar el HCLK a 8 MHZ y 16 MHZ, la frecuencia del bus I2C debió ser disminuida a los 100 KHZ.

2.3 Metodologías de medición de tiempo en un sistema embebido

En el análisis de requisitos no funcionales, el tiempo de respuesta para un SE es una tarea compleja. Esto es como consecuencia de que para obtener información sobre el rendimiento de la ejecución, se deben realizar las lecturas de sensores dentro del propio SE, almacenarlas y posteriormente analizarlas en una computadora fuera del SE. Por ende, para realizar dichas tareas, se pueden utilizar herramientas que recolectan la información en forma externa o también pueden realizarse manualmente mediante lecturas internas.

2.3.1 Análisis comparativo de herramientas que miden desempeño

En esta investigación se analizaron las herramientas que están disponibles para la familia de microcontroladores STM32 [2]. Por ende, para lograr realizar mediciones de desempeño del sistema, básicamente se pueden llevar a cabo empleando dos aplicaciones. Ambas poseen licenciamiento comercial, debido a que otorgan la posibilidad de uso evaluativo por un breve periodo de tiempo, pero con funcionalidades limitadas. Estas herramientas se llaman: *IAR™ EWARM* y *Keil® MDK-ARM µVISION*. En las especificaciones de la herramienta *µVISION* de *Keil®*, se observó que al ejecutar el código con distintas velocidades de reloj, se pueden obtener lecturas estadísticas incorrectas sobre el rendimiento de la ejecución [3]. También existe otro tipo de herramienta, con similar tipo de licenciamiento. Este es el caso de *Tracealyzer* de *Percepio* [4], que se integra al programa que ejecuta en el SE como una biblioteca. Sin embargo, esta se encuentra especialmente preparada para Sistemas Operativos de Tiempo Real (RTOS). En este tipo de arquitectura, se puede ejecutar el S.O *FreeRTOS*, ya que es totalmente compatible con dicha herramienta. No obstante se descartó su utilización, dado que consume recursos y acota el limitado espacio disponible de memoria. Por estos motivos se decidió realizar las mediciones manualmente en forma interna.

2.3.2 Lectura de desempeño interna

Existen dos técnicas para tomar mediciones de tiempo, desde el código fuente del propio programa a ejecutar. La primera de ellas utiliza el temporizador de cuenta descendiente *Systick*. Este funciona mediante interrupciones, que se activan cuando el

[6] I2C: *Inter Integrated Circuits*, Circuito Inter-Integrado que funciona como bus serie topología maestro-esclavo.

valor de su registro contador interno llega al valor cero. De esta manera se puede utilizar *Systick* para que genere una interrupción cada un determinado tiempo, por ejemplo cada un milisegundo. Esto permite que se pueda utilizar el temporizador para medir el tiempo de ejecución, debido a que se registra la interrupción desde que se inicia el sistema, de forma tal que sea utilizado como un temporizador dedicado. Así mismo una de las características del modelo empleado, es que posee 24 bits de representación para reinicio automático. Además puede enmascarar la interrupción cuando el contador descendiente llega a cero y la fuente de reloj es programable [13]. Por otra parte, la segunda forma de medición de tiempo, es a través de componentes internos. Para ello se debe utilizar el registro CYCCNT[7], que es un contador de ciclos del reloj del CPU, que forma parte de la unidad de depuración DWT[8], proporcionada por la arquitectura STM32. Esta es una técnica que es utilizada en *profiling,* desarrollada por *Serj Bashlayev* [8]. La cual permite generar marcas de tiempo a través del valor inicial del CYCCNT, que contenga en un momento dado. De forma tal, que después se contrarresta el valor que posee ese registro en distintos instantes de tiempo. En consecuencia para realizar ese cálculo se debe tener en cuenta la frecuencia del reloj principal, en base a la cantidad de pulsos por microsegundos que este genera.

2.4 Mediciones de tiempos realizadas durante la ejecución del sistema

Debido a las limitaciones mencionadas sobre las herramientas de desempeño externas, en esta investigación se han utilizado los componentes internos del microcontrolador para medir el desempeño del sistema. Para ello se han utilizado las dos técnicas antes mencionadas, la desarrollada por *Serj Bashlayev* y en determinadas ocasiones se ha empleado el método de *Systick*. En ese sentido se han realizado mediciones en diferentes puntos críticos del SE, con el objetivo de analizar su rendimiento y comprobar que se puedan cumplir los límites de tiempo. A continuación se enumeran los puntos claves en donde se han realizado las mediciones de tiempo pertinentes, para las distintas velocidades de reloj de sistema.

- Lectura del tiempo demandante desde que se solicitan los datos al acelerómetro, hasta el momento que son leídos, dentro de *LeerAcelerómetro*.
- Interiormente en la máquina de estado de *LeerGPS*, se midió el tiempo que le demanda al sistema para empezar a leer una línea de datos del GPS, a través de un puerto serial, hasta que se determina si esa línea contiene un formato NMEA[9] válido. Como así también, cuanto se tarda para la detección del NMEA completo.
- Dentro del *NotificarEventoPorGSM*, se obtuvo el tiempo que necesita el algoritmo para ejecutar la secuencia de comandos AT. La cual es necesaria para poder enviar un SMS o ya sea un mensaje del tipo REST.

---

[7] CYCCNT: *CYcle Clock CouNTer*, Contador de ciclos de reloj de CPU.
[8] DWT: *Debug Watchpoints data Tracing*, Unidad de depuración de datos.
[9] NMEA: *National Marine Electronics Association*, es una especificación instrumentos marítimos y receptores GPS.

- La cuantificación de tiempo que requiere la máquina de estados del Controlador Principal en ejecutar un ciclo completo, en donde se obtenga las lectura de acelerómetro, las coordenadas de GPS y se envíe un mensaje por GSM.

Estos puntos fueron seleccionados debido a que en ellos se pueden producir demoras en su ejecución.

3 Análisis de resultados

En las tabla 1 se detallan los resultados de los tiempos obtenidos, luego de la realización de las mediciones en los puntos críticos previamente mencionados. En ella se puede observar la cantidad de mediciones que fueron efectuadas por cada uno de los eventos descriptos, como así también los valores máximos y mínimos de tiempo registrados en esas mediciones. Los valores mostrados dependen de la configuración del reloj principal seleccionada en ese momento.

Eventos	CLOCK 72 MHZ			CLOCK 16 MHZ			CLOCK 8 MHZ		
	Número Mediciones	Valor max [us]	Valor min[us]	Número Mediciones	Valor max [us]	Valor min[us]	Número Mediciones	Valor max [us]	Valor min[us]
Lectura Acelerómetro	3881	1802	1108	2420	4402	2857	1004	6840	3995
Lectura GPS Línea NMEA	65	63996	54995	69	80009	75975	51	4675005	730004
Lectura GPS Línea Serial	1908	254	74	2735	984	203	192	4489	406
Lectura GPS carácter NMEA	482	103	1	650	312	7	1060	871	14
Envió SMS	5	5404263	4495155	3	4810253	4404262	4	4549081	4448007
Envió REST	8	15504805	8434725	4	8962651	8052403	6	16448783	8424796
Bucle principal	15791	782	1	13023	649	3	15717	272	6

Tabla 1. Tiempos de ejecución según frecuencia 72/16/8MHz del reloj principal.

En las tabla 1 se puede observar que al aumentar la frecuencia del reloj, para los distintos puntos críticos de los diferentes algoritmos, los tiempos de ejecución van disminuyendo. Esto ocurre en la mayoría de los casos a excepción de las pruebas en donde el GSM[10] realiza el envió de un SMS y mensajes tipo REST. Dado que los tiempos requeridos para ejecutar las secuencias de comandos AT[11] para dichas operaciones se mantienen constantes, por más que se modifique la velocidad del reloj principal. La razón de esta linealidad en los tiempos de ejecución de las operaciones GSM, se debe a que siempre se realizó la transferencia de datos por los puertos seriales con la misma velocidad en baudios, por lo que estos tiempos no se ven afectados

[10] GSM: *Groupe Spécial Mobile*, Sistema estándar digital de telefonía móvil.
[11] AT: *for getting the modem's ATtention*, obtener atención del modem.

significativamente. Otra excepción que se observó luego de las mediciones, es que el valor máximo de tiempo obtenido al ejecutar el bucle principal, aumenta a mayor frecuencia del reloj. Sin embargo, en los valores mínimos de tiempo registrados se produce el efecto contrario, por lo que en ese caso si se cumple la regla mencionada inicialmente. Luego del análisis se descubrió que la causa de este suceso se debe a que durante el bucle principal se ejecutan instrucciones que demoran instantes con tiempos máximos muy breves, generando así picos máximos de duración en microsegundos. No obstante el tiempo promedio de la duración del bucle principal se mantiene cercano a los valores mínimos registrados, en cada una de las configuraciones mostradas.

