Centro de Investigación y de Estudios
Avanzados
del Instituto Politécnico Nacional

Departamento de Ingeniería Eléctrica

Sección de Computación

Desarrollo de una interfaz de programación en Matlab para la simulación de tareas de control sobre un kernel de tiempo real

Tesis que presenta

Roberto Manuel Linares Zamora

para obtener el Grado de

Maestro en Ciencias en la Especialidad

de Ingeniería Eléctrica

Directores de la Tesis

Dr. Pedro Mejía Álvarez
Dr. Albero Soria López

México, D.F. **Junio 2006**

Dedicatorias...

A mis padres

Qué puedo decirles sino mil gracias por ser los mejores ejemplos que he podido tener. Les doy gracias por todos sus cuidados y porque siempre creyeron en mi. Son las mejores personas que conozco y les dedico este trabajo porque es algo que sin ustedes no hubiera podido ser. No puedo expresar aquí mismo todo lo que siento por ustedes, así que solamente digo GRACIAS *por todo su apoyo y amor. Este trabajo es suyo.*

A mis hermanos Marisela e Isaac

Les dedico este trabajo porque su éxito me ha servido como inspiración y me ha alentado a seguir superándome día a día. Y aunque se que están lejos, sé que siempre se preocupan por mi y están conmigo. Los quiero mucho.

A mi Chiquininita

A ti peque, gracias por todo tu apoyo, tu confianza y por creer en mi. Por ser el aliento que me da ánimo para seguir siempre adelante y por tus consejos. Gracias por ayudarme a hacerle caso a mis sentimientos. Eres y siempre serás parte muy importante de mi vida... je t'aime beaucoup

Agradecimientos

Agradecimiento especial para mis asesores Dr. Pedro Mejía Álvarez y Dr. Alberto Soria López, por darme la oportunidad de participar en su proyecto, por su preocupación, por su entusiasmo y por quienes sin su ayuda no podría haber terminado este trabajo. Al Dr. Rubén A. Garrido y al Dr. Adriano de Luca por sus consejos en la escritura de este documento.

A los doctores y el personal administrativo de la Sección de Computación del Departamento de Ingeniería Eléctrica por sus excelentes clases. En especial al Dr. Pedro Mejía por su enseñanzas en sistemas de tiempo real, así como a Sofia Reza, al Dr. Buenabad, Dr. Sergio Víctor Chapa, Dr. Adriano de Luca, Dr. Francisco Rodríguez Henríquez y Dr. Arturo Díaz por sus consejos en clase.

A CONACyT por el apoyo económico que me brindó durante el transcurso de la maestría. A servicios escolares y a la biblioteca de Ingeniería Eléctrica por los servicios y ayuda que me brindaron cuando lo necesité.

A mis padres, por la educación que inculcaron en mi. Gracias por todo su cariño, por el apoyo incondicional que me dieron en todo momento y por enseñarme a tener fortaleza para seguir adelante, sobreponiéndome siempre a las adversidades que la vida presenta. Gracias por guiarme por el camino de la educación. A mis hermanos y sobrinas, que siempre me brindan ánimo y apoyo desde lejos.

A mis amigos Efrén, Dennis, Mario, Paco, Pablo y Samuel. Gracias a todos ustedes

por su amistad, ya que un verdadero amigo es alguien que te conoce tal como eres, comprende dónde has estado, te acompaña en tus logros y tus fracasos, celebra tus alegrías, comparte tu dolor y jamás te juzga por tus errores. Todo eso han sido han sido ustedes para mi, por lo que no me queda más que decirles GRACIAS POR TODO.

Resumen

La mayoría de los sistemas de control son desarrollados en un microprocesador usando un kernel de tiempo real o un sistema operativo de tiempo real. En la mayor parte de los casos el algoritmo de control es ejecutado periódicamente, enfocándose solamente en el dominio del problema sin preocuparse de cómo la planificación afecta al sistema de control o como la ejecución concurrente de los procesos de cómputo afecta el comportamiento y la estabilidad del sistema de control. Actualmente, existen aplicaciones que auxilian en el diseño de sistemas de control como Matlab/Simulink, LabVIEW, Wincon, etc. Estas aplicaciones presentan una serie de herramientas (funciones, bloques y modelos) desarrolladas para la simulación del sistema de control. Sin embargo, son pocas las herramientas que ayudan al diseño de sistemas de control que contemplen restricciones de tiempo, y que además garanticen la ejecución concurrente de los algoritmos de control , o bien, que interactúen con otros ambientes de software externos, que permitan aplicar políticas de planificación y utilizar una interfaz gráfica para la visualización de los procesos.

Este trabajo tesis, presenta el desarrollo de una interfaz de programación en Matlab para la ejecución de tareas de control sobre un kernel de tiempo real. El sistema se compone de tres etapas: Una etapa de generación de código (*módulo generador de tareas*), otra etapa de manejo de código (*interfaz de programación*) y una última etapa de ejecución (*micro-kernel de aplicación*). En la primera etapa del sistema, se implementa un módulo generador de tareas, que construye código a partir de un diagrama de Simulink. Este módulo utiliza la opción ERT de la herramienta RTW (*Real Time WorkShop*) de Matlab/Simulink para la generación de código. En la segunda etapa, se desarrolla la interfaz de programación, donde se realiza la conexión funcional entre Matlab/Simulink y el micro-kernel. La interfaz es un programa escrito en lenguaje C que contiene las dependencias y funciones que necesitan los códigos generados por la herramienta RTW de los modelos Simulink. Esta interfaz ejecuta

el código dentro del micro-kernel. En la tercera etapa, se utiliza un micro-kernel de tiempo real, donde el código generado por el módulo se ejecuta como un proceso dentro del micro-kernel, comportándose como una tarea de tiempo real. El micro-kernel puede manejar más de un proceso o tarea y su función principal es la de permitir la creación y ejecución concurrente de varios procesos. El micro-kernel que se utiliza se ejecuta sobre el sistema operativo MS-DOS.

Este sistema puede ser utilizado para el diseño e implementación de sistemas de control con características de tiempo real, integrando así la funcionalidad de la herramienta Matlab/Simulink/*Embedded Real Time* Worshop con otra aplicación fuera del ambiente de Matlab(microkernel de tiempo real).

Las pruebas del código se realizaron en un micro-kernel de tiempo real y los modelos Simulink utilizados, se generaron con el fin de construir modelos sencillos con los bloques más utilizados de Simulink y obtener la descripción en código de los bloques empleados. El modelo de tareas de control se ejecuta dentro del micro-kernel sin recursos compartidos, teniendo la posibilidad de ejecutar varios procesos de forma concurrente. El método de planificación utilizado en la aplicación es uno de prioridad fija, en particular se emplea la política de RR (*Round Robin*).

Abstract

Most control systems are implemented on a microprocessor with a real time kernel or a real time operating system. In most cases the control algorithm is executed periodically focusing on the control problem and no attention is paid to how the planning affects the control loop or how the concurrent execution of the computation process affects the behavior and stability of the control system. Currently, a number of applications for control algorithms development are available such as Matlab/Simulink, LabVIEW, Wincon, etc., that help with the design of control systems. These applications offer a series of tools (functions, blocks, and models) developed for the simulation of control processes. However, there are few tools that help in the design of control systems which take into account time restrictions in addition to guaranteeing the concurrent execution of the control algorithms, or more specifically, that interact with other external software environments which allow the application of planning policies and the use of a graphical interface for visualization of the processes.

In this work, we present the development of a Matlab programming interface for the execution of control tasks on a real time kernel. The system is composed of three stages; a code generation stage (code generation module), a code handling stage (programming interface), and an execution stage (micro-kernel application). A task generator module implemented in the first stage of the system constructs code from a Simulink diagram. This module utilizes the ERT option of the Matlab/Simulink RTW tool for code generation. The programming interface, where the functional connection between Matlab/Simulink and the microkernel takes place, is developed in the second stage. The interface is a C program that contains the dependencies and functions need by the code generated by the RTW tool from the Simulink models. This interface executes the code within the micro-kernel. In the third stage, a real time microkernel is used, where the code generated by the module is executed as a process within the microkernel, behaving like a real time task. The micro-kernel can

handle more than one process or task; its principal function is to allow the concurrent creation and execution of several processes. The micro-kernel used is executed over a DOS operating system.

This system can be used for the design and implementation of control systems with real time characteristics, therefore integrating the functionality of the MATLAB/ Simulink/ Real Time Worshop tool with other applications outside the MATLAB environment (real time micro-kernel).

The code tests took place within the real time microkernel and the Simulink models used, they were generated with the purpose of constructing simple models with the most used blocks of Simulink to obtain the code description of the blocks used. The task control model is executed within the micro-kernel without shared resources, having the possibility of concurrently running several processes. The planning method used in the application is one of fixed priority, in particular we used the RR (Round Robin) policy.

Índice general

Índice de figuras

Lista de Tablas

Capítulo 1

Sistemas de Tiempo Real y Sistemas de Control

1.1. Introducción

Los sistemas de control con características de tiempo real, generalmente poseen requerimientos críticos en términos de comportamiento temporal, estabilidad y seguridad. Este tipo de aplicaciones puede encontrarse en campos tan diversos como la robótica, automatización de procesos de manufactura, la industria aeroespacial o en sistemas de transporte (control de automóviles, ferrocarriles, etc.)[1]. Incluso en el campo de control de procesos industriales existen aplicaciones en donde un conjunto de subsistemas deben cooperar y cumplir ciertos requisitos temporales o de seguridad.

La mayor parte de estos sistemas de control por computadora son sistemas empotrados o embebidos, donde la computadora es un componente dentro de un sistema más grande de ingeniería. Estas leyes de control son a menudo implementados como una o varias tareas sobre un microprocesador que usa un sistema operativo de tiempo real. En la mayoría de los casos el microprocesador ejecuta otras tareas concurrentes que realizan funciones diferentes, las cuales podrían ser por ejemplo, la comunicación entre procesos o la utilización de interfaces de usuario [2]. Por lo que un Sistema Operativo típicamente, usa la multiprogramación, el cambio de contexto, y las políticas de planificación para ejecutar concurrentemente las diferentes tareas sobre un solo procesador.

En el diseño de sistemas de control con características de tiempo real es necesario contemplar el uso de alternativas de software eficientes y flexibles para complementar todas las etapas del diseño [3]. En este tipo de aplicaciones, aspectos como el período del proceso, su prioridad y las políticas de planificación deben ser considerados durante el diseño. Sin embargo, las técnicas, herramientas y arquitecturas existentes utilizadas para el diseño y desarrollo de sistemas de control, fueron diseñadas para software de propósito general y no son útiles para modelar o simular sistemas de control con características de tiempo real.

Por esta razón, ha surgido la necesidad de generar herramientas para el diseño y simulación de sistemas de control con características de tiempo real para tratar adecuadamente todas las fases del diseño (desde la especificación, análisis y simulación, hasta la generación de código de la aplicación). Esta problemática es muy amplia y ha dado lugar a diversas áreas de investigación para dar respuesta a las necesidades actuales y para producir sistemas más fiables, potentes y cuya complejidad sea manejable.

