Versión del Sistema Operativo XINU para la Arquitectura AVR con la Finalidad de ser Utilizado como RTOS Académico

Rafael Ignacio Zurita*, Candelaria Alvarez,
Miriam Lechner, and Alejandro Mora

Departamento de Ingeniería de Computadoras, Facultad de Informática,
Universidad Nacional del Comahue, Neuquen, Argentina
{rafa,candelaria.alvarez,mtl,alejandro.mora}@fi.uncoma.edu.ar
http://www.se.fi.uncoma.edu.ar/

Resumen Xinu es un sistema operativo académico desarrollado originalmente por Douglas Comer en la Universidad de Purdue, a fines de los 70. Desde entonces, ha sido portado a una gran variedad de plataformas de hardware y arquitecturas, y es utilizado, principalmente, como herramienta de investigación y educación. Xinu utiliza primitivas sencillas para proporcionar varios de los componentes y funcionalidades que existen en muchos sistemas operativos convencionales. Actualmente existen versiones de Xinu para x86 (PC), ARM, MIPS, y maquinas virtuales. Sin embargo, la mayoría de las versiones existentes se ejecutan en microprocesadores capaces de ejecutar sistemas operativos mas completos como Linux o Windows. En este artículo se presenta el trabajo realizado para lograr una implementación de XINU para un microcontrolador, de arquitectura Harvard, con sólo 32KB de memoria flash, y 2KB de RAM. Se evaluó el resultado portando un shell con varias utilidades de tipo UNIX, y también con un trabajo experimental de tesis de grado en donde se requirió el uso de un sistema operativo de tiempo real. Los resultados muestran que XINU tiene potencial para ser portado a otras familias de microcontroladores con pocos recursos, y ser utilizado también como RTOS académico en esas plataformas.

Keywords: Sistema Operativo de Tiempo Real, RTOS, Sistema Embebido, Sistema Operativo, Xinu, AVR, atmega328p, Arduino, microcontrolador

1. Introducción

Regionalmente, en Argentina, la enseñanza universitaria de programación de sistemas embebidos de tiempo real se realiza utilizando como herramienta de apoyo práctico algún sistema operativo de tiempo real (RTOS) [1]. Estos, se ejecutan principalmente en microprocesadores de baja complejidad, y mayormente, en microcontroladores.

*Agradecemos al profesor retirado Ing. Rodolfo del Castillo, quien nos presentó a Xinu como opción académica de diseño elegante.

Los sistemas operativos de tiempo real (RTOS) no son sistemas operativos completos. Están compuestos, básicamente, por un gestor de tareas apropiativo (preemptive), y de mecanismos para la sincronización y comunicación entre las mismas [2]. Su principal característica es que permite la ejecución concurrente de tareas o procesos de manera predecible, por lo que, usualmente, el planificador de CPU funciona en base a prioridades, y de manera round-robin en caso de que las prioridades sean las mismas.

Este reducido conjunto de características está directamente relacionado con el hardware donde se ejecuta. Los RTOS son utilizados mayormente en plataformas de cómputo basado en microcontroladores de bajos recursos para automatización y control. En los últimos años, con el crecimiento exponencial de desarrollo de dispositivos IOT, su uso ha crecido en ambientes de sensores conectados a Internet, también controlados generalmente por microcontroladores [3].

Desafortunadamente, no existe una gran variedad de RTOS académicos. Generalmente, en cursos especializados y en materias de grado relativas a sistemas embebidos de tiempo real, se emplea como herramienta práctica, algún RTOS de uso profesional, el cual suele estar diseñado y desarrollado para uso industrial. Algunos ejemplos de estas herramientas son FreeRTOS, ChibiOS/RT, Cesium RTOS, o VxWorks [4], [5], [6] y [7]. Los primeros dos son sistemas open source, los últimos son privativos. Todos estos RTOS proveen una API orientada al programador profesional de estos sistemas, y no al estudiante universitario de grado. Además, su estructura interna no suele ser fácil de comprender durante sólo un cuatrimestre (duración promedio de materias de grado universitaria). Esto limita el recorrido académico, y el RTOS seleccionado por el docente se acota a ser utilizado únicamente como herramienta de apoyo, sin tener la posibilidad de explorar otros caminos, como lo son el análisis de sus estructuras de datos y algoritmos internos, o la expansión y/o modificación de sus funcionalidades y servicios; requisitos necesarios usualmente en investigación. Por tal motivo se propone en este trabajo la modificación de un sistema operativo académico, para ser utilizado en ambientes de microcontroladores, y también como sistema operativo de tiempo real. Se seleccionó el sistema operativo Xinu, debido a que su estructura interna es elegante, su API es sencilla, y ya cuenta con un planificador de CPU apto para ser utilizado como RTOS. Como plataforma de hardware destino se seleccionó el microcontrolador AVR atmega328p, ya que es el microcontrolador original de las placas de desarrollo Arduino, muy disponible en el mercado regional.