En la Tabla 2 se muestran las especificaciones del fabricante del STM32F103C8T6 [1], con respecto a la variación del consumo del microcontrolador para distintas frecuencia del reloj externo (f_{HCLK}). Como aclaración estas mediciones corresponden a la temperatura 25 ºC y un voltaje de entrada 3,3 V. Al contrarrestar esto valores con los consumos de las placas de desarrollo Arduino Uno [9] y Arduino NANO [10], el consumo resultante del STM32 es menor. Esto se debe a que las dos placas Arduino mencionadas trabajan a una frecuencia definida de 16 MHz, en donde su consumo energético es de 46 mA y 15 mA respectivamente. Ambos superiores a los 9,3 mA que consume esta arquitectura configurada para la misma frecuencia.

Parámetro	f_{HCLK} [MHz]	Consumo [mA]
Modo ejecución con los dispositivos conectados.	72	36
	48	24,2
	36	19
	24	12,9
	16	9,3
	8	5,5

Tabla 2. Consumo máximos según frecuencia del STM32F103C8

Utilizando la herramienta *STM32CubeMx*, se puede predecir cuanto será la duración de la batería. Esto se describe en la Tabla 3, en donde la primera fila demuestra cómo se puede prolongar la vida útil de la batería para bajas frecuencias del microcontrolador. Mientras que en la fila dos, se comparan cuando se agregan al cálculo del consumo los sensores y los mecanismos de comunicación utilizados. Para ello se configuraron los puertos *USART1*[12] para utilizar bluetooth, *USART2* para GPS y *USART3* para GSM, Además se utilizó el puerto *I2C* para la conexión con el sensor acelerómetro *MPU6050* [6]. Esta configuración produce un consumo adicional de 118 mA, el cual se ve reflejado en los consumos promedios que se pueden observar en la tabla. En la

[12] USART: *Universal Asynchronous Receiver-Transmitter*, Transmisor-Receptor Asíncrono Universal.

predicción se tomó en cuenta el uso constante de los dispositivos y se descartaron los picos de consumo, como así también los modos en ahorro de energía.

Eventos	CLOCK 72 MHZ		CLOCK 16 MHZ		CLOCK 8 MHZ	
	Duración Batería (hrs)	Consumo Promedio (mA)	Duración Batería (hrs)	Consumo Promedio (mA)	Duración Batería (hrs)	Consumo Promedio (mA)
Solo micro-controlador	68	31,94	258	8,5	427	5,15
Micro-controlador+ dispositivos I/O	14	149,94	17	127,5	18	124,15

Tabla 3. Duración estimada de la batería en horas.

4 Conclusiones

Se han logrado establecer varios puntos de comparación con el uso de técnicas de medición de tiempos y rendimiento del sistema embebido. Con respecto a las mediciones que se han llevado a cabo, se evaluaron herramientas y técnicas, a partir de ello se han elegido emplear dos métodos. El primero de ellos por contadores de ciclos, que permite medir intervalos de tiempo en determinadas secciones especificas del sistema. Mientras que el segundo método empleado, *Systick*, se ha utilizado para ejecutar eventos de mediciones luego de un determinado tiempo. Por un lado se constató que al hacer uso de máquinas de estados en toda la solución, permite atender simultáneamente a varios dispositivos, dejando ciclos de reloj del microcontrolador para atender con prioridad a los sensores más críticos. También se pudo constatar que ajustando la velocidad de la arquitectura utilizada se permite minimizar el consumo energético. Además se verifica que si bien varía el tiempo de respuesta de las funciones críticas, se logra conservar el límite de tiempo requerido.

En base a las mediciones realizadas, se considera que la solución adecuada para implementar en el sistema consiste en configurar la velocidad del reloj principal para que trabaje a 16 MHZ. Esta elección de configuración permite un adecuado ahorro energético, inclusive menor que la arquitectura Arduino, con tiempos de respuestas acorde a los que son requeridos en la solución. En contraste, para frecuencias mayores a la elegida, se producirá mayor consumo energético. Mientras que por debajo de esta velocidad podría ocurrir que los algoritmos necesarios, en conjunto, no puedan cumplir con sus requerimientos de funcionamiento.

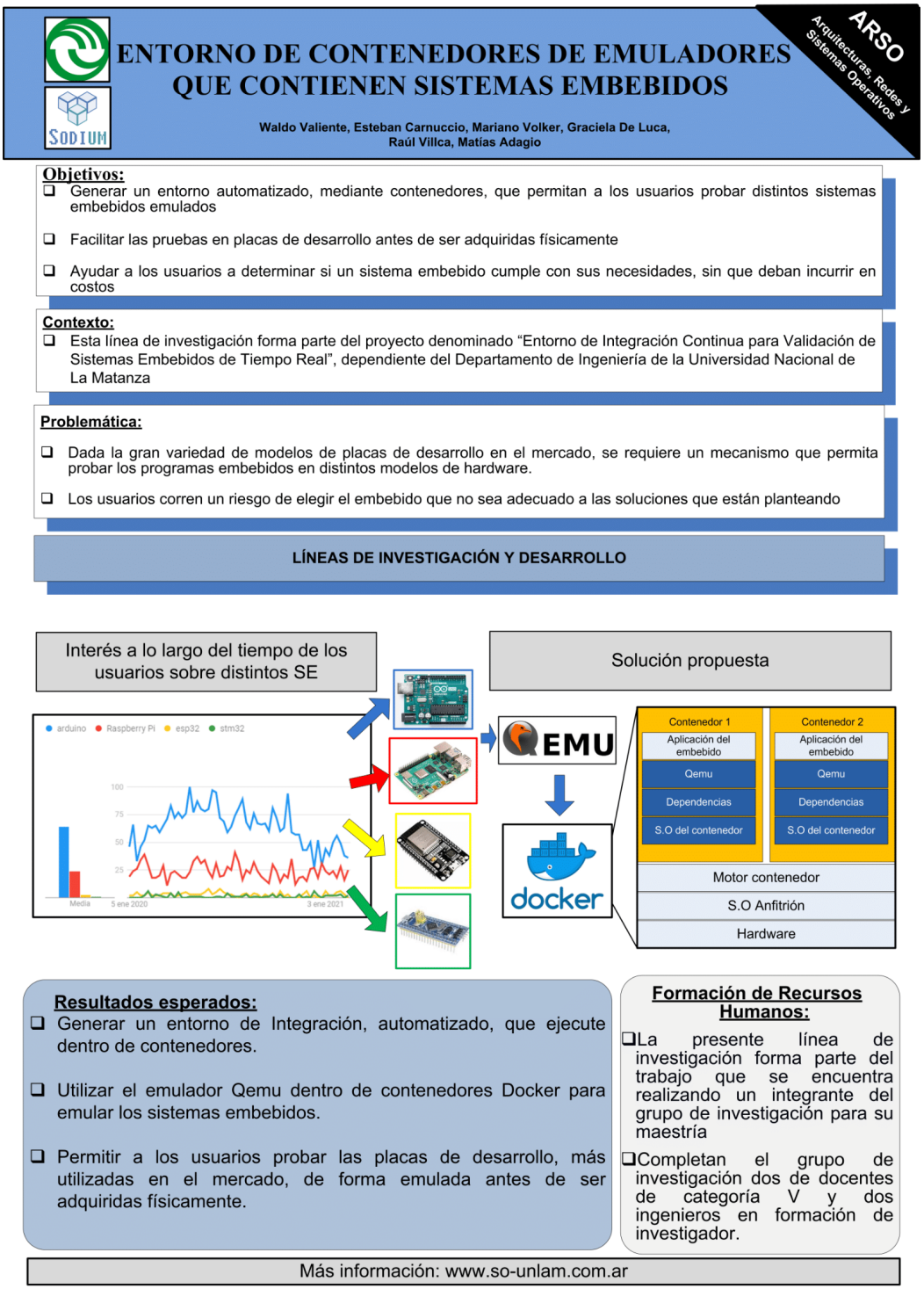
En futuros trabajos se plantea la posibilidad de armar un entorno que permita estos tipos de pruebas generales con diferentes sistemas embebidos, ya sean reales o virtuales. De esta formase podrá elegir el hardware más conveniente a utilizar de acuerdo a las necesidades de un proyecto que se desee llevar a cabo.