El uso de herramientas como MatLab/Simulink, se ha convertido en una práctica común para el diseño de muchas aplicaciones de control. Simulink es una herramienta útil para el desarrollo y simulación de sistemas de control [4]. Sin embargo, estas herramientas sólo se concentran en el problema de control para un solo proceso, sin considerar la ejecución concurrente de varios procesos y su cumplimiento de plazos temporales ni tampoco incluyen políticas de planificación de tiempo real.

La ausencia de herramientas que permitan llevar a cabo un diseño formal y sistemático para construir y simular un sistema de control con características de tiempo real, llevan a la construcción de sistemas con errores en su funcionamiento y en su comportamiento temporal. Es por esto que en esta tesis se plantea el desarrollo de una herramienta que permita simular y modelar sistemas de control con características de tiempo real.

En esta tesis, se propone el desarrollo de una interfaz de programación en MatLab para la simulación de tareas de control sobre un kernel de tiempo real. Mediante esta herramienta se genera de forma visual el código de los modelos generados en MatLab/Simulink. El código generado se incluye como una tarea más dentro de un micro-kernel de tiempo real.

En el diseño se plantea la generación de código de C a partir de un diagrama de simulación de Simulink y utilizarlo a través de una API[1] en un micro-kernel de tiempo real como proceso o tarea control. El modelo de tareas de control se ejecuta como un proceso más del microkernel, sin recursos compartidos, teniendo la posibilidad de ejecutar varios procesos de forma concurrente. El método de planificación utilizado en la aplicación es de prioridad fija, en particular se implementa la política RR (*Round Robin*); pero también se contempla el uso de prioridades dinámicas como RM (*Rate Monotonic*) y EDF (*Earliest Deadline Fisrt*).

1.2. Motivación

La mayoría de los sistemas de control de tiempo real están compuestos de dos partes: el proceso controlado o ambiente, y la computadora que efectúa el control del proceso [2]. La computadora interactúa con el ambiente, basándose en la información que percibe de varios sensores. Esta interacción debe permitir a la computadora percibir correctamente el estado actual del ambiente. De otra forma, los efectos al sistema podrán ser muy severos, como por ejemplo la desestabilización del sistema, o bien, que los módulos que lo componen no funcionen correctamente. Por lo tanto, es necesaria una monitorización periódica del ambiente y un procesamiento a tiempo de esta información.

En el lazo de realimentación de un sistema de control, la computadora realiza un conjunto de funciones en forma periódica, utilizando las entradas de control (por ejemplo la lectura de sensores), y ejecuta una secuencia de instrucciones que añaden al proceso controlado un retardo, conocido como retardo del tiempo de cómputo. Este es un retardo adicional al retardo propio del proceso el cual es necesario tomar en cuenta debido a que puede afectar en forma adversa al desempeño del sistema de control [5].

Cuando un sistema de control demanda de restricciones de tiempo real se añaden otros factores dentro del lazo de control que afectan al sistema. Estos factores son la ejecución estable de los procesos cuando se manejan como tareas concurrentes y el cumplimiento de tiempos de estas tareas [3]. Esto complica el comportamiento del

[1] Ver definición en lista de acrónimos

sistema de control, debido a que hace necesaria una monitorización periódica del ambiente, resultando en un aumento en el tiempo de ejecución de las tareas; ocasionando que estas excedan su plazo de respuesta.

Cuando un componente falla o existe una alteración del ambiente (como una elevación de temperatura o una interferencia electromagnética), el tiempo dedicado a la detección, localización, y recuperación del fallo deben ser añadidos al tiempo de ejecución del programa de control, lo cual incrementa significativamente el retardo del tiempo de cómputo del sistema de control.

Este retardo pudiera causar que se excediera el plazo de respuesta previamente establecido, provocando una degradación en el comportamiento del sistema. Por lo tanto, en un sistema de control es necesario analizar los efectos del retardo del tiempo de cómputo y el comportamiento de los procesos, cuando estos se ejecutan de manera concurrente [6].

Bajo este nuevo esquema, el sistema de control de tiempo real debe:

- Realizar sus funciones dentro de plazos de respuesta, ejecutandose dentro de su periódo, tal que se garantice la estabilidad en lazo cerrado.
- Planificar todos los procesos de control que conforman una aplicación y garantizar en todos ellos el cumplimiento de plazos de respuesta.
- Proporcionar robustez, es decir poca sensibilidad de las variables del proceso respecto a variaciones en los parámetros del mismo y permitir la ejeción del algoritmo de control durante la operación.
- Garantizar una operación en forma confiable de los procesos ejecutados concurrentemente.

Así, el objetivo más importante de un sistema de control en tiempo real es la ejecución de sus procesos a tiempo sin afectar su estabilidad. Para satisfacer los requerimientos temporales de la aplicación su diseño debe ser capáz de analizar las características de las tareas de tiempo real con el fin de garantizar su planificabilidad, es decir, que verifique el cumplimiento de los plazos de todas las tareas de tiempo real.

Este nuevo enfoque, el cual unifica la Teoría de Control y la Teoría de Sistemas en Tiempo Real, genera posibilidades para el desarrollo de aplicaciones integradas [4].

Para abordar el diseño de este tipo de sistemas resulta fundamental el disponer de herramientas que abarquen todas las fases del diseño (desde la especificación, análisis y simulación, hasta la generación de código de la aplicación).[7]

Este problema de actualidad ha motivado el presente trabajo de tesis, que consiste en el desarrollo de un módulo generador de código simple a partir de un diagrama de Simulink y una interfaz de programación (API) de manejo de código. El código generado se utiliza como una aplicación dentro de un micro-kernel de tiempo real por medio de la interfaz, comportándose como un proceso más del mismo; esta herramienta puede ser utilizada para el diseño e implementación de sistemas de control, integrando así la funcionalidad de la herramienta MatLab/Simulink/*Embedded Real Time Worshop* con otra aplicación fuera del ambiente de MatLab (micro-kernel de tiempo real).

1.3. Trabajo Relacionado

Las aplicaciones de procesos industriales, son usualmente desarrolladas en dos fases: el diseño de las leyes de control y la implementación del sistema de tiempo real. Esta combinación de diseño cooperativo crea la posibilidad de desarrollar sistemas mas integrados, flexibles y dinámicos [8]. Tradicionalmente, estas dos fases han sido desarrolladas de forma separada.

Recientemente, algunos trabajos de investigación se han enfocado en la necesidad de integrar el diseño de control y su implementación con sistemas de planificación. Uno de los primeros trabajos de este nuevo enfoque, donde el control y el tiempo real se encuentran integrados, fué propuesto por [Seto et al., 1996][1]. En este trabajo, la taza de muestreo de un conjunto de controladores que comparten el mismo CPU, es calculada usando las métricas de control estándar. Otro trabajo relacionado con el diseño cooperativo entre control y la planificación en tiempo real se reporta en [Ryu et al, 1997] [9], en donde el desempeño del control es especificado en términos del sobretiro (*overshoot*), el tiempo de subida y el tiempo de respuesta. Estos parámetros de desempeño se muestran como funciones del período de muestreo y de la latencia de entrada y salida. Esta integración de enfoques requiere de nuevas herramientas para su análisis, diseño, simulación e implementación.

Con la necesidad de crear nuevas aplicaciones para sistemas de control en tiempo real, se generaron numerosas herramientas que soportan la simulación de sistemas de

control (Simulink) o la simulación de la planificación en tiempo real (STRESS [Audsley et al., 1994] y DRTSS [Storch and Liu, 1996]). Sin embargo, pocas herramientas soportan la simulación de diseños cooperativos de control y planificación en tiempo real.

Una primer herramienta para este fin, fue un prototipo desarrollado en MatLab presentado por [Eker, 1999] [1], el cual ejecuta tareas concurrentes solo de forma simulada. Otra herramienta desarrollada fue RTSIM, la cuál es un simulador de planificación en tiempo real, desarrollado en C++. Esta herramienta fué extendida a un módulo numérico que soporta la simulación de sistemas contínuos [Palopoli et al., 2000] [2]. Sin embargo, esta herramienta, no contempla un ambiente gráfico del modelo de control y su manejo de tiempos de ejecución es limitado.

Una de las aportaciones más importantes en el desarrollo de herramientas para la simulación de sistemas en tiempo real, fue la desarrollada por [A. Cervin, 2003] [9], donde la Teoría de Control y la Teoría de Sistemas en Tiempo Real se conjuntan para crear las herramientas *TrueTime* y *Jitterbug*. Estas herramientas permiten simular un sistema de control con tareas planificadas en tiempo real y son usadas para generar una primera vista o escenario del diseño del sistema de control en tiempo real. Estas herramientas permiten determinar qué tanto afecta el control a la planificación de tareas.

En el trabajo de [G. Quaranta, 2003] [8], se introduce un ambiente de tiempo real de Linux que puede ser usado en el diseño e implementación de sistemas de control digital. En este trabajo se integra la funcionalidad de MatLab/Simulink con la herramienta RTW (Real Time Workshop), ejecutando dentro del ambiente que provee el Sistema operativo en RTAI (extensión pública del kernel de tiempo real de Linux). En esta aplicación se emplean las ventajas de la herramienta RTW de MatLab/Simulink para modelar las características del RTAI en cuanto a su modelo de planificación y ejecución de tareas. Este trabajo todavía no ha sido terminado, pero propone una alta confiabilidad en el diseño de sistemas de control digital.

Asimismo, existen Sistemas Operativos como Tornado y xPC [10], los cuales permiten ejecutar código generado a partir del RTW de MatLab/Simulink. Estas herramientas generan código ejecutable el cual puede ser portable y ejecutable en cualquier plataforma, utilizando el generador de código de RTW. Otra herramienta del mismo tipo es Wincon [11], la cual es una aplicación de prototipado rápido desarrollada para trabajar con MatLab. Esta herramienta sirve para probar modelos de control en

tiempo real, permitiendo una facíl implementación del modelo; sin embargo, es una herramienta separada de MatLab, lo cual incrementa su costo y aún no contempla el manejo de varios modelos de control en forma simultánea, lo que limita el alcance de esta herramienta.

Otra de las aplicaciones de modelado de sistemas de control es LabVIEW. Esta herramienta es un entorno de programación destinado al desarrollo de aplicaciones, similar a los sistemas de desarrollo comerciales que utilizan el lenguaje C o BASIC. Sin embargo, LabVIEW se diferencia de dichos programas en un importante aspecto: los citados lenguajes de programación se basan en líneas de texto para crear el código fuente del programa, mientras que LabVIEW emplea la programación gráfica o lenguaje G para crear programas basados en diagramas de bloques.