2. Antecedentes

2.1. Sistemas embebidos de tiempo real y RTOS

Un sistema es de tiempo real si su correctitud está definida por la correctitud de los resultados computados, y también, por cumplir con los tiempos de respuesta definidos en sus especificaciones [8]. Si el sistema no es capaz de responder a un evento o tarea (definida en los requerimientos del sistema), en el

tiempo máximo permitido (tambien definido en las especificaciones del sistema) entonces se dice que el sistema falló, y no es de tiempo real.

Usualmente, al diseñar un sistema de tiempo real, se definen eventos y acciones a ser llevadas a cabo. Para cada uno de estos eventos se define también, con precisión, el tiempo de respuesta máximo permitido por parte del sistema. Luego, este diseño se divide usualmente en tareas independientes, que serán ejecutadas de manera concurrente, en tiempo de ejecución. Como software de apoyo, se utiliza un sistema operativo de tiempo real. Este software de apoyo es el encargado de ejecutar las tareas de manera concurrente, y también, da soporte para lograr los tiempos de respuesta para cada evento. Esto no es un proceso automático. Es decir, utilizar un sistema operativo de tiempo real no implica que el sistema logrado será de tiempo real. Para lograr el correcto funcionamiento del sistema, los desarrolladores deben especificar para cada tarea una prioridad, relacionar las tareas con los eventos, y definir como deben sincronizarse y comunicarse esas tareas. En tiempo de ejecución, el sistema operativo de tiempo real será el encargado de asegurar que siempre se está ejecutando la tarea de mayor prioridad. De esta manera, en tiempo de diseño y desarrollo, se podrá evaluar el modelo diseñado ante todas las situaciones posibles, para corroborar que siempre el tiempo de respuesta para un evento y/o tarea, es menor a la cota definida en los requerimientos. Si la cantidad de estados del sistema y situaciones posibles son inabarcables, los desarrolladores pueden realizar simulaciones y aproximaciones, para evaluar si el modelo diseñado cumplirá con los plazos requeridos.

Generalmente, el planificador de CPU (scheduller) de un sistema operativo de tiempo real es de prioridad fija (la prioridad de cada tarea se define en tiempo de compilación) y apropiativo (preemptive). Cuando el sistema está en ejecución, cada vez que el RTOS activa una tarea, esto es, coloca la tarea en estado de "listo para ejecutar", verifica su prioridad. Si esta prioridad es mas alta que la tarea que actualmente utiliza la CPU, entonces el RTOS realiza un cambio de contexto y coloca a la nueva tarea de mas alta prioridad a ejecutar. Si existen varias tareas listas con la misma prioridad que la tarea que está actualmente utilizando la CPU, entonces el RTOS realiza una expropiación de la CPU en intervalos regulares, empleando round-robin para asignarla a otra tarea.

Existen otros métodos y formas de desarrollar un sistema de tiempo real, pero el uso de un RTOS es la situación mas común. En síntesis, un RTOS debe contar con las siguientes caracterísitcas:

- Un RTOS (Real-Time Operating System) debe poder gestionar tareas diferentes (multiprogramación/multi hilos), y ser apropiativo (preemptive).
- Cada tarea debe contar con una prioridad, y el RTOS siempre ejecuta la tarea lista de mayor prioridad.
- El RTOS debe contar con mecanismos de sincronización y comunicación de tareas.
- Debe existir un sistema de herencia de prioridad.
- El RTOS debe permitir eventos asincrónicos, como pueden ser interrupciones de E/S.