5 Referencias

1. STMicroelectronics. STM32F103x8 DataSheets.l. : STMicroelectronics, 2015. pág. 48, STM32F103x8, STM32F103xB - Medium-density performance line ARM®-based 32-bit MCU with 64.
2. STMicroelectronics. STM32 microcontroller debug toolbox. 2017.
3. Arm Limited. µVision User's Guide - Execution Profiler. [En línea] 2019. [Citado el: 24 de 07 de 2019.] http://www.keil.com/support/man/docs/uv4/uv4_db_dbg_execprofiler.htm.
4. Percepio. Percepio Tracealyzer. [En línea] https://percepio.com.
5. Douglass, Bruce. Software Engineering for Embedded Systems. Software Design Architecture and Patterns for Embedded Systems. s.l. : Elsevier Inc, 2013, 4.
6. InvenSense, Inc. MPU-6000 and MPU 6050 Product Specification Revision 3.4. . Sunnyvale : Inc.InvenSense, 2013.
7. De Luca, Graciela, y otros. Desarrollo de un prototipo detector de caídas utilizando la placa Intel Galileo Generación I y el sensor MPU6050. Anales del XXIII Congreso Argentino de Ciencias de la Computación. La Plata : s.n., Octubre 2017, págs. 954-963.
8. Bashlayev, Serj. https://github.com/Serj-Bashlayev. [En línea] 4 de Octubre de 2018.
9. Arduino. Arduino UNO - DataSheet. [En línea] 1998. https://www.arduino.cc/en/uploads/Tutorial/595datasheet.pdf.
10. Arduino. DataSheet Arduino NANO. [En línea] 2008. https://www.arduino.cc/en/uploads/Main/ArduinoNanoManual23.pdf.
11. Valiente, Waldo, y otros. Uso colaborativo del procesador en sistemas embebidos para múltiples interfaces. San Juan : WICC2019, 2019. 978-987-3619-27-4.
12. Amazon Web Services, Inc. FreeRTOS. [En línea] 2019. https://www.freertos.org/.

7. CACIC 2020 – Certificado

8. WICC 2021 – Poster



9. WICC 2021 – Artículo

Entorno de Contenedores de Emuladores que contienen Sistemas Embebidos

Waldo Valiente, Esteban Carnuccio, Mariano Volker, Graciela De Luca, Raúl Villca, Matías Adagio
Departamento de Ingeniería e Investigaciones Tecnológicas
Universidad Nacional de La Matanza
Dirección: Florencio Varela 1703 – CP 1754 – {wvaliente, ecarnuccio, mvolker, gdeluca}@unlam.edu.ar, {raul.villcasd, mati.adagio}@gmail.com

RESUMEN

Aunque la población no se percate, cada vez más, los sistemas embebidos forman parte de la vida cotidiana de las personas. Como consecuencia de dicho auge de la tecnología, surgieron en el mercado diferentes sistemas embebidos conformados por placas de desarrollo, que inicialmente se utilizan para prototipado. Estas son de fácil aprendizaje, lo que permitió que sean muy utilizadas en los últimos años. Por ese motivo para un usuario sin experiencia, resulta un desafío la elección de la placa correcta para sus proyectos. Esto se debe a que muchas veces se puede equivocar en dicha selección, ya que no cumpliría con sus expectativas. Por ende, en esta investigación se pretende minimizar esos riesgos, a través de la utilización de emuladores que permitan imitar el comportamiento de estos sistemas embebidos. Aprovechando estas características se procura generar un entorno de integración, empaquetado y automatizado mediante contenedores. Los cuales permitirán al usuario probar determinados programas rápidamente, en el hardware emulado del sistema embebido, sin necesidad de incurrir en costos. De forma tal de permitirle determinar si la placa elegida puede llegar a cumplir sus expectativas, sin necesitar adquirir el hardware físico.

Palabras clave: Sistemas Embebidos, Emuladores, Qemu, Contenedores, Docker.

CONTEXTO

Nuestra Línea de Investigación es parte del proyecto "*Entorno de integración continua para validación de sistemas embebidos de tiempo real"*, dependiente de la Unidad Académica del *Departamento de Ingeniería e Investigaciones Tecnológicas*, perteneciente al programa de Investigaciones CyTMA2 de la Universidad Nacional de La Matanza, el cual es formado por docentes e investigadores de la carrera de ingeniería en informática. Este proyecto es continuación de los trabajos que viene realizando el grupo de investigación en sistemas operativos, computación de alto rendimiento y en el área de Internet de las cosas.

1. INTRODUCCIÓN

En los últimos años los sistemas embebidos han tenido un crecimiento exponencial, debido a que son utilizados en la vida cotidiana de las personas con mayor frecuencia. Este gran auge se debe principalmente a sus características que

son su miniaturización, su menor consumo energético, su gran capacidad de procesamiento y por la interconexión entre ellos. Gracias a su evolución, han surgido nuevos conceptos tecnológicos, los cuales hace décadas atrás eran impensados. Tales como dispositivos vestibles, domótica, telefonía móvil, internet de las cosas, ciudades inteligentes, industria 4.0, telemedicina, entre otros. A su vez algunos de ellos se pueden combinar con otras nuevas herramientas de procesamientos de datos, tales como Big Data o Machine Learning. Esto se debe a que existen sistemas embebidos que recompilan gran cantidad de información, para obtener conocimiento de su entorno y así aprender de los datos en forma automática.

Durante la evolución de los sistemas embebidos, hubo un punto importante en su línea de tiempo. Massimo Banzi en el año 2005 inicia, junto a sus alumnos, el proyecto Arduino en el instituto IVRAE en Italia. Este tenía como premisa permitirles aprender a trabajar, en muy poco tiempo y de forma económica, con microcontroladores a sus estudiantes. El proyecto se fundamentó en el diseño, construcción y prototipado de hardware libre. Además de ser multiplataforma y de fácil aprendizaje a través del lenguaje Wiring. [1] Estas características permitieron que en poco tiempo las placas Arduino tuvieran una gran aceptación, de forma tal que actualmente existe una gran comunidad de usuarios que aportan sus conocimientos y constantemente van optimizando el proyecto. [2]

A partir del surgimiento de las placas de prototipado Arduino, emergieron otros sistemas embebidos que tomaron como base las premisas del proyecto de Massimo Banzi, ya que permiten el rápido aprendizaje de su uso e intentan ser de costo accesible para los usuarios. Actualmente existen una gran cantidad de familias de placas de desarrollo en el mercado que pretenden seguir dicho principio. De esta manera, junto con Arduino, las placas más importantes y buscadas por los usuarios para sus proyectos son las familias de Raspberry Pi, las familias de placas que poseen los microcontroladores STM32, las ESP8266 y su sucesor, que poseen el microprocesador ESP32. Siendo Arduino la más buscada entre todas las demás. Esta preponderancia se puede visualizar en el gráfico siguiente [3].

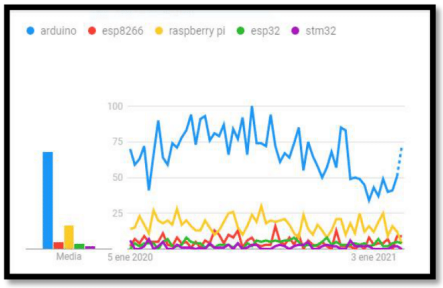


Fig. 1 interés a lo largo del tiempo de los usuarios de distintas placas de desarrollo en Argentina

En Argentina las placas de la familia Arduino es la más buscada y consultada en Internet. Mientras que el resto se mantienen muy por debajo de estas, en cantidad de búsquedas. Diferente es la situación en los países de América del Norte y Europa, en donde la cantidad de búsquedas están lideradas no solo por Arduino, sino también por la Raspberry Pi. Mientras que el resto se mantienen muy por debajo de ellas. Esta tendencia se debe a diversos factores. Por un lado, los STM32 son más potentes que las Arduino, pero a veces no presentan bibliotecas drivers desarrolladas para determinados dispositivos. Mientras que, para Arduino, los drivers son compatibles con la gran mayoría de los periféricos. A su vez los microcontroladores de Arduino, pueden continuar funcionando hasta que las baterías se agoten por completo. En cambio, los STM32

funcionarán hasta alcanzar el umbral de carga de energía definido por el fabricante [4]. Por su parte las Raspberry Pi, debido a su gran capacidad de procesamiento, posibilidad de ejecución de distintos Sistemas Operativos de escritorio y de conexión de periféricos y conectividad al exterior, se utilizan principalmente como Servidores, Gateway o central de procesamiento de datos e imágenes. No obstante, debido a su alto costo no suelen ser utilizadas para la lectura de sensores y la activación de actuadores, ya que se emplean como intermediarios. Por otro lado, comparando las placas Arduino con las ESP8266, desde el punto de vista de la comunicación, se puede apreciar que una de las características que presentan las placas tradicionales de Arduino, es su falta de conectividad al exterior. Para realizar dicha comunicación se deben utilizar módulos adicionales de comunicación de Wifi, Bluetooth, entre otros. Mientras que el microcontrolador ESP8266, trae incorporado un módulo de Wifi. Por su bajo precio lo hace ideal para proyectos de Internet de las Cosas. No obstante, no tiene una comunidad tan grande de usuarios, como los de las plataformas de Arduino. Además, presenta algunas limitaciones en cuanto a compatibilidad y uso de periféricos. Para resolver dichas falencias en 2016 Espressif, lanzó el microprocesador de doble núcleo ESP32 que amplía las capacidades de cómputo de su predecesor. Al mismo tiempo presenta un módulo WIFI y Bluetooth incorporado. Lo que permite un abanico de utilidades.