En la caso de la aplicación desarrollada en esta tesis, la herramienta que se presenta puede ser utilizada para el diseño e implementación de sistemas de control, integrando así la funcionalidad de la herramienta MatLab/Simulink/*Embedded Real Time Worshop* con otra aplicación fuera del ambiente de MatLab (micro-kernel de tiempo real). Con esto se tiene la posibilidad de manejar varios modelos y observar su comportamiento; característica que en los otros trabajos anteriormente mencionados era un limitante o bien, es complicado el uso de varios porcesos simultaneos. Aunado a esto, la aplicación desarrollada es fácil de usar y sirve de base para investigaciones de tipo aplicado, ya que no es necesario implementar en un lenguaje de tiempo real los diseños obtenidos de Simulink (un proceso tedioso y de mucho cuidado sobre todo en el caso de sistemas que contienen subsistemas). En su lugar se generara automáticamente el código listo para una aplicación externa, utilizando el ERT del *Real Time WorkShop*.

1.4. Objetivos de la Tesis

El objetivo principal de este trabajo de tesis consiste en la implementación de un programa de generación de código C a partir de un diagrama de simulación de control de Simulink. Se pretende que este código se incluya como una tarea de tiempo real dentro de un micro-kernel, a través de una interfaz de aplicación. Los objetivos específicos se describen a continuación.

1.4.1. Objetivos Específicos

Para llevar a cabo el objetivo general de esta tesis, el trabajo se ha dividido en las siguientes actividades:

1. Desarrollar una aplicación de manejo de código entre MatLab/Simulink y un micro-kernel de tiempo real.

2. Simular procesos en el micro-kernel de tiempo real utilizando políticas de planificación, en particular las políticas de asignación de prioridades estáticas (*Round Robin*) y dimánicas (*Earliest Deadline First*).

3. Integrar los elementos anteriormente descritos en el micro-kernel de tiempo real y desarrollar una aplicación de análisis y validación que permita diseñar sistemáticamente sistemas de control de tiempo real, utilizando la interfaz gráfica.

1.5. Sistemas de Tiempo Real

En un sistema de control de tiempo real, se pueden distinguir dos partes principales: el proceso controlado o ambiente, y la computadora que efectúa el control del proceso. El sistema de tiempo real debe de integrar restricciones de tiempos y mecanismos de planificación; por lo que un sistema de tiempo real está compuesto comúnmente por tareas que se ejecutan concurrentemente y un estricto cumplimiento de tiempos o plazos de respuesta, donde las tareas son la unidad de cómputo a ser planificada y ejecutada y los procesos o tareas proporcionan un servicio o una cierta cantidad de datos. Estas tareas proporcionan sus resultados a tiempo, de acuerdo a un plazo de respuesta definido por su aplicación, por lo que la alteración de estas restricciones de tiempos en sistemas con altos requisitos de seguridad, puede resultar en graves pérdidas económicas, o de vidas humanas.

Los sistemas de tiempo real se definen como aquellos sistemas en los que su correcto funcionamiento no sólo depende de los resultados lógicos, sino también del momento en que se producen dichos resultados [2]. Esta interpretación se debe a que estos sistemas interactúan con un ambiente físico a controlar. Sus entradas reflejan eventos ocurridos en su ambiente, y sus salidas se traducen en efectos sobre dicho ambiente. Los resultados deben producirse en los momentos en que aún tengan válidez dentro del ambiente a controlar. Por esta razón, los sistemas de tiempo real controlan un ambiente que tiene restricciones de tiempo bien definidas, es por ello, que se vuelven

más complejos y por tanto demandan una alta confiabilidad, con resultados correctos, predecibles y a tiempo [6].

Actualmente, los sistemas de tiempo real abarcan un amplio rango de aplicaciones, como son los sistemas de control de procesos, sistemas médicos, sistemas de manufactura, sistemas de telefonía móvil, sistemas multimedia, robótica, por nombrar solo unos cuantos.

1.5.1. Elementos de un Sistema de Tiempo Real

Un sistema de tiempo real consiste principalmente de computadoras, y elementos externos con los cuales el software de la computadora debe interactuar simultáneamente. En la Figura 1.1 se muestran los elementos generales de un sistema de tiempo real donde el objetivo principal es controlar un ambiente. En dicha Figura se distinguen los siguientes elementos:

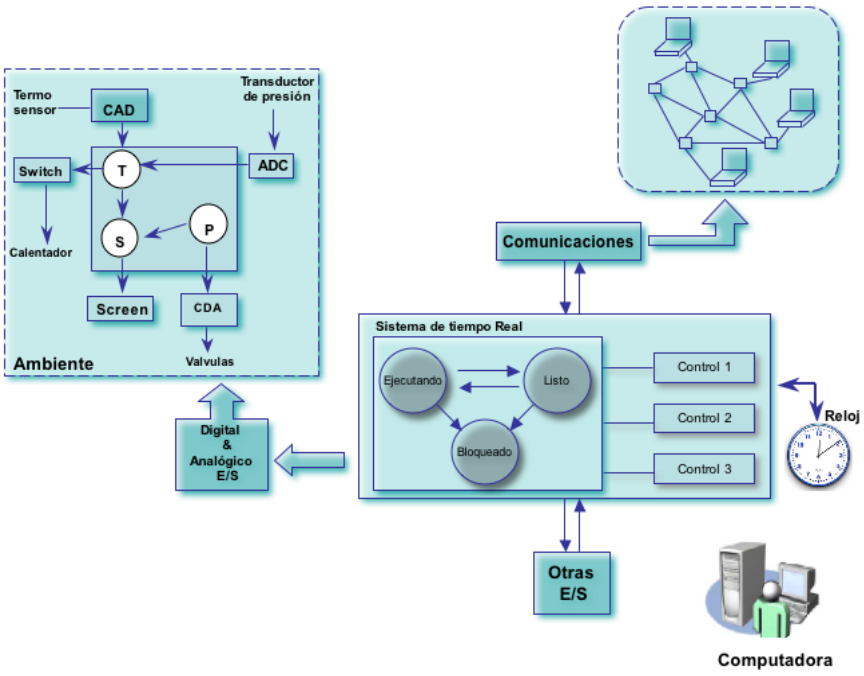

Figura 1.1: Elementos de un Sistema de Tiempo Real.

- **Ambiente**.
  El término ambiente de la Figura 1.1 se refiere al sistema controlado. Por ejemplo, un motor, un sistema de manufactura, un robot, o un avión, etc. El estado del ambiente (entorno físico) es supervisado por los sensores y puede cambiar por los actuadores.

- **Convertidores**.
  Los convertidores análogico-digital convierten las señales generadas por el ambiente (analógicas) a una serie de datos que la computadora interpreta (digitales).

- **Reloj de Tiempo Real**.
  El reloj de tiempo real permite al sistema contar el tiempo en que se ejecutan acciones. De la misma forma, el reloj de tiempo real permite al sistema configurar los tiempos para la planificación de las tareas. Mediante el reloj de tiempo real es posible configurar interrupciones para controlar dispositivos externos, recibir y sincronizar señales de comunicación, y monitorear el cumplimiento de los requerimientos temporales de las tareas del sistema. Sin el reloj de tiempo real no sería posible configurar los tiempos de ejecución de las tareas de tiempo real, y por tanto la planificación del sistema, ni tampoco sería posible saber si las tareas cumplen con sus restricciones temporales.

  En una aplicación que involucra a varias computadoras es necesario que los relojes de tiempo real se encuentren sincronizados, a fin de que todos ellos lleven la misma cuenta de tiempo.

- **Software de Tiempo Real**.
  El software de tiempo real esta compuesto de un sistema operativo (o kernel) y de tareas las cuales son planificadas por el kernel. La estructura del kernel esta compuesta de manejadores de tiempo, de tareas, de memoria, de dispositivos, y de todos los recursos del sistema de cómputo.

  Las tareas del sistema (o procesos) son las entidades de software que permiten controlar el medio ambiente. Cada tarea es un procedimiento de software que se ejecuta de forma continua, sin embargo, la ejecucion concurrente de las tareas es controlada por el manejador de procesos del kernel.

- **Comunicaciones**.
  En el sistema de tiempo real pueden existir distintas computadoras que interactúan entre sí. Entre ellas existe un medio físico de comunicación (hardware) y

un protocolo de comunicaciones (software) que les permite enlazarse, compartir información y sincronizar su ejecución.

Mediante la comunicación es posible compartir recursos de hardware (por ejemplo, dispositivos de entrada y salida), y mejorar la eficiencia de aplicaciones mediante la distribución del cómputo, y lograr mayor rapidez de ejecución.

- **Otras E/S (entradas y salidas)**.
  Un sistema de tiempo real tiene principalmente como entrada el comportamiento del sistema físico controlado. Sin embargo, existen otras entradas y salidas principalmente aquellas con las que interactúan con el usuario, como son el teclado, el ratón, y otros dispositivos de interacción con el usuario.

1.6. Estados de los Procesos

Un **proceso** se define como un programa en ejecución, tal que su ejecución procede de manera secuencial. En cualquier instante, solo una instrucción se estará ejecutando por proceso en el CPU. Un proceso está representado por su código, sus datos y su pila o *stack*. El código del proceso es ejecutado en el CPU en forma concurrente con otros procesos, lo cual quiere decir que el CPU es repartido entre varios procesos. Al ejecutar a varios procesos en forma concurrente es necesario saber que cantidad máxima de tiempo de CPU ó (*quantum de tiempo*), se le asigna a cada proceso. Al finalizar este *quantum* el planificador asignará el CPU al proceso con mayor prioridad en el sistema. Al procedimiento de cambio de CPU se le conoce como cambio de contexto. El procedimiento de **cambio de contexto** es el siguiente:

- Interrumpir la ejecución del proceso en ejecución.
- Guardar el lugar de ejecución mediante el contador del programa, o (*Instruction Pointer*, IP) del proceso, sus datos (*Data Segment*, DS) y su stack (*Stack Segment*, SS).
- Buscar el siguiente proceso con mayor prioridad en el sistema.
- Asignar el CPU al proceso con mayor prioridad.

El detalle de la ejecución de los procesos se guarda en una estructura de datos conocida como PCB (*Process Control Block*). En esta estructura de datos se guarda, el estado del proceso, la dirección y tamaño asignado de su código, datos y stack, los

recursos que tiene asignados, y posibles manejadores de excepción asignados al proceso. Debemos considerar que, un programa por sí sólo no es un proceso; un programa es una entidad *pasiva*, como el contenido de un archivo guardado en disco, mientras que un proceso es una entidad *activa*, con el contador de programa especificando la siguiente instrucción que se ejecutará, y un conjunto de recursos asociados. Aunque podría haber dos procesos asociados, al mismo programa, de todas maneras se toman como dos secuencias en ejecución distintas. Por ejemplo, varios usuarios podrían estar ejecutando copias del programa de correo, o un mismo usuario podrá invocar muchas copias del programa editor. Cada una de éstas es un proceso aparte y, aunque las secciones de texto sean equivalentes, las secciones de datos variarán. También es común tener un proceso que engendra muchos procesos durante su ejecución.