Contando con estas características, los desarrolladores pueden diseñar un sistema cuyo funcionamiento es predecible, la cual es una condición necesaria para evaluar el diseño, y conocer si el sistema resultante cumple con los plazos definidos, y por lo tanto, ser de tiempo real.

2.2. Xinu

Xinu es un sistema operativo académico desarrollado originalmente por Douglas Comer en la Universidad de Purdue, a fines de los 70. Desde entonces, ha sido re-escrito y portado a una gran variedad de plataformas de hardware y arquitecturas, y es utilizado, principalmente, como herramienta de investigación y educación. Las versiones mas recientes (2015) de Xinu son para x86 (PC), ARM, MIPS, y maquinas virtuales [9]. A pesar de que Xinu comparte algunos conceptos y alguna terminología de componentes con UNIX, el diseño interno de Xinu difiere completamente de UNIX. Xinu es un sistema operativo con un diseño elegante, cuya estructura interna está conformada por una jerarquía multinivel de componentes esenciales, pero con las cuales puede luego desarrollarse otras más complejas. Tiene soporte para la creación dinámica de procesos, reserva dinámica de memoria, sistemas de archivos, soporte de red, y funciones de E/S independiente de los dispositivos. También ofrece servicios de comunicación y sincronización entre procesos. Xinu no implementa memoria virtual, ni tiene soporte para hardware de paginación. En tiempo de ejecución, la imagen de Xinu con un conjunto de aplicaciones se carga completamente en memoria RAM, utilizando un único espacio de direcciones para todos los procesos. Esto significa que los procesos comparten la memoria, aunque Xinu gestiona para cada proceso su propia pila.

A pesar de que las versiones de Xinu existentes son para procesadores de 32bits, el sistema fue igualmente seleccionado para este trabajo debido a que también es indicado en ambientes embebidos. El código fuente de Xinu es pequeño en comparación con otros sistemas operativos académicos (como por ejemplo MINIX). La versión de Xinu para x86 contiene aproximadamente 10 mil líneas de código fuente en lenguaje C. De estas, el 75 % pertenece a código fuente de drivers, soporte de red, y cabeceras .h; por lo que el kernel está desarrollado en sólo unas 2500 líneas de código en lenguaje C.

2.3. Trabajos Relacionados

En [10] y [11] se detallan el port del sistema operativo de tiempo real uC/OS-II a dos arquitecturas diferentes, ambas de 16 bits. Los trabajos describen ports que siguieron la metodología propuesta por la documentación oficial del RTOS en [13]. uC/OS-II es un RTOS open source, desarrollado por Micrium Inc. Ofrece una API orientada a la industria, y Micrium ofrece soporte comercial y licencias no libres para empresas que lo requieran. En [14] y [12] se detalla el proceso realizado para portar FreeRTOS y VxWorks a nuevas plataformas (para x86 en el caso del artículo de FreeRTOS, y MPC8313E para el caso de VxWorks). El eje de ambos trabajos fue exponer un ejemplo de como portar un sistema

operativo de tiempo real a nuevas arquitecturas, orientado a lectores que no tengan experiencia previa en este tipo de trabajo. En [15] la IEEE documenta un estándar que describe las características necesarias de un sistema operativo de tiempo real para sistemas embebidos compuestos por procesadores de 16 bits, sin MMU. El estándar reúne las características generales que pueden encontrarse de manera básica en casi todos los RTOS disponibles. En [16] se describe un modelo para el diseño de RTOS independientes del hardware. El foco está puesto en separar el código general del kernel RTOS de las características particulares del hardware. A partir de una descripción del hardware final se podrá, en tiempo de compilación, generar código específico faltante, por ejemplo, del cambio de contexto de tareas para un microcontrolador específico.

En nuestra búsqueda, no hemos encontrado trabajos recientes de cómo lograr un port de un sistema operativo académico convencional para ser ejecutado en ambientes embebidos de microcontroladores, y en caso de ser necesario, poder ser utilizado como RTOS. Sin embargo, los trabajos mencionados nos aportaron principios de diseño y metodologías, que pudimos emplear para lograr la versión de Xinu presentada en este artículo.