En las placas indicadas anteriormente, a excepción de la Raspberry Pi, la forma tradicional de programarlas es con el mecanismo de Bare Metal. Esto quiere decir que funcionan sin Sistema Operativo. No obstante, si el usuario lo desea, se puede instalar un Sistema Operativo, incluso de Tiempo Real, como por ejemplo FreeRTOS.

Estas placas fueron especialmente diseñadas para realizar prototipos de proyectos, pero en algunas situaciones también se los utilizan como producto final. Todas ellas siguen la premisa del proyecto Arduino, la cual consiste en que su uso sea de rápido aprendizaje, de forma tal que cualquier persona con poco conocimiento en electrónica pueda aprender a usarlas [5] [6] [7] . Muchas veces, al plantear un proyecto, un usuario común no puede saber que placa de desarrollo es adecuada para su trabajo. Para saber esto, el usuario necesita contar con el hardware físico para hacer las pruebas necesarias de funcionamiento, lo que incurre en un costo de inversión para adquirir la placa elegida. Muchas veces ocurre que el hardware seleccionado no es el adecuado para el proyecto, lo que deriva en un cambio de plataforma, conllevando así a un costo extra para la adquisición del nuevo producto, que se pueda conseguir en el mercado local. Por ese motivo, una posible alternativa para minimizar los riesgos de este impedimento, en este trabajo se investigan los emuladores para dichas placas de desarrollo.

2. LINEAS DE INVESTIGACIÓN Y DESARROLLO

Los emuladores permiten a los usuarios ejecutar en una computadora de escritorio programas de un sistema embebido, sin necesidad de contar con él físicamente. Esto se ejecuta dentro de un ambiente que imita su comportamiento [8]. A través de su utilización, una persona puede probar el sistema embebido en su computadora, antes de adquirir el hardware físico, y así verificar que le es de utilidad. Por consiguiente, teniendo en cuenta dichas características, la presente investigación se centra en construir un entorno emulado que ejecuta distintos sistemas embebidos. Todo esto funcionando con el esquema de contenedores de Docker. De manera tal, que permita realizar las pruebas de

rendimiento y las verificaciones de distintas optimizaciones de su ejecución, antes de llegar a adquirir el hardware del dispositivo.

Emulador que se utiliza en la investigación:

Existen en el mercado diferentes emuladores que imitan el comportamiento de distintos sistemas embebidos. El más utilizado es Proteus. Si bien es considerado un simulador de circuitos, también permite emular distintas placas de desarrollo, con sus actuadores y sensores de manera óptima. Este software puede emular placas de la familia Arduino, STM32, ESP8266 y Raspberry Pi. El punto negativo de este software es que su uso debe ser a través de licenciamiento pago. Por otro lado, se encuentra el simulador de Arduino Thinkercad, que es muy utilizado en ambientes educativos. La desventaja que presenta esta herramienta es que se limita únicamente a simular una placa de desarrollo Arduino UNO. Adicionalmente existe el emulador Qemu, el cual va a ser el utilizado en esta investigación. Este emulador es muy utilizado en el mercado. Entre sus principales ventajas [9], se destaca que permite trabajar con una gran cantidad de modelos de dispositivos. Además, permite detectar problemas lógicos en etapas tempranas del desarrollo, ya que brinda la depuración del dispositivo. Al mismo tiempo, presenta licenciamiento GPL. Así como también permite generar script de automatización de programas al poder ser ejecutadas desde línea de comandos. Esta característica resulta muy útil para esta investigación. De esta manera se pueden emular distintas familias de dispositivos Arduino, STM32, ESP32, ESP8266 y Raspberry Pi.

Emulador dentro de un contenedor:

Un contenedor de software es un ambiente aislado que permite compilar y ejecutar paquetes de software sin utilizar virtualización del hardware. Estos se diferencian de una máquina virtual, que realiza la completa virtualización del dispositivo físico y necesita ejecutarse sobre un Sistema Operativo que es redundante. Los contenedores se ejecutan sobre el S.O anfitrión y encima del motor de contenedores, que los emplean como base para poder funcionar. Estos contenedores a su vez pueden tener o no instalados un Sistema Operativo propio. Esta es la gran diferencia con las máquinas virtuales [10] [11].

En esta investigación se utiliza el sistema Docker, como motor de contenedores. De esta forma se está desarrollando un entorno de integración empaquetado y automatizado. Este motor va a generar contenedores de software, donde se ejecutará el emulador Qemu, que estará configurado para emular el hardware físico de las placas de desarrollo seleccionadas. El diseño de esta implementación se puede visualizar en la siguiente figura.

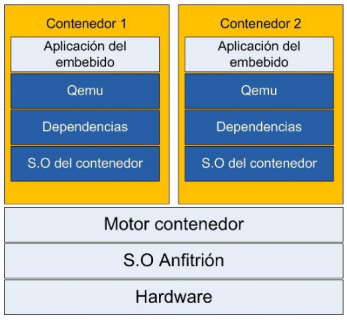


Fig. 2 Contenedor con el emulador de Sistemas embebidos

Como se puede observar en la figura, en la base de la pila se encuentra el hardware de la computadora. Sobre este funciona el Sistema Operativo Anfitrión y por encima del mismo el motor Docker. Dentro del contenedor se apilan las aplicaciones. En la parte inferior se encuentra instalado el Kernel mínimo del S.O Ubuntu. Luego se instalan las dependencias necesarias que necesita el emulador. Esto se debe a que Qemu necesita sus componentes para

poder ejecutarse. Posteriormente, en la capa siguiente se instala Qemu, ya preparado para que emule determinadas placas de desarrollo. Finalmente, en la capa superior se instala las aplicaciones del embebido, que el usuario desea ejecutar sobre la placa emulada en Qemu. De esta forma se pretende generar contenedores con imágenes de los sistemas embebidos, que permitan realizar pruebas en un entorno estandarizado, de manera tal que las aplicaciones del embebido puedan funcionar tanto en un sistema real como virtual.

3. RESULTADOS OBTENIDOS/ESPERADOS

En esta investigación se espera generar ambientes de trabajos automatizados, utilizando contenedores Docker que posean instalado y configurado el emulador Qemu, para poder utilizar distintos modelos de placas de desarrollo STM32, ESP32 y Arduino en forma virtual. De esta manera permitirá generar entornos de trabajos para automatizar pruebas de rendimiento y procesamiento simulado, de forma que los usuarios usen sus aplicaciones en el hardware físico adecuado de acuerdo con sus necesidades, incluso antes de adquirirlos.

4. FORMACIÓN DE RECURSOS HUMANOS

La presente línea de investigación dentro del departamento de Ingeniería e Investigaciones Tecnológicas forma parte del trabajo que uno de los investigadores se encuentra realizando para su maestría. Completan el grupo de investigación dos de docentes de categoría V y dos ingenieros en formación de investigador.

5. BIBLIOGRAFÍA

[1] J. Novillo-Vicuña, D. H. Rojas, B. M. Olivo, J. M. Ríos y O. C. Villavicencio, Arduino y el Internet de las cosas, Alicante: Editorial Area de Innovaciòn y Desarrollo, 2018.

[2] G. L. Nicolas Goilav, Arduino: Aprender a desarrollar para crear objetos inteligentes, Barcelona, España: Ediciones ENI, 2016.

[3] G. Trends. [En línea]. Available: https://trends.google.com/trends/explore?date=2020-01-01%202021-02-18&geo=AR&q=arduino,esp8266,raspberry%20pi,esp32,stm32.

[4] E. Carnuccio, G. De Luca, W. Valiente, M. Volker, R. Villca y M. Adagio, «Análisis de rendimiento y consumo para sistema embebido con requisitos de tiempo explícitos,» de *Libro de Actas del XXVI Congreso Argentino de Ciencias de la Computación*, San Justo, Universidad de La Matanza, 2020.

[5] J. Teel, How to Turn Your Arduino Project into a Sellable Product, Predictable Designs, 2016.

[6] M. Montero. [En línea]. Available: https://tecnoticias.net/2019/07/20/por-que-la-raspberry-pi-no-es-una-buena-opcion-para-productos-comerciales/#8230.

[7] D. Aranda, Electronica: Plataformas Arduino y Raspberri Pi, Buenos Aires: RedUsers, 2014.

[8] R. Milo-Sanchez, A. Fajardo-Moya y W. Rodriguez, «QEMU, una alternativa libre para la emulación de arquitecturas de [hardware,» de *V Taller de Software Libre*, 2014.

[9] A. Kotovsky, How to Develop Embedded Software Using The QEMU Machine Emulator, Apriorit Inc, 2019.

[10] J. M. Ortega Candel, DOCKER. Seguridad y monitorización en contenedores e imágenes, RC-Libros, 2019.

[11] Microsoft. [En línea]. Available: https://docs.microsoft.com/es-es/learn/modules/intro-to-docker-containers/3-how-docker-images-work.