A medida que un proceso se ejecuta, cambia de *estado*. El estado de un proceso está definido en parte por la actividad actual de ese proceso. Cada proceso puede estar en uno de los siguientes estados:

- Nuevo: El proceso esta en disco y no se le han asignado recursos. En este estado, el proceso no puede ejecutarse.
- En ejecución: Se le ha asignado el CPU y estará en ejecución hasta que se termine su *quantum* de tiempo.
- En espera: El proceso está esperando que ocurra algún evento (como la terminación de una operación de entrada o salida o la recepción de una señal), o que transcurra un lapso de tiempo.
- Listo: El proceso esta listo para ejecución y esperando a que se le asigne el procesador.
- Terminado: El proceso terminó su ejecución. Se le retiran los recursos y no puede competir más por tiempo de ejecución.

Es importante tener en cuenta que sólo un proceso puede estar *ejecutándose* en cualquier procesador en un instante dado, pero muchos procesos pueden estar *listos* y *esperando*. El diagrama de estados correspondiente se presenta en la figura 1.2. Para poder asignar tiempo de CPU a los procesos, debemos especificar primero un orden de ejecución. A este procedimiento se le conoce como planificación. Posteriormente en este capítulo, se discutirá distintas políticas de planificación para manejo concurrente de procesos.

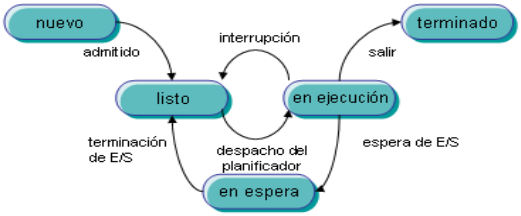

Figura 1.2: Diagrama de estados de un proceso.

1.7. Clasificación de los Algoritmos de Planificación

De entre la gran variedad de algoritmos propuestos para la planificación de tareas en tiempo real, podemos identificar las siguientes clases:

- **Planificación expulsiva**. La tarea en ejecución puede ser interrumpida en cualquier momento para asignar al procesador a otra tarea, de acuerdo a una política de planificación predefinida.
- **Planificación no expulsiva**. Una tarea una vez iniciada, es ejecutada por el procesador hasta que termine su actividad. En este caso, todas las decisiones de planificación son tomadas cuando una tarea inicia o termina su ejecución.
- **Planificación estática**. Los algoritmos de planificación estáticos son aquellos en los cuales las decisiones de planificación están basados en parámetros fijos, los cuales son asignados a las tareas antes de su activación.
- **Planificación dinámica**. Los algoritmos de planificación dinámicos son aquellos en los cuales las decisiones de planificación están basadas en parámetros dinámicos que pueden cambiar durante la ejecución del sistema.
- **Planificación fuera de línea**. Decimos que un algoritmo de planificación es usado fuera de línea si es ejecutado sobre el conjunto completo de tareas antes de la ejecución actual de las tareas. La planificación generada en esta forma, es almacenado en una tabla y después ejecutado por un despachador.
- **Planificación en línea**. Decimos que un algoritmo de planificación es usado en línea si las decisiones de planificación son tomadas a tiempo de ejecución cada vez que una nueva tarea llega o cuando una tarea termina su ejecución.

- **Planificación óptima**. Se dice que un algoritmo es óptimo si minimiza algunas funciones de costo definidas sobre el conjunto de tareas. Cuando ninguna función de costo es definida y lo único concerniente es alcanzar una planificación factible, entonces se dice que un algoritmo de planificación es óptimo si este encuentra una solución de planificación que no puede ser contradecida por ningún otro algoritmo de planificación. La solución de planificación indica si un conjunto de tareas es factible o no factible (cumple o no con sus plazos de respuesta.

- **Planificación heurística**. Un algoritmo heurístico encuentra una solución aproximada que intenta estar cerca de la solución óptima.

1.7.1. Métodos de Planificación

Dentro de los métodos de planificación se encuentran:

Planificación basada en el reloj (basada en el tiempo).

En este método, el plan de ejecución se calcula fuera de línea y se basa en el conocimiento de los tiempos de inicio y de cómputo de todos las tareas o *jobs*. El plan de ejecución está en una tabla y no es concurrente.

Planificación Round Robin.

Este es uno de los algoritmos más sencillos y equitativos en el reparto de recursos entre los procesos, muy válido para entornos de tiempo compartido. Cada proceso tiene asignado un intervalo de tiempo de ejecución, llamado *quantum*. Si el proceso agota su *quantum* de tiempo, se elige a otro proceso para ocupar el recurso. Si el proceso se bloquea o termina antes de agotar su *quantum* también se alterna el uso del recurso.

Este algoritmo presupone la existencia de un reloj en el sistema. Un reloj es un dispositivo que genera periódicamente interrupciones. Esto es muy importante, pues garantiza que el sistema operativo (en concreto la rutina de servicio de interrupción del reloj) tome el mando del procesador periódicamente. El *quantum* de un proceso equivale a un número fijo de pulsos o ciclos de reloj. Al ocurrir una interrupción de reloj que coincide con la agotación del *quantum* se llama al despachador (*dispatcher*) para activar otro proceso.

Planificación basada en prioridades.

En este método las prioridades las asigna el algoritmo de planificación. La tarea con mayor prioridad se ejecuta en cualquier instante. Dentro de esta planificación se encuentran la planificación por prioridades fijas y la planificación por prioridades dinámicas.

- **Planificación por prioridades fijas.**
  El algoritmo de planificación RM (*Rate Monotonic*) se basa en una simple regla que asigna las prioridades de las tareas de acuerdo a su frecuencia. Las tareas con períodos más cortos tendrán prioridades más altas. Debido a que los periodos son constantes, RM es una asignación de prioridades fijas o estáticas. Las prioridades son asignadas a las tareas antes de su ejecución y no cambian durante la ejecución. Por otra parte, RM es una política de planificación desalojable ya que permite que las tareas en ejecución puedan ser desalojadas por tareas con más alta prioridad.

  Liu y Layland [14] demostraron en 1973 que RM es óptimo entre todos los métodos de asignación de prioridades fijas en el sentido que ningún otro algoritmo de prioridades fijas puede planificar un conjunto de tareas que no puedan ser planificadas por RM.

  El límite superior mínimo del factor de utilización bajo el algoritmo de planificación RM para un conjunto de tareas n es:

  $$U_{lsm} = n(2^{\frac{1}{2}} - 1) \tag{1.1}$$

  Este valor decrece con n, para valores altos de n, el límite superior converge en:

  $$U_{lsm} = ln2 \approx 0{,}69 \tag{1.2}$$

- **Planificación por prioridades dinámicas.**
  El algoritmo de planificación EDF (*Earliest Deadline First*) es un planificador dinámico que selecciona tareas de acuerdo a sus plazos de respuesta absolutos. Las tareas con plazos más cercanos tendrán mayor prioridad.

  EDF es una política de planificación dinámica. La tarea en ejecución es desalojada en el momento en que otra tarea con plazo más cercano se llegará a activar.

La planificación EDF no hace una suposición especifica sobre la periodicidad de las tareas, por lo que puede ser usado para planificación de tareas periódicas como aperiódicas.

La planificabilidad de un conjunto de tareas manejadas por EDF puede verificarse a través del factor de utilización del procesador. En este caso, el límite superior mínimo es uno. Un conjunto de tareas periódicas es planificable con EDF si y solo si:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{1.3}$$

1.8. Sistemas de Control

Los sistemas de control automático son sistemas dinámicos, donde el conocimiento de la teoría de control proporciona la base para entender el comportamiento de tales sistemas [15]. Estos sistemas emplean frecuentemente componentes de diferentes tipos, por ejemplo, componentes mecánicos, eléctricos, hidráulicos, neumáticos y combinaciones de estos; por lo tanto, al trabajar con controladores es necesario estar familiarizado con las leyes fundamentales que rigen a estos componentes.

Los sistemas de control son muy comunes en todos los sectores industriales, desde el control de calidad de productos industriales, líneas de ensamble automático, control de máquinas herramientas, tecnología espacial, armamento, sistema de transportación, robótica y muchos otros. Incluso problemas como el control de inventarios, los sistemas de control sociales y económicos, pueden resolverse con enfoques de la teoría del control.

Para comprender los sistemas de control automáticos, es necesario definir los siguientes términos de los sistemas de control[17]:

- **Planta**. Una planta es un equipo o quizás simplemente un juego de piezas de una máquina funcionando juntas, cuyo objetivo es realizar una operación determinada.
- **Sistema**. Un sistema es una combinación de componentes que actúan conjuntamente y cumplen un determinado objetivo. Un sistema no está limitado a los objetos físicos. El concepto de sistema puede ser aplicado a fenómenos abstractos y dinámicos, como son los sistemas de tiempo real.

- **Perturbaciones**. Una perturbación es una señal que tiende a afectar adversamente el valor de la salida de un sistema. Un ejemplo de perturbaciones en los sistema de tiempo real es la variación en los tiempo de ejecución de las tareas con respecto a los tiempo estimados o del peor caso.
- **Sistema de control retroalimentado**. Es aquel que tiende a mantener una relación preestablecida entre la salida y la entrada de referencia, comparando ambas y utilizando la diferencia como parámetro de control.
- **Sistema de control automático**. Es un sistema de control en el que la entrada de referencia o la salida deseada son constantes o varían lentamente en el tiempo, y donde la tarea fundamental consiste en mantener la salida en el valor deseado a pesar de las perturbaciones presentes.

Cualquiera que sea el tipo de sistema de control considerado, los componentes básicos del sistema pueden describirse en término de:

1. Objetivos del control.
2. Componentes del sistema de control.
3. Resultados.

En la figura 1.3 (a) se ilustra la relación entre estos tres componentes básicos en forma de diagrama de bloques. Estos tres componentes básicos pueden identificarse como entradas (referencias), componentes del sistema (controlador y planta) y resultados (salidas), respectivamente, como se muestra en la figura 1.3 (b).

En general, el objetivo del sistema de control consiste en controlar las salidas y de una manera predeterminada, la cual está dada por la referencia r y el controlador. A las entradas del sistema se le llama también consigna de operación y a las salidas variables controladas.

Como ejemplo simple del sistema de control descrito en la figura 1.3, consideremos el sistema direccional de un automóvil. La dirección de las dos ruedas frontales puede considerarse como la variable controlada "y"ó salida; la dirección del volante es la señal de control u o entrada a la planta. La planta o proceso en este caso está constituído por los mecanismos de la dirección y la dinámica de la totalidad del automóvil. Sin embargo, si el objetivo consiste en controlar la velocidad del vehículo, entonces, el grado de precisión ejercida sobre el pedal del acelerador es la señal de control u y la

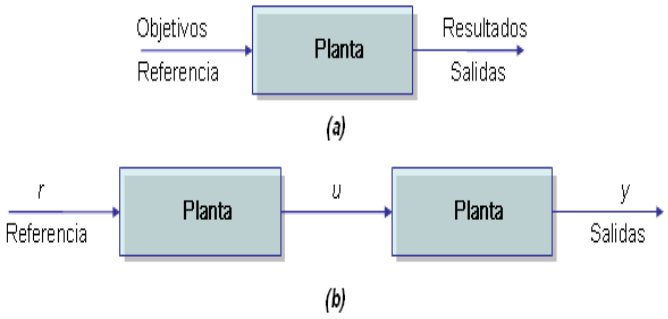

Figura 1.3: Diagrama de Control.

velocidad lograda es la velocidad controlada y. En su conjunto, podemos considerar al sistema de control del automóvil como constituído, por dos entradas (volante y acelerador) y dos salidas (dirección y velocidad). En este caso, los dos controles y salidas son independientes entre sí; pero en general, existen sistemas en los que los controles están acoplados [7]. A los sistemas con más de una entrada y una salida se le llama sistemas multivariables.