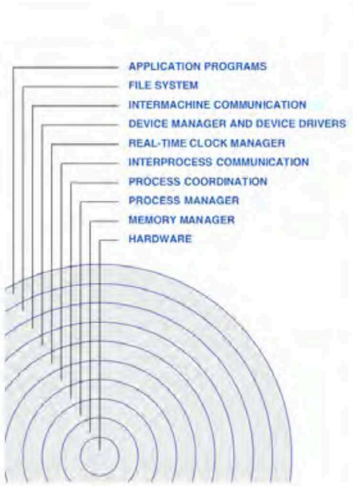
3. Metodologia y Desarrollo

El trabajo de transferencia de Xinu al microcontrolador AVR atmega328p se realizó utilizando, principalmente, la metodología descripta en [17].

Los pasos (etapas) de esta metodología se pueden observar en la Fig. 1 (b), y está directamente relacionada a la estructura interna de Xinu. En la Fig. 1 (a) se puede observar un diagrama de esta estructura, donde los componentes de software de Xinu están organizados en una jerarquía multinivel. Cuando un software está diseñado de esta manera, las interconexiones entre los componentes son claras, y el diseño interno es fácil de comprender. Ambas figuras fueron extraídas de [17].

No todas las etapas fueron necesarias en este trabajo, debido a que un RTOS no contiene todos los componentes de un sistema operativo tradicional. Además, del listado en la Fig. 1 (b) no fue necesario realizar los pasos 13 a 17, ya que el microcontrolador destino no cuenta con el hardware necesario para esos componentes.

Etapa 1. La primer tarea comprendió estudiar la arquitectura de hardware destino. Esta tarea tuvo una gran relevancia en este trabajo, debido a que la estructura de hardware del microcontrolador seleccionado difiere en gran medida de los procesadores donde Xinu se ejecuta. Los procesadores x86, ARM y MIPS soportados por Xinu presentan un modelo Von Neumann. Esto es, una única memoria para los programas y los datos (aún si algunos de estos contienen cachés de memoria interna en el procesador, diferentes para datos y código, ya que esta es transparente para el código binario del programa). En cambio, la arquitectura destino AVR posee una arquitectura Harvard. Esta arquitectura presenta al menos dos memorias accedidas por diferentes buses y direcciones:



(a)

Step	Description
1.	Learn about the new hardware
2.	Build cross-development tools
3.	Learn the compiler's calling conventions
4.	Build a bootstrap mechanism
5.	Devise a basic polled output function
6.	Load and run a sequential program
7.	Port and test the basic memory manager
8.	Port the context switch and process creation functions
9.	Port and test the remaining process manager functions
10.	Build an interrupt dispatcher
11.	Port and test the real-time clock functions
12.	Port and test a tty driver
13.	Port or create drivers for an Ethernet and other devices
14.	Port the network stack, including Internet protocols
15.	Port the remote disk and RAM disk modules
16.	Port the local and remote file system modules
17.	Port the Xinu shell and other applications

(b)

Figura 1. (a) Estructura interna de Xinu en jerarquía multinivel. (b) Serie de pasos (etapas) secuenciales necesarios para realizar un port de Xinu.

una memoria para los programas, y una memoria para los datos. La memoria para el código a ejecutar por la CPU, en el atmega328p, está compuesta de una memoria FLASH de 32KB. La memoria para los datos dinámicos es una pequeña memoria RAM de 2KB. Además, se cuenta con una tercera memoria de uso general para datos, no volatil, de 1KB. Esta tercera memoria presente en el AVR es de tipo EEPROM. AVR presenta además una mejora con respecto a una arquitectura Harvard teórica. Los procesadores AVR pueden almacenar en la memoria de programa (usualmente de mayor tamaño que la RAM) datos de solo lectura. Esto significa que las variables que sean constantes pueden estar almacenadas en la FLASH, y de esta manera, no ocupar espacio en RAM. Algo útil, siendo que las variables constantes no serán modificadas durante toda la vida del programa. Otra característica importante que se debió tener en cuenta es que la CPU del AVR es de 8 bits, y que las direcciones son de 16 bits. Un puntero de C es entonces de 16 bits, mientras que el dato natural con el que trabaja el procesador es de 8 bits.