10. WICC 2021 – Certificados

RedUNCI

WICC 2021 Chilecito La Rioja

UNdeC

Por medio del presente se certifica que:

VALIENTE, Waldo

ha participado en calidad de Autor del artículo Nro. 13651 **"Entorno de Contenedores de Emuladores que contienen Sistemas Embebidos"** aceptado en el XXIII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2021, organizado en modalidad virtual por la Universidad Nacional de Chilecito los días 15 y 16 de abril de 2021.

Chilecito, La Rioja, Argentina.

Lic. Patricia Pesado
Coordinadora
Red UNCI

Ing. Norberto Raúl Caminoa
Rector
UNIVERSIDAD NACIONAL de CHILECITO

RedUNCI

WICC 2021 Chilecito La Rioja

UNdeC

Por medio del presente se certifica que:

CARNUCCIO, Esteban

ha participado en calidad de Autor del artículo Nro. 13651 **"Entorno de Contenedores de Emuladores que contienen Sistemas Embebidos"** aceptado en el XXIII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2021, organizado en modalidad virtual por la Universidad Nacional de Chilecito los días 15 y 16 de abril de 2021.

Chilecito, La Rioja, Argentina.

Lic. Patricia Pesado
Coordinadora
Red UNCI

Ing. Norberto Raúl Caminoa
Rector
UNIVERSIDAD NACIONAL de CHILECITO

RedUNCI

WICC 2021 Chilecito La Rioja

UNdeC

Por medio del presente se certifica que:

VOLKER, Mariano

ha participado en calidad de Autor del artículo Nro. 13651 **"Entorno de Contenedores de Emuladores que contienen Sistemas Embebidos"** aceptado en el XXIII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2021, organizado en modalidad virtual por la Universidad Nacional de Chilecito los días 15 y 16 de abril de 2021.

Chilecito, La Rioja, Argentina.

Lic. Patricia Pesado
Coordinadora
Red UNCI

Ing. Norberto Raúl Caminoa
Rector
UNIVERSIDAD NACIONAL de CHILECITO

RedUNCI

WICC 2021 Chilecito La Rioja

UNdeC

Por medio del presente se certifica que:

DE LUCA, Graciela

ha participado en calidad de Autor del artículo Nro. 13651 **"Entorno de Contenedores de Emuladores que contienen Sistemas Embebidos"** aceptado en el XXIII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2021, organizado en modalidad virtual por la Universidad Nacional de Chilecito los días 15 y 16 de abril de 2021.

Chilecito, La Rioja, Argentina.

Lic. Patricia Pesado
Coordinadora
Red UNCI

Ing. Norberto Raúl Caminoa
Rector
UNIVERSIDAD NACIONAL de CHILECITO

RedUNCI

WICC 2021 Chilecito La Rioja

UNdeC

Por medio del presente se certifica que:

VILLCA, Raúl

ha participado en calidad de Autor del artículo Nro. 13651 **"Entorno de Contenedores de Emuladores que contienen Sistemas Embebidos"** aceptado en el XXIII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2021, organizado en modalidad virtual por la Universidad Nacional de Chilecito los días 15 y 16 de abril de 2021.

Chilecito, La Rioja, Argentina.

Lic. Patricia Pesado
Coordinadora
Red UNCI

Ing. Norberto Raúl Caminoa
Rector
UNIVERSIDAD NACIONAL de CHILECITO

RedUNCI

WICC 2021 Chilecito La Rioja

UNdeC

Por medio del presente se certifica que:

ADAGIO, Matías

ha participado en calidad de Autor del artículo Nro. 13651 **"Entorno de Contenedores de Emuladores que contienen Sistemas Embebidos"** aceptado en el XXIII WORKSHOP DE INVESTIGADORES EN CIENCIAS DE LA COMPUTACIÓN – WICC 2021, organizado en modalidad virtual por la Universidad Nacional de Chilecito los días 15 y 16 de abril de 2021.

Chilecito, La Rioja, Argentina.

Lic. Patricia Pesado
Coordinadora
Red UNCI

Ing. Norberto Raúl Caminoa
Rector
UNIVERSIDAD NACIONAL de CHILECITO

11. CACIC 2021 - Artículo

Entorno de contenedores con emuladores de sistemas embebidos STM32

Esteban Carnuccio[1], Waldo Valiente[1], Mariano Volker[1], Raúl Villca[1], Matías Adagio[1]

[1] Universidad Nacional de La Matanza,
Departamento de Ingeniería e Investigaciones Tecnológicas
Florencio Varela 1903 - San Justo, Argentina
{ecarnuccio, wvaliente, mvolker, rvillca, maadagio}@unlam.edu.ar
www.unlam.edu.ar

Resumen. Ante la gran importancia que han tenido los sistemas embebidos, en los últimos años, debido al auge de Internet de las Cosas. Resulta de vital importancia conocer y probar el hardware antes de adquirirlo, para saber si cumple las necesidades de un proyecto determinado. En ese contexto, esta investigación se centró en la creación de un entorno automatizado de emulación, que permita probar rápida y fácilmente determinadas placas de desarrollo, sin necesidad de adquirir el hardware físico. Con esa premisa, se desarrolló un entorno dentro de un contenedor Docker, que permite realizar la emulación de determinadas placas de la familia STM32 a través del programa Qemu, listo para funcionar. De esta forma se podrá realizar distintas pruebas, sin la necesidad de realizar una tediosa configuración e instalación de los componentes y las dependencias.

Palabras Clave. Docker, Emulación, Internet de Las Cosas, Qemu, Sistemas Embebidos, STM32.

1 Introducción

En los últimos años creció de forma exponencial la tecnología de Internet de las Cosas. El término IoT toma relevancia cuando se superó la cantidad de dispositivos conectados a internet, que el número de personas que existían en el mundo en ese momento. Según las proyecciones de [1], se estima que en el año 2025 habrá aproximadamente cien mil millones de sistemas embebidos conectados, que transmitirán los datos de sus sensores para ser procesados en servidores externos, a través de internet. En este sentido, el tiempo y los costos que incurren para configurar un entorno de trabajo de software encargado de construir el sistema embebido, se vuelve un factor importante. Ya que para determinar si la placa de desarrollo puede cumplir con las necesidades, es necesario adquirir la placa físicamente, con el agregado de la curva de aprendizaje de su utilización. Muchas veces se trabaja sobre un producto que, en una etapa avanzada de un proyecto, se descubre que no es la indicada para cumplir con sus requerimientos. Como consecuencia, se debe adquirir un nuevo sistema embebido, que pueda cumplir con sus objetivos, produciendo así atrasos y aumento en los costos.

Para tratar de minimizar estos riesgos existen simuladores y emuladores de sistemas embebidos, como *Thinkercad* y *Proteus*. Pero estos presentan ciertas limitaciones de uso, que dificultan su utilización en los distintos proyectos IT [2]. Por ese motivo en esta investigación se desarrolló un entorno de integración automatizado a través de contenedores Docker, que permiten ejecutar programas en distintas placas STM32 emuladas a través de Qemu. De forma tal que permita realizar rápidamente distintas pruebas de aprendizaje en el entorno y el sistema embebido. Estos ejemplos a su vez funcionan en el embebido real.

En este documento, inicialmente se describe brevemente la comparación con otros entornos. Posteriormente se brinda una introducción al funcionamiento del contenedor de Docker y su registro de contenedores llamado Docker Hub. Luego, se detalla la forma en que se implementó y configuró el entorno de trabajo utilizando Qemu, dentro de Docker para los sistemas embebidos seleccionados. Finalmente, se detalla los ejemplos de código que se pueden ejecutar dentro del entorno.

2 Trabajos relacionados

Esta investigación se sustentó en diferentes trabajos relacionados sobre el emulador de Qemu para diferentes sistemas embebidos. A continuación, se realiza un repaso de los principales. Existen desarrollos como [3], donde se presenta una extensión de Qemu para integrarlo con la aplicación Eclipse. Esta debe ser instalado manualmente por el usuario y es un poco complicada su configuración. Además, solo se explica el ejemplo de encendido y apagado del led de testeo de la placa, mejor conocido como "*Blinky Led*". Sin embargo, este no ofrece explicación de cómo se deben emular otros sensores, actuadores y componentes del embebido.

Por otro lado, existe el proyecto realizado por *Beckus* [4]. El cual presenta su propia versión adaptada del código fuente de Qemu, que a su vez fue modificada por otros autores, como se describen en [5]. Pero para lograr utilizarlos, es difícil hacerlos funcionar correctamente, ya que se debe hacer la instalación y compilación de forma manual. No obstante, el proyecto de *Beckus* ofrece la forma de crear un contenedor Docker, pero no se logró hacerlo funcionar.

En [6] se menciona el proyecto de *Pebble*, que es una adaptación de *Beckus*, también es difícil su configuración e instalación. Pero es un proyecto en proceso, que no registra cambios presentes de actualización.