Otro ejemplo de un sistema de control, es el control del motor de un automóvil con un régimen de marcha en reposo. El objetivo de este tipo de sistema de control consiste en mantener la marcha en reposo del motor a un valor relativamente bajo (para economía del combustible) cualquiera que sea la carga aplicada al motor (por ejemplo, transmisión, dirección hidráulica, aire acondicionado, etc.). Sin contar con el control de marcha en reposo, cualquier aplicación repentina de una carga al motor causaría una caída de la velocidad del mismo y podría provocar que se detuviera. De esta manera, los objetivos principales del sistema de control con marcha reposo son (1) eliminar o reducir al mínimo la caída de velocidad del motor cuando se le aplica una carga y (2) mantener la marcha en reposo en el valor deseado.

La figura 1.4 muestra el diagrama de un sistema de control de marcha en reposo desde el punto de vista de entradas-sistema-salida. En este caso, el ángulo del obturador de la gasolina a es la entrada, el par de carga T_L representa una perturbación externa debido a la aplicación del aire acondicionado, dirección hidráulica, transmisión, etc. Las revoluciones del motor ω son la salida y el motor es el proceso o sistema controlado.

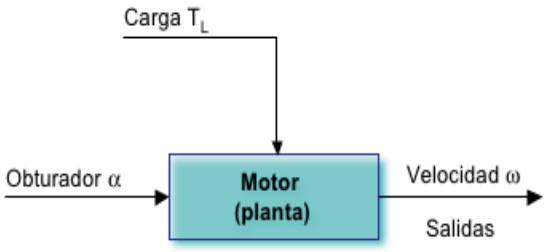

Figura 1.4: Diagrama de Control de marcha en proceso.

1.8.1. Sistemas de Control de Lazo Abierto

El sistema de control de marcha en reposo que se ilustra en la figura 1.4 es poco sofisticado y se clasifica como *sistema de control en lazo abierto*. Resulta fácil apreciar que dicho sistema, no cumpliría en forma satisfactoria con los requerimientos de desempeño deseado. Por ejemplo, si el ángulo del obturador α se fija un cierto valor inicial que corresponda a una determinada velocidad del motor, al aplicar una carga de par T_L, no hay manera de evitar una caída de la velocidad del motor. La única forma en que podrá operar el sistema será contando con los medios para ajustar a en respuesta a un cambio de T_L, para mantener ω en el nivel deseado. Debido a la simplicidad y economía de los sistemas de control de lazo abierto, estos se usan en la práctica en muchas situaciones. De hecho, casi todos los automóviles fabricados antes de 1981 no contaban con un sistema de control de marcha en reposo.

Otro ejemplo de sistemas en lazo abierto son las lavadoras eléctricas, pues en su diseño típico el ciclo de lavado queda determinado en su totalidad por la estimación y el criterio del operador humano. En una lavadora eléctrica verdaderamente automática contaría con los medios para comprobar el grado de limpieza de la ropa en forma continua y suspendería la operación por sí misma al alcanzar el grado de lavado deseado. Los elementos del sistema de control en lazo abierto casi siempre pueden dividirse en dos partes: el controlador y el proceso controlado, tal como lo ilustra el diagrama de bloques de la figura 1.5. Se aplica una señal de entrada o comando r al controlador, cuya salida actúa como señal de control u; la señal actuante controla el proceso controlado, de tal manera que la variable controlada y se comporte de acuerdo con estándares predeterminados.

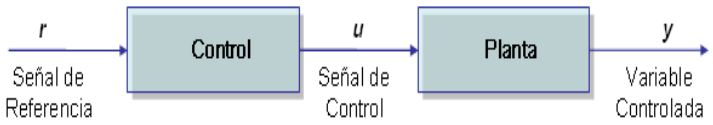

Figura 1.5: Elementos de un sistema de control en lazo abierto

1.8.2. Sistemas de Control de Lazo Cerrado

En los sistema de control en lazo abierto, el elemento faltante para lograr un control más preciso es un enlace o retroalimentación de la salida a la entrada del sistema. Para obtener un control más preciso, la señal controlada $y(t)$ debe retroalimentarse y compararse con la entrada de referencia, tras lo cual se envía a través del sistema una señal de control proporcional a la diferencia entre la entrada y la salida, con el objetivo de corregir el error. A los sistemas con uno o más lazos de retroalimentación de este tipo se le llama *sistema en lazo cerrado*. En la figura 1.6 se muestra el diagrama de bloques de un sistema de control en marcha en reposo en lazo cerrado. La entrada de referencia ω_r fija la velocidad deseada. Comúnmente, cuando el par de carga es cero, la velocidad del motor en reposo debe concordar con el valor de referencia ω_r, y cualquier diferencia entre la velocidad real y el valor deseado, causado por cualquier perturbación del par de carga T_L, es detectada por el transductor de velocidad y el detector de errores, con lo que el controlador operará sobre esta diferencia y proporcionará una señal para ajustar el ángulo del obturador α que corrija el error.

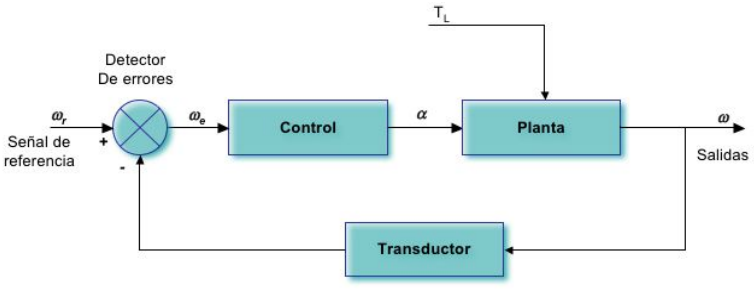

Figura 1.6: Sistema de control de marcha en reposo en lazo cerrado

1.8.3. Sistemas de Control Discretos

Los sistemas de control en tiempo discreto que se consideran en esta tesis, son en su mayoría lineales e invariables en el tiempo, aunque ocasionalmente se incluyen en las discusiones como sistemas no lineales y/o variantes en el tiempo. Un sistema lineal es aquel en el que se satisface el principio de superposición. De esta manera, si y_1 es la respuesta del sistema a la entrada x_1 y y_2 es la respuesta a la entrada x_2, entonces el sistema es lineal si y sólo si, para cualesquiera escalares α y β, la respuesta a la entrada $\alpha x_1+\beta x_2$ es $\alpha y_1+\beta y_2$.

Un sistema lineal se puede describir mediante ecuaciones diferenciales o en diferencias lineales. Un sistema lineal e invariable en el tiempo es aquel en el que los coeficientes en la ecuación diferencial o en diferencias no varían con el tiempo, esto es, es aquel sistema cuyas propiedades no cambian con el tiempo[17].

Sistemas de control en tiempo continuo y en tiempo discreto.

Los sistemas de control en tiempo discreto son aquellos sistemas en los cuales una o más de las variables pueden cambiar sólo en valores discretos de tiempo. Estos instantes, los que se denotarán mediante kT o $t_k(k = 0, 1, 2, ...)$, pueden especificar los tiempos en los que se lleva a cabo alguna medición de tipo físico o los tiempos en los que se extraen los datos de la memoria de una computadora digital. El intervalo de tiempo entre estos dos instantes discretos se supone que es lo suficientemente corto de modo que el dato para el tiempo entre éstos se pueda aproximar mediante una interpolación sencilla.

Los sistemas de control en tiempo discreto difieren de los sistemas de control en tiempo continuo en que las señales para los primeros están en la forma de datos muestreados o en la forma digital. Si en el sistema de control está involucrada una computadora digital como un controlador, los datos muestreados se deben convertir a datos digitales.

Los sistemas en tiempo continuo, cuyas señales son continuas en el tiempo, se pueden describir mediante ecuaciones diferenciales. Los sistemas en tiempo discreto, los cuales involucran se´ ales de datos muestreados o señales digitales y posiblemente señales en tiempo continuo, también se pueden describir mediante ecuaciones en diferencias después de la apropiada discretización de las señales en tiempo continuo.

Proceso de muestreo.

El muestreo de señales en tiempo continuo reemplaza la señal en tiempo continuo por una secuencia de valores en puntos discretos de tiempo. El proceso de muestreo se emplea siempre que un sistema de control involucra un controlador digital, puesto que son necesarias una operación de muestreo y una de cuantificación para ingresar datos a ese controlador. También se da un proceso de muestreo cuando las mediciones necesarias para control se obtienen en forma intermitente. Por ejemplo, en un sistema de seguimiento por radar, a medida que la antena del radar gira, la información acerca del azimut[2] y de la elevación se obtiene una vez por cada vuelta que da la antena. De este modo, la operación de rastreo del radar produce un dato muestreado. En otro ejemplo, el proceso de muestreo se necesita cuando un controlador o computadora de gran tamaño se comparte en tiempo entre varias plantas con el fin de reducir los costos. En este caso se envía periódicamente una señal de control para cada una de las plantas y de esta manera la señal se convierte en una de datos muestreados. El proceso de muestreo es seguido por un proceso de cuantificación. En el proceso de cuantificación, la amplitud analógica muestreada se reemplaza por una amplitud digital (representada mediante un número binario). Entonces la señal digital se procesa por medio de la computadora. La salida de la computadora es una señal muestreada que se alimenta a un circuito de retención. La salida del circuito de retención es una señal en tiempo continuo que se alimenta al actuador.

El término *discretización* en lugar de *muestreo* se utiliza con frecuencia en el análisis de sistemas con entradas y salidas múltiples, aunque ambos significan básicamente lo mismo. Es importante observar que de manera ocasional la operación de muestreo o discretización es enteramente ficticia y se ha introducido sólo para simplificar el análisis de los sistemas de control que en realidad sólo contienen señales en tiempo continuo. De hecho, a menudo se utiliza un modelo en tiempo discreto apropiado para un sistema en tiempo continuo. Un ejemplo es la simulación en una computadora digital de un sistema en tiempo continuo. Dicho sistema simulado en una computadora digital se puede analizar para obtener los parámetros que optimizan un índice de desempeño dado.

En la figura 1.7 se muestra un diagrama de bloques de un sistema de control discreto que presenta la configuración del esquema de control básico. En el sistema se incluye el control realimentado y el prealimentado. En el diseño de dicho sistema

[2] Ángulo que con el meridiano forma el círculo vertical que pasa por un punto de la esfera celeste o del globo terráqueo.

de control, se deberá observar que la "bondad"del sistema de control depende de circunstancias individuales. Se requiere elegir un índice de desempeño apropiado para un caso dado y diseñar un controlador de modo que optimice el índice de desempeño elegido.