Etapa 2. La segunda tarea consistió en contar con un cadena de herramientas de desarrollo para la arquitectura AVR. La computadora de desarrollo (host) seleccionada fue una PC con GNU/Linux, lo cual facilitó esta tarea en gran medida, ya que la distribución Linux utilizada trae empaquetado un compilador cruzado de C para AVR, los binutils (vinculador, ensamblador, etc) para AVR y la biblioteca de C avr-libc.

Etapas 3 a 6. La convención de llamadas utilizada fue la implementada por GCC. El mecanismo de bootstrap fue realizado utilizando el bootloader de Arduino, y la tarea de cargar y ejecutar un programa básico fue realizada compilando un programa en C, y utilizando los scripts ld del vinculador predeterminados para ese microcontrolador. De esta manera, el compilador avr-gcc junto con su ensamblador y vinculador, son capaces de generar un ejecutable que puede ser transferido a la flash del atmega328p y ser ejecutado por el bootloader de Arduino ya almacenado en la flash. La complejidad aquí radica en asignar, en tiempo de compilación, las direcciones ROM (flash) y RAM reales físicas a los distintos símbolos del programa binario (por ejemplo, el entry point del programa en C, o la dirección para el puntero de pila). En nuestro caso, esta asignación se realizó automaticamente via los scripts ld predeterminados del vinculador gcc para el microcontrolador atmega328p.

Etapa 7. La cuarta tarea consistió en adaptar las cuatro funciones básicas de gestión de memoria de Xinu, y verificarlas. Se adaptaron las funciones utilizando la asignación de direcciones RAM de la etapa anterior, la cual permite conocer los límites de la memoria física disponible. También se modificó el sistema de compilación, adaptando el archivo `compile/Makefile` para que utilice el compilador GCC para avr.

Etapas 8 y 9. Esta quinta tarea demandó la mayor cantidad de tiempo, y consistió en portar el cambio de contexto: rutina en ensamblador dependiente de la arquitectura, la cual resguarda el estado del procesador para un proceso en su pila, y carga un estado previo del proceso que continuará usando la CPU, desde la pila del proceso a reanudar. También se realizó, en esta tarea, el port de varias rutinas de creación y gestión de procesos. Una vez completada esta etapa fue posible contar con multiprogramación. Además, el mismo kernel de Xinu se convierte en un proceso, y los detalles de implementación, de la conversión de un código secuencial, en ejecución, a varios procesos, son tal vez los mas complejos de comprender. Encontramos aquí una dificultad extra de suma importancia para todo el port: la memoria RAM física disponible ya no fue suficiente para los componentes portados en esta etapa. Por tal motivo, se estudiaron alternativas, y utilizando diferentes técnicas en simultáneo, se lograron los objetivos de esta fase. A continuación se enumeran estas técnicas, las cuales pueden ser de interés para futuros ports, incluso de otros sistemas operativos a microcontroladores:

1. Utilizar opciones de optimización del compilador, para reducir el tamaño del ejecutable.
2. Reducir las estructuras de datos siendo portadas. Por ejemplo, el elemento PID de la estructura de datos PCB de la tabla de administración de procesos, era de 4 bytes de tamaño. Se utilizó un tipo de datos uint8, ya que para esta arquitectura es altamente probable que no haya nunca mas de 255 procesos activos (de hecho, no se cuenta con los recursos para que esto suceda). Con la misma metodología se estudiaron los demás elementos de cada estructura

de datos, con el fin de cambiar los tipos por tipos mas pequeños, y de esta manera, reducir sustancialmente el tamaño de las estructuras de datos en RAM, sin perder su semántica ni su funcionalidad.