Finalmente, en el registro de contenedores de Docker[7], se puede descargar una imagen que emplea el proyecto *Beckus*, pero este no permite ejecutar los ejemplos. Además, no posee la documentación detallada con las formas en que se deben ejecutar dichos ejemplos. Tampoco tiene habilitado el ingreso por protocolo *SSH* dentro del contenedor, para su utilización.

Para subsanar las falencias descritas de los proyectos antes expuestos, en este trabajo de investigación se detalla cómo se armó el entorno contenido en una imagen Docker, en el que se modificó y adaptó el proyecto de *Beckus*. Obteniendo así un entorno de emulación bien documentado y funcional. Para ello se describen los distintos cambios realizados y se comentan brevemente los distintos tutoriales explicativos, para la

emulación de los sensores y actuadores en ciertas placas de desarrollo de la familia de microcontroladores STM32. Todo esto se hizo de forma automatizada, para que se pueda probar el mismo programa, que funciona en forma física, dentro del contenedor en forma emulada.

3 Desarrollo

3.1 Docker

Como se comenta en [8], Docker es una de las herramientas más populares en estos días en el mundo de la tecnología (IT). Básicamente, hay dos corrientes principales en el paradigma de Docker. El Primero, la plataforma Docker es de código abierto, esto permite que se equipe continuamente con nuevas características y funcionalidades relevantes. Siendo aprovechada no solo por programadores, sino también por equipos operativos de IT. La segunda tendencia es la adopción sin precedentes de la tecnología de contenedores, que es complementada por varios proveedores de soluciones y servicios de IT en todo el mundo. Estos dos permiten una mayor simplicidad en el desarrollo de aplicaciones, gracias a la implementación automatizada y acelerada de contenedores Docker. Siendo ampliamente promocionada como la clave diferenciadora del éxito sin precedentes de este paradigma.

3.2 Docker Hub

Docker Hub[1] es un registro de contenedores mantenido por Docker Inc. En el repositorio se encuentran las principales imágenes oficiales, como por ejemplo *Busybox*, Sistemas operativos como Ubuntu o Windows, incluso programas ya empaquetados como Apache, Node.js o WordPress entre otras. También permite a cualquier usuario crear una cuenta en forma gratuita y subir sus propias imágenes [9]. La protección de seguridad brindada es muy simple. Solo los propietarios de las imágenes publicadas y los usuarios habilitados, pueden realizar cambios en ellas. Además, tiene un sistema de puntuación por estrellas, similar al utilizado en repositorio GitHub o incluso el que utiliza Android en sus aplicaciones. Esto permite en el caso de imágenes con similares funcionalidades, seleccionar a la imagen mejor posicionada.

El repositorio de Docker permite probar nuevas versiones de las aplicaciones publicadas, o buscar nuevas aplicaciones que sirvan para un propósito dado. Las imágenes de Docker son una forma fácil de experimentar sin interferir la configuración actual de la computadora, ni aprovisionar una máquina virtual y no tener que preocuparse por los pasos de instalación. Además, permite la centralización en un único canal, facilitando compartir públicamente la imagen entre diferentes integrantes [10].

3.3 Imagen de Docker Creada

Acorde a los objetivos antes mencionados, se creó una imagen Docker que ya dispone el entorno utilizable, evitando el proceso de instalación y configuración de las herramientas necesarias para su funcionamiento. De manera tal, que el desarrollador

[1] URL Docker Hub: http://hub.docker.com

pueda de forma rápida y sencilla emular sus programas para STM32, en un entorno a través de Qemu. Para esto, este trabajo se basó en el proyecto de *Beckus* [4], adaptándolo a los objetivos planteados. En este sentido, se modificó el código fuente de Qemu, para que pueda emular funcionalmente las placas *Stm32-p103*, *Stm32-Maple* y *Stm32-f103c8t6* (conocida como *BluePill*). Siendo publicado en un repositorio propio de Github, junto con los códigos modificados de los ejemplos de las placas antes mencionadas. Por esa razón, la siguiente es la dirección web de dicho repositorio.

https://github.com/soaunlam2021/soa-entorno-integracion	**(1)**

Este se encuentra estructurado en tres directorios, de la siguiente manera:

- **Qemu_stm32:** Este directorio contiene el código fuente del emulador Qemu, que fue adaptado para esta investigación.
- **Stm32_demos:** Contiene diversos ejemplos de código fuente que fueron adaptados, para poder ser ejecutados en la emulación y en el hardware real de las placas mencionadas. A su vez contiene información acerca del hardware de dichos SE.
- **Workflows:** En donde se encuentra la configuración de la generación automática de la imagen en Docker Hub.

El método que se utiliza para construir la imagen de Docker de forma automática es a través del archivo *docker-image.yml.* Este archivo se configuró, para que se ejecuten automáticamente ciertos comandos, que se encuentran indicados dentro un archivo llamado *Dockerfile.* Esta secuencia de comandos, son ejecutados dentro de una máquina virtual (VM) que se encuentra en los servidores de Github, y son accionados cada vez que se actualiza el repositorio. Una vez que dentro de la VM se genera la imagen de Docker, automáticamente el archivo *yml* la carga y la publica dentro del registro de contenedores de Docker Hub. Para que de esta forma cualquier persona pueda descargarla y utilizarla fácilmente. Por consiguiente, la imagen publicada en esta investigación se puede encontrar desde línea de comando de la siguiente manera, mediante el nombre *soaunlam/qemu_stm2*:

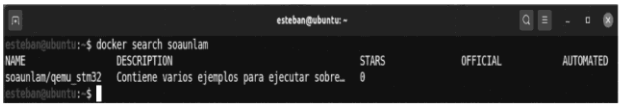


Fig. 1 Búsqueda de la Imagen Docker generada

La ventaja de haber utilizado los *workflows* de Github, a través del archivo *yml,* es que la creación de la imagen se realiza en los servidores de Github y no en la máquina local. Con lo cual, esto nos evita tener que realizar el trabajo manualmente y el tiempo de procesarlo localmente.

3.4 Contenido del Archivo Dockerfile

Los archivos *Dockerfiles* permiten generar imágenes personalizadas. En estos archivos se especifican los comandos, en forma de meta instrucciones, que Docker interpretará para construir la imagen deseada. El archivo Dockerfile correspondiente a este trabajo se encuentra publicado en el repositorio de esta investigación. El contenido de dicho archivo fue organizado, para usar la menor cantidad de capas resultantes que conforman la imagen, como es explicado en la sección de mejores prácticas [10]. Para ello en el Dockerfile, se especifica como capa base a la versión adaptada del Sistema Operativo Ubuntu 20.04. Se consideró conveniente, utilizar una versión determinada y no la última existente, debido a que entre distintas versiones pueden variar sus dependencias, generando una imagen de Docker defectuosa. Luego se instalaron las dependencias y bibliotecas de Linux que necesita Qemu, junto con programas adicionales. Algunos de ellos son *apt-utils, gcc-arm-none-eabi, gcc, python2.7, pkgconf, git, make, libglib2.0-dev, libpixman-1-dev,* entre otros. Es importante mencionar que, para el correcto funcionamiento de algunos ejemplos, fue necesario instalar las dependencias *open-ssh* y *net-tools,* dado que estos deben ser ejecutados en parte a través de terminales ssh. Una vez instaladas todas las dependencias, se borra la cache utilizada para su instalación, de manera de poder liberar espacio en la imagen generada. Luego se descarga el repositorio de esta investigación desde Github (1). Posteriormente, dentro se configura y compila el código de Qemu, que se encuentra dentro del directorio *Qemu_stm32*. A su vez también se compilan y se generan los archivos binarios de los distintos ejemplos. Seguidamente se configura la imagen para que el usuario pueda acceder al contenido de este a través de protocolo SSH. Para ello se modifican los archivos *sshd_config* y *.bashrc* para ello. De esta forma se genera la imagen de Docker con todo configurado e instalado, para que el desarrollador pueda ejecutar ejemplos de código emulados en Qemu rápidamente. También para que tenga la posibilidad de modificar su código accediendo a estos de forma externa, a través de SSH. De tal manera que pueda acceder a los archivos contenidos dentro de la imagen, a través de distintos programas, como por ejemplo editores de código.

3.5 Diferencias entre las placas soportadas por la imagen de Docker

Las tres placas *Stm32-p103*, *Stm32-Maple* y *Bluepill,* Fig. 2 (a, b y c), pertenecen a la misma serie denominada “convencional” de microcontroladores de STM32F1 con 32 bits. Esta arquitectura se encuentra bien equilibrada y se adapta a las necesidades básicas que se esperan en los mercados de consumo, donde las limitaciones de costos y el tiempo de comercialización son esenciales. Además, están diseñadas para responder a los requisitos de forma simple, robusta y con larga vida de utilidad.