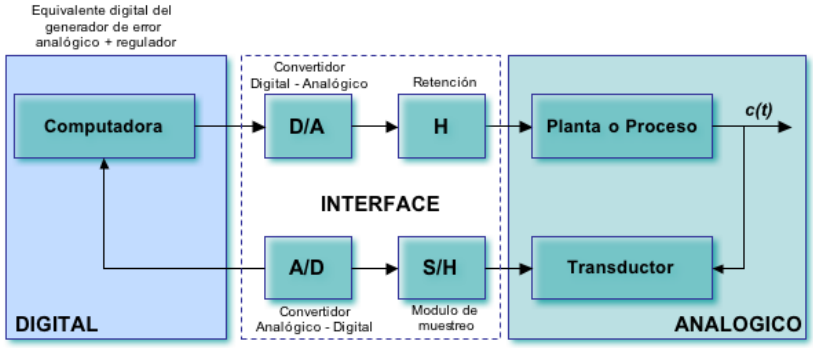

Figura 1.7: sistema de control discreto

1.9. Sistemas de Control con Simulink

Simulink es una herramienta para el modelado, análisis y simulación de una amplia variedad de sistemas físicos y matemáticos, inclusive aquellos con elementos no lineales y aquellos que hacen uso de tiempos continuos y discretos. Como una extensión de MatLab, Simulink adiciona muchas características específicas a los sistemas dinámicos, mientras conserva toda la funcionalidad de propósito general de MatLab. Simulink no es completamente un programa separado de MatLab, sino un anexo a él. El ambiente de MatLab está siempre disponible mientras se ejecuta una simulación en Simulink.

Simulink tiene dos fases de uso: la definición del modelo y el análisis del modelo. La definición del modelo significa construir el modelo a partir de elementos básicos construidos previamente, tal como, integradores, bloques de ganancia o servomotores. El análisis del modelo puede incluir la simulación, linealización y determinación del punto de equilibrio de un modelo previamente definido. Una vez construidos los diagramas de bloques, se puede ejecutar simulaciones y analizar los resultados, también de forma gráfica.

1.9.1. Simulación de los sistemas de Control en Simulink

Cada bloque mediante un modelo de Simulink tiene las siguientes características generales: Un conjunto de entradas u, un conjunto de salidas y y un conjunto de estados x (figura 1.8).

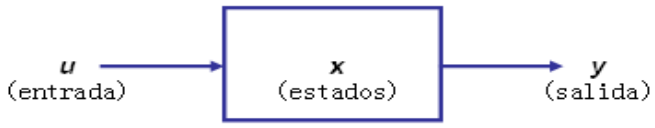

Figura 1.8: Modelo general de un bloque Simulink.

El vector de estado puede constar de estados continuos, estados discretos o una combinación de ambos. Así, la operación de Simulink consta de dos faces: inicialización y simulación.

- Durante la fase de inicialización, se tienen las siguientes 4 operaciones:

En primerlugar, los parámetros del bloque se pasan a MatLab para su evaluación. Los valores numéricos resultantes se utilizan como los parámetros actuales de bloque.

En segundo lugar, la jerarquía del modelo se reduce a su nivel inferior. Es decir, cada subsistema se sustituye por los bloques que contiene.

En tercer lugar, los bloques se disponen en el orden en que se necesita que se actualicen. El algoritmo de ordenación construye una lista tal que cualquier bloque con alimentación directa no se actualiza hasta que se calculan los bloques que excitan sus entradas. Es durante este paso cuando se detectan los lazos algebraicos.

Por último, se comprueban las conexiones entre bloques para asegurar que la longitud del vector de salida de cada bloque coincide con la entrada que esperan los bloques a los que se conecta.

- Durante la fase de simulación, se tienen las siguientes 4 etapas:

La primera etapa, comienza con la inicialización de los estados x_0. Estos estados son típicamente los valores corrientes de algún sistema, de ahí se modelan las salidas, que son funciones de los valores anteriores de sus variables temporales. Simullink maneja dos tipos de estados: estados discretos y estados continuos. Un estado continuo se cambia continuamente y un estado discreto es una aproximación de un estado continuo donde el estado es actualizado utilizando intervalos finitos (periódicos o aperiódicos).

En la segunda etapa, los estados discretos de x_d, son una función del estado actual y la entrada (1.4). Con esto, cada bloque que tenga un estado discreto se actualiza su estado en una función de simulink llamada `MdlUpdate`[3].

$$x_{d+1} = f_u(t, x_d, u) \tag{1.4}$$

En la tercera etapa, se utilizan las derivadas de x que se especifican en la función `MdlDerivatives` y estan dadas por la ecuación (1.5). Cada bloque que contiene estados continuos, proporciona sus derivadas a una función de solución que resuelve la operación del modelo Simulink (por ejemplo : ode5[4]). De acuerdo con Simulink, las funciones de solución son un conjunto de programas conocidos como "*solvers*". Estos programas producen una solución al sistema de ecuaciones diferenciales especificadas en (ec.1.5), obteniendo de esta manera el estado x.

$$\dot{x} = f_d(t, x_c, u) \tag{1.5}$$

En la última etapa, la salida y es función del estado continuo x_c, del estado discreto x_d, y la entrada u (1.6). En la figura 1.9, se muestra un diagrama a bloques del proceso que sigue Simulink para realizar una simulación.

$$y = f_0(t, x_c, x_d, u) \tag{1.6}$$

1.10. Relación entre Sistemas de Tiempo Real y Sistemas de Control

La mayoría de los sistemas de control con características de tiempo real están compuestos de dos partes: el proceso controlado o ambiente, y la computadora que

[3]Función de actualización, generada por el constructor de código de Simulink
[4]La función "ode5", es un algoritmo de solución para modelos Simulink.

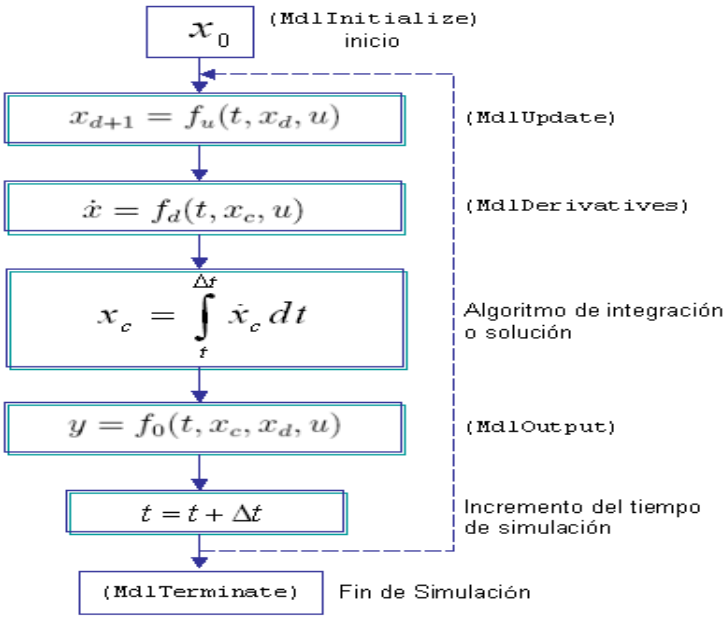

Figura 1.9: Diagrama a bloques del proceso de simulación.

efectúa el control del proceso. La computadora interactúa con el ambiente basándose en la información que percibe de varios sensores y el controlador aplica una ley de control que maneja el proceso mediante la información obtenida. Esta interacción permite a la computadora percibir correctamente el estado actual del ambiente. De otra forma, los efectos al sistema podrían ser muy severos.

La relación control-tiempo real, se establece directamente en el tiempo que toma el lazo de control en realizar su proceso, es decir, cada tiempo de muestreo en el lazo de control toma un plazo de tiempo para ejecutarse. Este plazo tiene períodos que son ejecutados por los elementos dentro del lazo de control.

Por lo que en los sistemas de control con características de tiempo real, se requiere de una cuidadosa invocación de tiempo para la ejecución del proceso del lazo de control (ya sea por medio de un interruptor o sensor o mediante un sistema de tiempo real común), para asegurar que el lazo de control se ejecute antes de que otro proceso u operación ocurra. Esto incluye el tiempo para leer y escribir datos desde un dispositivo externo. La figura 1.10, ilustra esta situación.

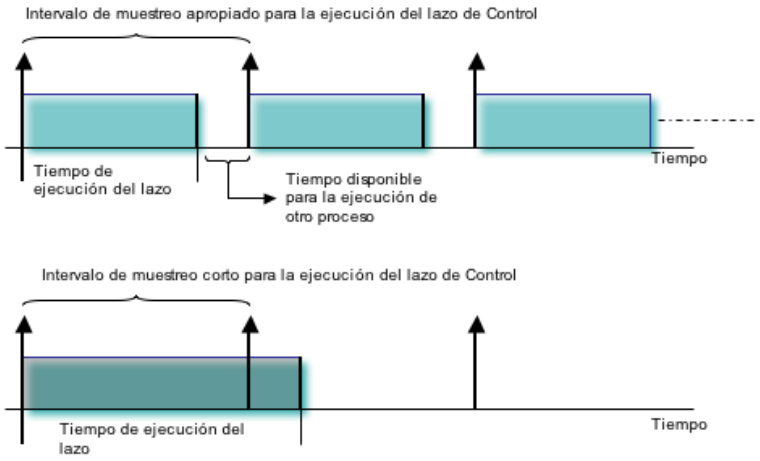

Figura 1.10: Tiempo de Respuesta de un sistema de control

El intervalo de muestreo debe considerarse y ser lo suficientemente largo para permitir que el lazo de control se ejecute entre las invocaciones de los otros procesos. En la figura 1.10, el tiempo entre las dos flechas verticales adyacentes es el intervalo de tiempo de ejecución del lazo de control. Los rectángulos en el diagrama superior muestran un ejemplo de un proceso de ejecución de un sistema de control, este puede completar su primera operación dentro del intervalo de tiempo, permitiendo ejecutar una tarea control (*background*) u otro proceso, debido a que está correctamente considerado el intervalo de muestreo dentro de la invocación del proceso en el sistema, con lo que se obtiene un sistema más estable. En el diagrama inferior el rectángulo sombreado indica que pasa si el intervalo de muestreo no es considerado o es muy corto. Si otra invocación de proceso ocurre antes de que el proceso termine, la ejecución de la operación se interrumpe, expulsándola o bien provocando un error en el sistema. Por lo que es necesario considerar como afecta la retroalimentación y el intervalo de tiempo al lazo de control. En la siguiente sección se describirá esta situación.