3. Almacenar las constantes en la memoria de código flash (ROM). Para esta arquitectura y compilador, esta técnica es alcanzada anteponiendo las palabras reservadas `const __flash` al tipo de una variable o estructura siendo declarada tentativamente. Estas palabras reservadas le indican al compilador que la variable o estructura será de solo lectura, y que debe ser alojada en la memoria FLASH junto con el código ejecutable. Una estructura de ejemplo que fue alojada en FLASH de esta manera es la estructura `devtab[]`, la cual es un arreglo de estructuras, donde cada elemento del arreglo representa un dispositivo de E/S, y contiene además de información de administración del dispositivo, punteros a funciones que implementan las llamadas al sistema para el dispositivo en particular (open(), read(), write(), etc). Todo este arreglo de estructuras no cambia en tiempo de ejecución, por lo que pudo ser declarada como constante y ubicada en la FLASH del microcontrolador.

Aplicando estas técnicas conforme se fue portando las estructuras necesarias se logró reducir el tamaño del ejecutable a unos pocos KB. Cabe mencionar, que en un principio, Xinu requiere de varios MB de RAM cuando se utiliza en computadoras de propósito general. Finalmente, cuando se alcanzó la meta de alojar el ejecutable de Xinu en su etapa actual en FLASH, y con cierta disponibilidad de RAM, se pudo contar con un sistema Xinu multiprogramado capaz de trabajar con tareas cooperativas concurrentemente.

Etapas 10 a 12. En esta última fase de nuestro trabajo se escribieron dos drivers de dispositivos de E/S: uno para controla un temporizador (timer) de hardware del AVR, y el segundo para controlar el dispositivo UART, el cual será utilizado por Xinu como tty (CONSOLA de interfaz con el usuario). El driver del temporizador realizará interrupciones cada un milisegundo. Dependiendo del QUANTUM configurado en Xinu, cada una cierta cantidad de interrupciones el planificador de CPU de Xinu realizará un cambio de contexto, para poner en ejecución otra tarea en estado de LISTO para ejecutar.

4. Evaluación

Se realizaron tres pruebas de evaluación experimental, para verificar el funcionamiento de todos los componentes internos de esta nueva versión Xinu para AVR. Como hardware se utilizó una placa Arduino Nano, la cual presenta una interfaz USB para transferencia del firmware, y que Xinu finalmente utilizará como CONSOLA en tiempo de ejecución. Las tres pruebas fueron las siguientes:

En primer lugar se ejecutaron los programas presentados en el capítulo 2 de [17]. Estos programas muestran lo que es la ejecución concurrente de procesos, un ejemplo productor consumidor, y la sincronización entre procesos.

Luego, se realizó el port de todo el shell de Xinu, y de varias herramientas de tipo UNIX. Esto permitió utilizar la placa Arduino Nano como una antigua

estación UNIX, con un shell y varias utilidades. Esto verificó casi todos los componentes del sistema operativo portado.

Finalmente, se verificó este port de Xinu utilizándolo como RTOS:

- En dos ediciones de la materia *Programación de Sistemas Embebidos*, de la carrera Licenciatura en Ciencias de la Computación de nuestra universidad, donde se estudian conceptos de sistemas operativos de tiempo real, y de como usar un RTOS para desarrollar un sistema embebido; y
- En un trabajo experimental de tesis de grado, donde se requirió de un RTOS. El sistema controla varios sensores, entre ellos un magnetómetro y encoders ópticos, cuyos eventos asincrónicos requerían reportar el valor obtenido en tiempos límites, para que el cálculo de odometría de un robot experimental tuviese un error en los márgenes esperados.[19]

5. Conclusiones

En este artículo se presentó una descripción del trabajo realizado para lograr el port del sistema operativo académico Xinu, para ser utilizado en la arquitectura AVR (Arduino), como RTOS académico. El proceso demandó tres tareas principales. En primer lugar se estudiaron las características de los sistemas operativos de tiempo real, y la estructura interna de Xinu. Luego, se analizó la metodología empleada para realizar otros ports y se seleccionó una propuesta. Finalmente, el trabajo resultante fue evaluado a través de programas de prueba, utilizando un shell estilo UNIX, y en un trabajo experimental de tesis de grado.