(a) Stm32-p103 (b) Stm32-Maple (c) BluePill

Fig. 2 Modelos de placas soportados por esta investigación

Estas placas de desarrollo tienen de similitud, el voltaje de trabajo (2.0V a 3.6V), su rango de temperaturas (-40 °C a 105 °C) y su velocidad de funcionamiento (hasta 72MHz). No obstante, sus diferencias radican en su tamaño y sus interfaces de entrada y salida. La placa Stm32-p103 [11], tiene las dimensiones de 100 x 90 mm. Además, internamente posee un botón y dos LEDs, uno de estado y otro de encendido. Como interfaces externas posee un conector JTAG, 3 puertos USART, 2 conectores SPI, 2 del tipo I2C, un puerto CAN, un conector USB y 2 conectores ADC. Mientras que la placa Stm32-Maple [12],posee las dimensiones de 69 x 54 mm. A su vez, internamente posee 2 botones, uno de reset y otro programable. Además posee 3 LEDs, dos de estado y uno de encendido. Como interfaces externas tiene un pin de conexión SWD, un conector de UEXT, otro de PWR JACK y distintos puertos CON1-POWER, CON2-ANALOG, ON3_DIGITAL y CON4 -DIGITAL. Además, posee un conector LI BAT, USB, un conector para tarjeta SD/MMC y un puerto CAN. Por otro lado, la placa BluePill [13]es la más pequeña de todas, con tan solo 23 x 53 mm. Además, internamente posee un botón de reset y contiene dos LED, uno de estado y otro de encendido. Adicionalmente, como conexiones externas posee un puerto USB, un pin de conexión SWD, 2 conectores del tipo I2C, 2 de SPI, un puerto CAN, un conector JTAG y 2 ADC.

3.6 Código fuente del contenido de la Imagen

Como se mencionó anteriormente, este trabajo se basó en el proyecto de *Beckus* y se lo adaptó para que pueda funcionar correctamente, en las emulaciones de las placas previamente mencionadas. En este apartado se describen brevemente las modificaciones que se debieron realizar en dicha adaptación. Estos ajustes fueron tanto en el código fuente base de Qemu y en los ejemplos. Dado que estos fueron creados para funcionar emulando solamente la placa *stm32-p103*, por lo que no estaban preparados para funcionar en su totalidad en las placas *stm32-Maple* y *BluePill*. Esto se debió principalmente a que los microcontroladores de esas placas poseen diferentes configuraciones internas. Por ese motivo en el repositorio de git (1), se crearon tres subdirectorios diferentes. Los cuales contienen los ejemplos de código fuente de programas, para ejecutar en cada una de las placas mencionadas. Ya que entre ellos presentan pequeñas modificaciones que lo hacen funcional. En este sentido, una de las adaptaciones que se debió

realizar, consistió en la asignación de la USART[2], la cual es diferente dependiendo del hardware que se esté utilizando. Los puertos de las USART en los sistemas embebidos resultan de vital importancia, dado que a través de ellos el desarrollador puede realizar una depuración indirecta de los programas que se ejecute en dicho hardware. La gran mayoría del código fuente de los ejemplos que se encuentran en la imagen de Docker creada, utilizan este mecanismo de depuración. Otra de las modificaciones que fue necesario realizar, fue la adaptación del manejador de interrupciones, dado que se mapea diferente dependiendo de la placa emulada. Además, se adaptó la conexión interna en determinados pines, como por ejemplo el LED de testeo. Por otra parte, se modificó el código fuente de Qemu, para poder generar la emulación de los eventos de pulsadores externos, en las placas *stm32-Maple* y *BluePill*. Estos eventos se crearon, de forma tal que se generen cuando el usuario envié un comando *sendkey,* desde la consola de Qemu, al ejecutar cualquier programa en él. De esta forma se podrá emular la acción de un actuador del tipo pulsador. Adicionalmente se adaptó el código fuente, para que la utilización de los registros de hardware sea a través de la utilización de funciones de bibliotecas. Las cuales corresponden al funcionamiento del hardware pertinente. Al mismo tiempo, se generó un mecanismo de compilación que permite generar los binarios de todos los ejemplos.

3.7 Ejecución del emulador Qemu dentro de la Imagen Docker

Cuando el usuario descargue la imagen de Docker de este trabajo, del repositorio Docker Hub, deberá asociarle un contenedor para poder trabajar con ella. Para ello una de las formas de uso, es a través del comando "`docker run -it`". El cual crea un contenedor, asociado a una pseudo-terminal interactiva, la cual permite interactuar por medio de línea de comandos. Esto se puede visualizar en la Fig. 3. Dependiendo de lo que desee hacer, desde dicha terminal se podrá ejecutar cualquiera de los ejemplos, que se encuentran dentro de alguno de los subdirectorios: *Stm32-p103, Stm32-Maple y BluePill*. Los ejemplos de código fuente que se encuentran en estos directorios, permiten entre otras cosas: emular el encendido y apagado de un led de testeo, emular un programa que trabaje con un pulsador, emular un sensor que trabaje con valores analógicos, hacer programas que emulen interrupciones por software y hardware, emular el trabajo del temporizador que posee cada placa, poder emular el trabajo de los puertos USART para la depuración remota indirecta y permitir trabajar con programas que funcionen con el Sistema Operativo de Tiempo Real (*FreeRTOS*). En esta última opción, se permite ejecutar, en los sistemas embebidos emulados, programas que funcionan en un único o múltiples hilos de ejecución.

[2] USART: Dispositivo que controla los puertos y dispositivos serie. Se encuentra integrado en la placa base o en la tarjeta adaptadora del dispositivo.

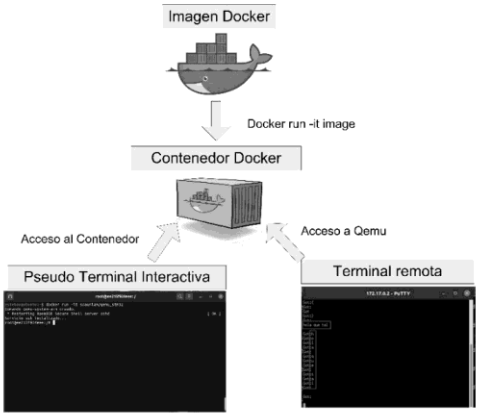


Fig. 3 Forma de ejecución de programas emulados en Qemu dentro del contenedor Docker

Para poder usar los binarios de los ejemplos en Qemu, que son generados cuando se compilan en el momento en que se genera la imagen, se debe ejecutar el emulador de una forma específica. Esta puede variar dependiendo del ejemplo que se quiera ejecutar, pero en la mayoría de los casos se debe seguir una forma base estándar de secuencia de pasos. Primero, el usuario deberá ejecutar el programa Qemu de la siguiente manera, desde la línea de comandos de la pseudo-terminal.

```
qemu-system-arm -M <nombre de la placa> -kernel <ar-
chivo.bin> -serial tcp::7777,server -nographic
```

El comando **qemu-system-arm,** es el archivo ejecutable del programa Qemu. Con el parámetro **-M,** se indica el modelo de placa que se desea emular. En nuestro caso, se puede emular las placas *stm32-p103, stm32-maple* y *stm32-f103c8.* Luego con la opción **-kernel,** se le indica la ubicación del programa binario que se desea ejecutar dentro del sistema embebido emulado. Adicionalmente con la opción **-serial,** se le dice al emulador que todas las entradas y salidas que se envíen al puerto serial de las placas sean redirigidas al puerto **7777** usando el protocolo TCP. Finalmente, con la opción **-nographic,** se deshabilita las salidas gráficas que genera el emulador.

Cuando se inicia la ejecución de Qemu, el emulador se quedará esperando una conexión externa a través del puerto 7777 del contenedor. Para ello el usuario deberá utilizar otra terminal remota y conectarse al puerto anteriormente indicado, utilizando la dirección IP que tenga asignado el contenedor de Docker, mediante el protocolo Telnet. Esto se muestra en la Fig. 4. Una vez que se establece la conexión, el programa emulado continuará su ejecución normalmente. Como ya se mencionó, la forma de ejecución puede variar, dependiendo del ejemplo que se desea ejecutar. Por ese motivo,

en esta investigación se generó un instructivo en forma de tutorial. En donde se detallan los pasos que se deben seguir, para poder ejecutar cada uno de los ejemplos disponibles dentro del entorno del contenedor. Este tutorial se encuentra dentro del repositorio generado (1), y por consiguiente, también se encuentra dentro de la imagen Docker generada.

En muchos de los ejemplos que se encuentran dentro de la imagen de Docker, se realiza depuración indirecta en forma remota a través de los puertos seriales. Como consecuencia de que el puerto serial es utilizado, tanto para mostrar datos en una terminal remota, como para ingresarlos a través de ella. Esto se puede visualizar en la siguiente figura, donde se muestra un caso de datos de entrada y salida a través de una terminal de este tipo.