1.10.1. Retroalimentación y sus efectos

En la figura 1.7, el uso de la retroalimentación tiene el propósito de reducir el error entre la entrada de referencia y la salida del sistema. Sin embargo, la importancia de los efectos de la retroalimentación en los sistemas de control es mucho más importante. La reducción del error del sistema es sólamente uno de los diferentes efectos

importantes que la retroalimentación tiene en un sistema, por lo también tiene efectos en las características de desempeño del sistema tales como estabilidad, ganancia total y sensibilidad.

En general, se puede afirmar que, cuando las variables de un sistema de control exhiben una secuencia cerrada de *relaciones de causa y efecto*, el sistema cuenta con una retroalimentación (figura 1.11).

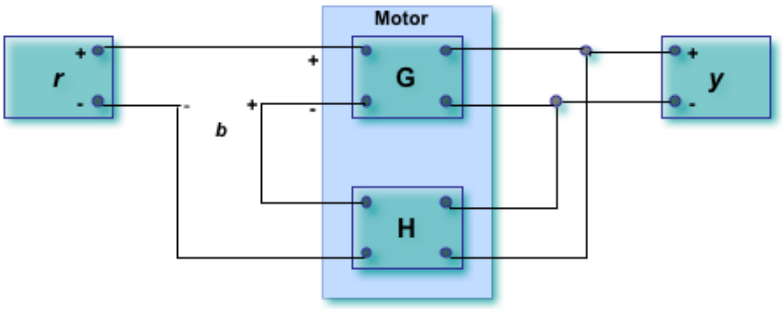

Figura 1.11: Sistema de retroalimentación

Tomando en cuenta la configuración del sistema de retroalimentación simple que se muestra en la figura 1.11, donde r es la señal de entrada, y es la señal de salida, e es el error y b es la señal de retroalimentación. Los parámetros G y H pueden considerarse como ganancias constantes. Mediante operaciones algebraicas simples, se puede demostrar que la relación entrada-salida de un sistema es:

$$M = \frac{y}{r} = \frac{G}{1 + GH} \tag{1.7}$$

Esta relación de la estructura del sistema retroalimentado permite conocer más a fondo los efectos de la retroalimentación.

Efecto de la retroalimentación sobre la ganancia total.

Como puede apreciarse en la ecuación 1.7, la retroalimentación afecta a la ganancia G de un sistema sin retroalimentación por un factor de, $1 + GH$. La referencia de la retroalimentación en el sistema de la figura 1.11 es negativa, pues a la señal de retroalimentación se le asigna un signo menos. La cantidad GH puede incluir en sí misma un signo negativo, por lo que el efecto general de la retroalimentación es que

puede incrementar o reducir la ganancia. En un sistema de control práctico, G y H son funciones de frecuencia, por lo que la magnitud de $1 + G\,H$ puede ser mayor que 1 en un intervalo de frecuencia pero inferior a 1 en otros. Por consiguiente, la retroalimentación puede aumentar la ganancia del sistema en un intervalo de frecuencia y disminuído en otro.

Efecto de la retroalimentación sobre la estabilidad.

La estabilidad es un concepto que describe si un sistema será capáz de seguir una entrada. Se dice que un sistema es inestable cuando su salida está fuera de control o aumenta sin límites. Para comprender los efectos de la retroalimentación sobre la estabilidad, podemos referirnos nuevamente a la expresión de la ecuación 1.7. Cuando $G\,H = -1$, la salida del sistema es infinita para cualquier entrada finita. Se puede decir que la retroalimentación puede causar inestabilidad en un sistema originalmente estable.

Efecto de la retroalimentación sobre la sensibilidad.

Las consideraciones de sensibilidad suelen tener un papel importante en el diseño de sistemas de control. Puesto que todos los elementos tienen propiedades que varían con el medio ambiente y su tiempo de uso, no siempre es posible considerar que los parámetros de un sistema pueden ser totalmente estacionarios en el intervalo total de la vida operacional del sistema. En general, un buen sistema de control debe ser insensible a estas variaciones de los parámetros y seguir siendo capaz de producir una respuesta adecuada.

Efecto de la retroalimentación sobre el ruido.

Todos los sistemas de control están sometidos a señales extrañas o ruidos durante su operación. Ejemplos de estas señales son las variaciones en los tiempo de ejecución de las tareas en un sistema de tiempo real, el voltaje de ruido térmico en los amplificadores electrónicos y el ruido de escobillas o conmutadores en los motores eléctricos. El efecto de la retroalimentación sobre el ruido depende en gran parte del punto de introducción del ruido al sistema, en muchas situaciones, la retroalimentación puede reducir el efecto del ruido sobre el desempeño del sistema.

1.11. Trabajo Propuesto

La mayoría de los sistemas de control son implementados en un microprocesador usando un kernel de tiempo real o un sistema operativo de tiempo real. En la mayor parte de los casos el control es ejecutado periódicamente, enfocándose solamente en el dominio del problema sin preocuparse de cómo la ejecución concurrente de los procesos de cómputo afecta el comportamiento y la estabilidad del sistema de control. Actualmente, existen aplicaciones que auxilian al diseño de sistemas de control como MatLab/Simulink, LabVIEW, Wincon, etc. Estas aplicaciones presentan una serie de herramientas (funciones, bloques y modelos) desarrolladas para la simulación de los procesos de control.

Por otro lado, se ha incrementado el interés por desarrollar aplicaciones o herramientas que ayuden al diseño y simulación de sistemas de control con características de tiempo real. Sin embargo, son pocas las herramientas que ayudan al diseño de sistemas de control que contemplen restricciones de tiempo, y que además garanticen la ejecución concurrente de los algoritmos de control , o bien, que interactúen con otros ambientes de software externos, que permitan aplicar políticas de planificación y utilizar una interfaz gráfica alternativa para la visualización de los procesos.

En este trabajo, se presenta el desarrollo de una interfaz de programación en MatLab para la ejecución de tareas de control sobre un kernel de tiempo real. El sistema propone tres etapas: una etapa de generación de código (*módulo generador de tareas*), otra etapa de manejo de código (*interfaz de programación*) y una última etapa de ejecución (*micro-kernel de tiempo real*).

1.11.1. Estructura de la aplicación

El esquema general de la aplicación se muestra en la figura 1.12. La figura muestra en una primera etapa un diagrama de control realizado en Simulink, al cual se le adiciona el módulo generador de tareas; esté construye código en C utilizando la herramineta RTW (*Real Time Workshop toolbox*) de MatLab. El RTW permite generar código o un programa ejecutable (C, .exe, .dll) desde Simulink [16]. Posteriormente este código se exportará al micro-kernel, donde se ejecutará por medio de la interfaz de aplicación como una tarea de tiempo real.

El generador de tareas es uno de los elementos más importantes de la aplicación, ya que por medio de este módulo se configura la estructura del RTW para generar

Figura 1.12: Esquema general de la interfaz de programación

código. Este módulo permite generar un código más simplificado a partir de un diagrama de control de Simulink (elaborado por el usuario), para que el código construido sea utilizado en la interfaz de programación y ejecutado como tarea de tiempo real dentro del micro-kernel de aplicación.

Para este caso, Simulink funciona como la interfaz gráfica del usuario para el ajuste de parámetros del modelo, mientras que el micro-kernel de tiempo real, funcionará como la etapa de ejecución del código ya previamente tratado. El modelo de tareas de control se ejecuta dentro del micro-kernel sin recursos compartidos, teniendo la posibilidad de correr varios procesos de forma concurrente. El método de planificación utilizado en la aplicación es uno de prioridad fija, en particular se emplea la política de RR (*Round Robin*).

Esta aplicación podrá ser utilizada para el diseño e implementación de sistemas de control con características de tiempo real, integrando así la funcionalidad de la herramienta MATLAB/Simulink/Embedded Real Time Worshop con otra aplicación fuera del ambiente de MATLAB (micro-kernel de tiempo real).

1.12. Organización del Documento

El presente documento consta de 5 capítulos , que describen toda la realización de la tesis. Como ya se expuso en los apartados anteriores en el presente capítulo de describió el marco general, estado del arte y los objetivos de la tesis, los cuáles proporcionan al lector un panorama general del trabajo realizado en la tesis. Así mismo, se introduce los conceptos más importantes sobre los sistemas de tiempo real, los sistemas de control y cuáles son los parámetros que los caraterizan, así como de la

relación entre ambos. También se definen los diferentes tipos en que se clasifican las políticas de planificación.

El capítulo 2, se describe el funcionamiento de la herramienta RTW como generador de código y el desarrollo del bloque generador, así como de las condiciones de configuración de la herramienta RTW para generar código. En este capítulo se describen también las ventajas y desventajas del módulo generador, así como de las limitantes de la herramienta y se enumeran las condiciones asumidas en el sistema.

En el capítulo 3, se describe el desarrollo de la API para el manejo de código producido por el módulo generador de tareas. Se describe la arquitectura del sistema y el modelo del sistema utilizado. También se presentan las características del microkernel utilizado y por último se explican como funciona el mecanismo de ejecución de las tareas de control.

En el capítulo 4 se describen las pruebas de simulación llevados a cabo para medir el desempeño de la aplicación propuesta para distintas políticas de planificación, las condiciones de la simulación y los resultados obtenidos para un caso de estudio particular.

Por último en el capítulo 5, se presentan las conclusiones del proyecto de tesis y la propuesta de trabajo futuro. En los apéndices A y B, se muestran cuadros y registros complementarios a los capítulos mencionados anteriormente y finalmente, el apéndice C, se presenta el código generado para un modelo Simulink utilizado en uno de los ejemplos mostrados en la tesis.

Capítulo 2

Proceso de generación de código

2.1. Introducción

Después de presentar los aspectos generales y la relación entre los sistemas de control y de tiempo real, en este capítulo se presentan las características fundamentales de la herramienta RTW de MatLab/Simulink. Se describe también el proceso de generación de código y el proceso que se sigue para generar el código como tarea de control, el cual es la parte fundamental de esta tesis. Así mismo, se describe detalladamente el desarrollo del bloque generador de código usado en la aplicación de esta tesis. Este bloque funciona como una librería de Simulink diseñada para la generación de código C, el cual utiliza un archivo de instrucciones *fixed-step* de MatLab escrita en lenguaje C que ejecuta y configura todos los parámetros de la herramienta. Finalmente se presenta como se realiza la selección del *Target*, donde el bloque generador define la configuración de este.

2.2. Real Time Workshop

El *Real Time Workshop* o RTW genera código ANSI C optimizado, portátil y personalizable de modelos de Simulink. Construye automáticamente programas que se ejecutan en tiempo real o como simulaciones independientes de tiempo no real. El código generado puede funcionar, como se verá más adelante, en hardware de PC, DSPs, microcontroladores en entornos de núcleo desnudo (sin sistema operativo) y con sistemas operativos en tiempo real (RTOS) comerciales o registrados, o bien,

como el de la aplicación realizada en esta tesis, que utiliza un micro-kernel de tiempo real para hacer las pruebas correspondientes de los modelos generados en simulink [16].

El código resultante acelera también las simulaciones, proporciona protección de la propiedad intelectual y funciona en diversos targets de prototipaje rápido en tiempo real. También es posible generar automáticamente código Ada 83 y Ada 95 utilizando para ello el *Real-Time Workshop Ada Coder*.