Como trabajo futuro se espera evaluar su impacto académico, por ejemplo, observando el enlace entre materias en donde se pueda utilizar la misma herramienta. Una situación posible es utilizar Xinu en materias introductorias a la temática, como lo es el curso de *Sistemas Operativos*. Por lo que quedaría observar, si en materias superiores, como *Programación de Sistemas Embebidos*, de nuestras carreras, el uso de la misma herramienta tiene un mejor impacto de aceptación y agiliza recorridos académicos, utilizando esa ganancia para otros objetivos, por ejemplo, profundizando otros conceptos no vistos actualmente por falta de horas disponibles. También se propone evaluar un posible uso de Xinu en ambientes de microcontroladores, realizando nuevos ports, o promoviendo su uso en proyectos industriales.

Distribución. El código fuente del kernel Xinu elaborado en este trabajo, y aplicaciones de ejemplo, están disponible para descarga desde el sitio Web `http://se.fi.uncoma.edu.ar/xinu-avr/`

Referencias

1. Congreso Argentino de Sistemas Embebidos. Libro de artículos y reportes tecnológicos. (2020-2022). ISBN 978-987-46297-7-7, 978-987-46297-8-4, 978-987-46297-8-4.

2. Phillip A. Laplante; Seppo J. Ovaska. Fundamentals of Real-Time Systems. In Real-Time Systems Design and Analysis: Tools for the Practitioner , IEEE, 2012, pp.1-25, doi: 10.1002/9781118136607.ch1.
3. 2019 Embedded Markets Study Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments. EE Times and Embedded https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf
4. Warren Gay. Beginning STM32: Developing with FreeRTOS, libopencm3 and GCC 1st ed. ISBN 978-1484236239. Apress (2018).
5. Giovanni Di Sirio. ChibiOS/RT - The Ultimate Guide Book. 2020. https://www.chibios.org/dokuwiki/doku.php?id=chibios:documentation:books:rt:start
6. Cesium RTOS web page. https://weston-embedded.com/products/cesium
7. VxWorks web page. https://www.windriver.com/products/vxworks
8. Alan Burns, Andy Wellings. Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX (4th Edition). Addison Wesley. ISBN 978-0321417459 (2009).
9. The Xinu Web Page https://xinu.cs.purdue.edu/
10. Sobaihi, Khaled. (2007). Porting the µC-OS-II Real Time Operating System to the M16C Microcontrollers. doi:10.13140/2.1.2701.9846
11. Kolhare, Nilima. R. and Nitin I.Bhopale. Porting & Implementation of features of uC/OS II RTOS on Arm7 controller LPC 2148 with different IPC mechanisms. International journal of engineering research and technology 1 (2012).
12. Y. Zhang, F. Lu and X. Kong. VxWorks porting based on MPC8313E hardware platform. 2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering, 2010, pp. 246-249, doi: 10.1109/CMCE.2010.5610170.
13. Porting µC/OS-II https://micrium.atlassian.net/wiki/spaces/osiidoc/pages/163858/Porting+C+OS-II
14. H. Hsu and C. Hsueh. FreeRTOS Porting on x86 Platform. 2016 International Computer Symposium (ICS), 2016, pp. 120-123, doi: 10.1109/ICS.2016.0032.
15. IEEE Standard for a Real-Time Operating System (RTOS) for Small-Scale Embedded Systems. (n.d.). doi:10.1109/ieeestd.2018.84456
16. Gomes, R. M., & Baunach, M. (2018). A Model-Based Concept for RTOS Portability. 2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA). doi:10.1109/aiccsa.2018.861286
17. D. Comer. Operating System Design - The Xinu Approach, Second Edition CRC Press, 2015. ISBN 9781498712439.
18. Rafael Zurita. Programación de Sistemas Embebidos. Materia del último año de la carrera Lic. en Ciencias de la Computación. Facultad de Informática, Universidad Nacional del Comahue. http://se.fi.uncoma.edu.ar/pse2020/
19. Candelaria Alvarez. Diseño e implementación de un sistema embebido de navegación por estima para la localización de robots móviles en ambientes de interiores, Abril 2022. Tesis de grado de la carrera Lic. en Ciencias de la Computación. Universidad Nacional del Comahue
20. Rafael Ignacio Zurita. Página web oficial del port de Xinu para arquitectura AVR. http://se.fi.uncoma.edu.ar/xinu-avr/