Fig. 4 Terminal remota con datos de entrada y salida a través del puerto serial

4 Conclusiones

En este trabajo se presentó una alternativa, para que los desarrolladores de soluciones de sistemas embebidos, que pueden ser utilizados para IoT, realicen pruebas en placas STM32 emuladas a través del programa Qemu. Este trabajo les puede ayudar a los desarrolladores, establecer si determinado hardware le es o no de utilidad para sus proyectos, sin necesidad de adquirir el hardware físico. De manera que lo pueda realizar de forma rápida y sencilla, sin tener que realizar tediosas instalaciones y configuraciones de programas. Debido a que el emulador Qemu se encuentra empaquetado, configurado y automatizado dentro de una imagen Docker, fácil de emplear. Si bien este trabajo se centró en la emulación de tres placas: *stm32-p103, stm32-Maple* y *stm32-f103c8,* se planea en un futuro realizar el mismo trabajo de automatización, configuración y emulación mediante contenedores Docker para otros tipos de placas de desarrollo.

5 Referencias

1. Rose, K., Eldridge, S., Chapin, L.: La Internet De Las Cosas - Una Breve Reseña. , Reston, United State (2015).
2. Valiente, W., Carnuccio, E., Volker, M., de Luca, G., Villca, R., Adagio Matías: Entorno de contenedores de emuladores que contienen sistemas embebidos. In: XXIII Workshop de Investigadores en Ciencias de la Computación. pp. 12–17 (2021).
3. Eclipse Foundation: https://eclipse-embed-cdt.github.io/debug/qemu/.
4. Beckus: http://beckus.github.io/qemu_stm32/.
5. Lovric, D., Olsson, C.: Virtual Controllers (Tesis de Maestría). Department of Automatic Control, Lund University, Sweden (2016).
6. Muñoz, J.F., Goenaga, I.M.: Ofera Project: Open Framework for Embedded Robot Applications, European Union's Horizon 2020, Unión Europea (2019).
7. Amamory: https://hub.docker.com/r/amamory/qemu-stm32.
8. Chelladhurai, J.S., Singh, V., Raj, P.: Learning Docker - Second Edition: Build, ship, and scale faster. Packt Publishing, Birmingham, Reino Unido (2017).
9. Miell, I., Sayers, A.H.: Docker in practice. Manning Publications Co., Shelter Island, NY (2019).
10. Goasguen, S.: Docker Cookbook: Solutions and Examples for Building Distributed Applications. (2015).
11. Olimex: STM-P103 development board - User's manual., Plovdiv, Bulgaria (2016).
12. Olimex: OLIMEXINO-STM32 development board - Users Manual. , Plovdiv, Bulgaria (2011).
13. STMicroelectronics: STM32F103x8 DataSheet. (2015).

12. CACIC 2021 - Certificados


CACIC 2021
RedUNCI
XXVII CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN
Por medio del presente se CERTIFICA que:
Mariano Volker
Ha participado en calidad de AUTOR del trabajo "Entorno de contenedores con emuladores de sistemas embebidos STM32" en el XXVII CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN llevado a cabo de manera virtual por la Facultad de Ciencias Exactas de la UNSa del 04 al 08 de octubre de 2021.
Salta, Argentina
Lic. Patricia Pesado
Coordinadora
Red UNCI
Ing. Daniel Hoyos
Decano
Facultad de Ciencias Exactas
UNSa


CACIC 2021
RedUNCI
XXVII CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN
Por medio del presente se CERTIFICA que:
Raúl Villca
Ha participado en calidad de AUTOR del trabajo "Entorno de contenedores con emuladores de sistemas embebidos STM32" en el XXVII CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN llevado a cabo de manera virtual por la Facultad de Ciencias Exactas de la UNSa del 04 al 08 de octubre de 2021.
Salta, Argentina
Lic. Patricia Pesado
Coordinadora
Red UNCI
Ing. Daniel Hoyos
Decano
Facultad de Ciencias Exactas
UNSa

CACIC 2021

RedUNCI

XXVII CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN

Por medio del presente se CERTIFICA que:

Matías Adagio

Ha participado en calidad de AUTOR del trabajo "Entorno de contenedores con emuladores de sistemas embebidos STM32" en el XXVII CONGRESO ARGENTINO DE CIENCIAS DE LA COMPUTACIÓN llevado a cabo de manera virtual por la Facultad de Ciencias Exactas de la UNSa del 04 al 08 de octubre de 2021.

Salta, Argentina

Lic. Patricia Pesado
Coordinadora
Red UNCI

Ing. Daniel Hoyos
Decano
Facultad de Ciencias Exactas
UNSa

Código	FPI-013
Objeto	Formulario de evaluación de alumnos integrantes de equipos de investigación
Usuario	Director de proyecto de investigación
Autor	Secretaría de Ciencia y Tecnología de la UNLaM
Versión	5
Vigencia	03/9/2019

ANEXO II:

Código	FPI-013
Objeto	Formulario de evaluación de alumnos integrantes de equipos de investigación
Usuario	Director de proyecto de investigación
Autor	Secretaría de Ciencia y Tecnología de la UNLaM
Versión	5
Vigencia	03/9/2019

Unidad Académica: Departamento de Ingeniería e Investigaciones Tecnológicas
Código: C2-ING-067
Título del Proyecto: Entorno de integración continua para validación de sistemas embebidos de tiempo real
Director del Proyecto: Lic. Graciela Elisabeth De Luca
Fecha de inicio: 01/01/2020.
Fecha de finalización: 31/12/2021

1. Datos del alumno
Apellido y Nombre: Adagio, Matías Ezequiel
DNI: 34424221
Unidad Académica: Departamento de Ingeniería e Investigaciones Tecnológicas
Carrera que cursa: Ingeniería en Informática
Período evaluado: 01-2020 al 12-2021

2. Dictamen de evaluación de desempeño del alumno:
Colocar una cruz donde corresponda

2.1 Satisfactorio: **X**
2.1 No satisfactorio:

Fundamentos del dictamen:

El alumno tuvo un buen desempeño en las actividades en las que participó durante el proyecto. Colaboró en la investigación de técnicas de profiling, que posteriormente se aplicaron sobre la imagen de Docker desarrollada. Al mismo tiempo ayudó en la confección de artículos científicos, que fueron presentados a distintos congresos. A su vez formo parte de las tareas de automatización de la creación de la imagen Docker, a partir de los cambios hechos en el repositorio de Github.

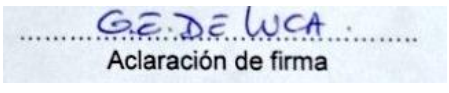
3. Propuesta de continuidad en el proyecto (si corresponde según duración estimada)
Colocar una cruz donde corresponda

3.1 Continuar en el presente proyecto: **X**
3.2 No continuar en el presente proyecto:

Fundamentos del dictamen:
El grupo de trabajo está muy conforme con el desempeño de este alumno. Por lo que queremos que continúe su participación tanto en investigación como en el aula.

San Justo,
24 de febrero de 2022
Lugar y fecha


Firma del Director del Proyecto



Aclaración de firma

Código	FPI-013
Objeto	Formulario de evaluación de alumnos integrantes de equipos de investigación
Usuario	Director de proyecto de investigación
Autor	Secretaría de Ciencia y Tecnología de la UNLaM
Versión	5
Vigencia	03/9/2019

Unidad Académica: Departamento de Ingeniería e Investigaciones Tecnológicas
Código: C2-ING-067
Título del Proyecto: Entorno de integración continua para validación de sistemas embebidos de tiempo real
Director del Proyecto: Lic. Graciela Elisabeth De Luca
Fecha de inicio: 01/01/2020.
Fecha de finalización: 31/12/2021

1. Datos del alumno
Apellido y Nombre: Villca, Raúl David
DNI: 34370611
Unidad Académica: Departamento de Ingeniería e Investigaciones Tecnológicas
Carrera que cursa: Ingeniería en Informática
Período evaluado: 01-2020 al 12-2021

2. Dictamen de evaluación de desempeño del alumno:
Colocar una cruz donde corresponda

2.1 Satisfactorio: **X**
2.1 No satisfactorio:

Fundamentos del dictamen:
El alumno se desenvolvió eficientemente en las tareas que se le asignaron. Participando en la ampliación y mejora del Entorno construido de la imagen Docker, que se publicó en el repositorio Docker Hub. Además, colaboró en la elaboración de los artículos científicos que fueron presentado en los congresos y revista expuestos en el informe.

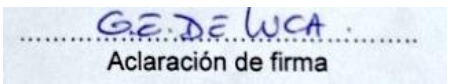
3. Propuesta de continuidad en el proyecto (si corresponde según duración estimada)
Colocar una cruz donde corresponda

3.1 Continuar en el presente proyecto: **X**
3.2 No continuar en el presente proyecto:

Fundamentos del dictamen:
El grupo de trabajo está muy conforme con el desempeño de este alumno. Por lo que queremos que continúe su participación tanto en investigación como en el aula.

San Justo,
24 de febrero de 2022
Lugar y fecha


Firma del Director del Proyecto



Aclaración de firma