2.2.1. Generalidades

El RTW permite generar código (portable y generalizado) o un programa ejecutable (*.exe o .dll*) desde un diagrama de Simulink y otros de sus usos importantes son: aceleración de las simulaciones, protección de la propiedad intelectual (S-function Target, Rapid Simulation Target), simulación en tiempo real, simulación HIV (hardware-in-the-loop) y cuenta con procedimientos para la generación de código, creación del ejecutable y transferencia de éste a un archivo objetivo o *target* (pro ejemplo, xPC y Tornado), donde Simulink funciona como la interfaz de usuario para monitoreo y ajuste de parámetros.

Los requerimientos para que el RTW funcione son: MatLab, Simulink (a partir de la versión 5) y un compilador soportado (Visual C++, Borlad, LCC, Watcom). Para la aplicación del proyecto de tesis, se utilizó Borland C, ya que este, tiene la capacidad para desarrollar aplicaciones tanto de 16 como de 32 bits, además, corre sobre Windows 95 y NT, con la posibilidad de generar código para un variado tipo de ambientes y sistemas operativos, como son DOS, Windows 3.1, 95 y NT. Esto es de gran utilidad, debido a que la plataforma del micro-kernel, donde se prueban los códigos generados es MS-DOS y la compilación se realiza bajo el compilador `BORLAND C 3.1`. Con esto se asegura de que se genere un código más manejables y estándar por si se pretende emplear el código en otras plataformas.

Una de las características más importantes del RTW, es que genera código para una plataforma diferente (*target*, sistema objetivo) a la que corre el MatLab (*host*, sistema anfitrión); aunque existen excepciones como RTWT y el GRT (self-targeting). También el RTW, puede trabajar en modo externo, donde es posible: monitorizar las variables y ajustar los parámetros, mientras el programa se ejecuta en el target (se tiene una comunicación tipo cliente-servidor vía TCP/IP, RS232 o memoria compartida).

Relación con Simulink.

La relación de simulink con RTW, principalmente se da con:

- Las plataformas con las que trabaja en modo externo: TRWT (Windows como host y target), Tornado (VxWoks), xPC, Real-Time Target (Linux como host y target).
- El Simulink junto con el RTW es un lenguaje de alto nivel (VHLL): los bloques de Simulink son elementos básicos del lenguaje (gráfico).
- El generador de código se incluye en el RTW y es un compilador gráfico de última generación basado en diagramas de bloques.
- El generador, construye código en C o Ada de manera rápida y correcta, legible y personalizado.
- Simulink comparte con el RTW un librería de código personalizable *Custom Code Library*, que permite insertar código personalizado dentro del código fuente generado.

2.3. Arquitectura RTW

El RTW es un sistema abierto diseñado para usarse con una amplia variedad de ambientes operativos y tipos de hardware. Existen varias maneras de modificar y extender los elementos principales del RTW. En la figura 2.1, se muestra la arquitectura general del RTW. Como se puede observar, uno de los pasos que se indican en este diagrama, es el de generación de código C a partir del modelo de Simulink [2] para poder ejecutar dicho modelo en un target. El Real Time Workshop soporta numerosos entornos de archivos objetivo o se puede diseñar uno como se vera más adelante. El mecanismo más importante del RTW es el compilador de lenguaje objetivo o TLC, ya que este se utiliza para obtener el código a partir del diagrama de simulación. Este TLC utiliza dos tipos de archivos (extención, `.tlc`), los cuales pueden personalizarse, estos son:

- *Sistema de archivos objetivo***.** Describe como se genera el código para un determinado target.
- *Bloque de archivos objetivo***.** Define el código para un modelo determinado.

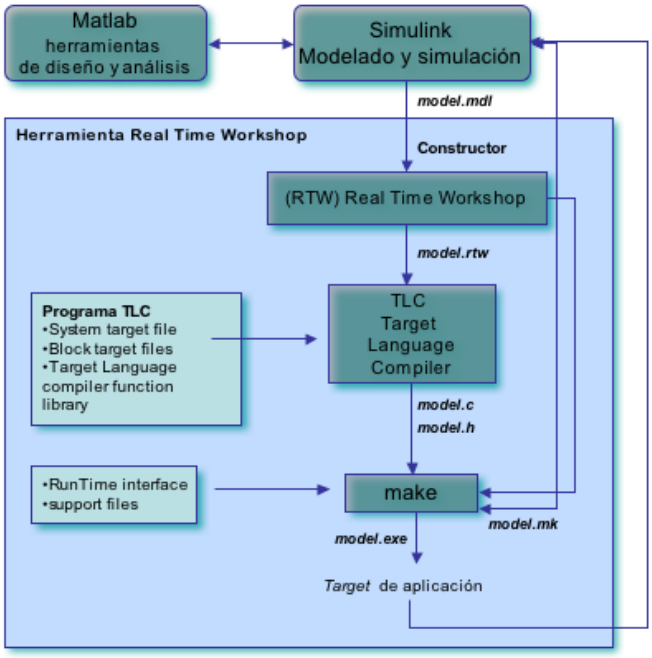

Figura 2.1: Arquitectura general.

Una rutina `make_rtw` genera un archivo `model. rtw` y de ahí crea el `model.mk` a partir de la plantilla `*.tmf`. El RTW utiliza estas plantillas `*.tmf` para la construcción de ejecutables, dependiendo del archivo objetivo o *target*. Así finalmente de el archivo `*.mk` se genera el ejecutable [16].

2.3.1. Generación de código en el RTW.

Un archivo de alto nivel o M-file controla el proceso de generación de código, el archivo que se suele usar para la mayoría de los *targets* es el `make.rtw` [16]. La generación de código con *Real Time Workshop* (RTW) comienza con cuatro pasos que se realizan automáticamente cuando se activa el proceso; los pasos son los siguientes:

1. El RTW analiza el modelo de bloques creado en Simulink, para ello evalúa:

   - Los parámetros de los bloques y de simulación.
   - Los períodos de muestreo y las dimensiones de las señales de propagación.

- El orden de ejecución de cada bloque del modelo de Simulink.

  Durante esta fase el RTW lee el archivo del modelo (`model.mdl`) y lo convierte en una representación intermedia del modelo que es almacenada un archivo ASCII llamado `model.rtw`.

2. El *Compilador de Lenguaje Objetivo* o TLC elegido transforma la descripción intermedia del modelo almacenado (*model.rtw*) en el código específico del archivo objetivo o *target*. El compilador del archivo objetivo ejecuta un programa TLC que comprende diversos archivos de instrucciones o (archivos *script*). Estos archivos de instrucciones especifican cómo generar el código a partir del modelo usando el archivo `model.rtw` como entrada [17].

3. El *Target Language Compiler (TLC)* construye un `makefile` específico llamado `model.mk`. Este `makefile` se encarga de dar las instrucciones adecuadas a la utilidad `make` para realizar la compilación y el enlazado del código fuente generado a partir del modelo. El RTW crea este archivo `makefile` a partir de otro llamado `system.tmf` (donde *system* representa el nombre del archivo objetivo o *target* seleccionado) a través del cual se especifican los compiladores, las opciones de compilado y otra información adicional que se utiliza en la creación del ejecutable [17].

4. La creación de un programa ejecutable es el último paso de este proceso, este paso es opcional. La creación de un ejecutable tiene lugar después de que el archivo *model.mk* ha sido creado. La utilidad *make* del sistema lee el `makefile` para compilar el código fuente, enlaza los archivos objeto, las librerías y genera un ejecutable (`model.exe`). Si se desea, la utilidad *make* también puede descargar el ejecutable en el archivo objetivo para el que fué creado. El proceso queda representado gráficamente en la Figura 2.2.

Así, el RTW ofrece una flexibilidad muy alta a la hora de configurar el proceso de generación de un programa ejecutable a partir de un diagrama de Simulink. Este permite crear archivos TLC propios para personalizar la generación de código en general o para un determinado bloque de Simulink, que es lo que se realiza en este trabajo de tesis. También permite personalizar las opciones de compilado y enlazado (*link*) de la plantilla *makefile* [17].

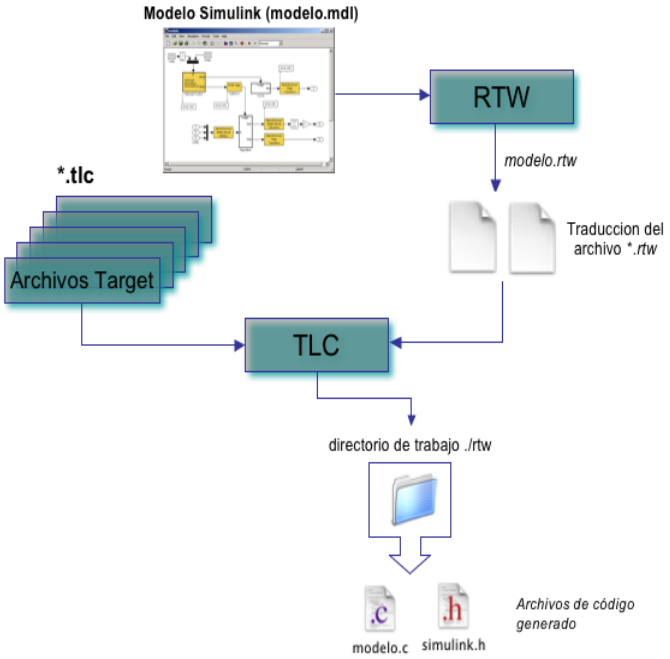

Figura 2.2: Proceso de generación de código en RTW.

2.3.2. Archivos objetivo del Real Time Workshop (*targets*)

El RTW, usa plantillas *targets* o archivos objetivos para traducir modelos de Simulink a código de ANSI/ISO C. Estas plantillas especifican el ambiente en el cual código generado se ejecutará. Los archivos *target* tienen la característica que se pueden personalizar o utilizar las configuraciones ejecutables. Los archivos objetivo que incluyen el RTW son:

- **GRTT**—El archivo objetivo o *target* de tiempo real genérico genera código para ajuste interactivo de parámetros de los modelos, despliege y registro de los resultados en tiempo real de la simulación y asigna datos estáticamente (para la ejecución en tiempo real eficiente).

- **GRTTM**—El archivo objetivo o *target* de tiempo real genérico de asignación, utiliza la asignación de memoria dinámica en el código generado, permitiéndole incluir múltiples instancias de un modelo o múltiples modelos.

- **SFT**—El archivo objetivo o *target* para Funciones S, convierte los modelos en DLL's de Simulink, permitiéndole compartir modelos para simulaciones sin el compromiso

de la propiedad intelectual.

- **RSim**—El archivo objetivo o *target* de simulación rápida o (RSim), proporciona una plataforma rápida y flexible para realizar pruebas por lote o la simulación de Monte Carlo, utilizando los *solvers*[1] de paso fijo o variable, haciendo fácil modificar los parámetros de los modelos y los datos de las señales de entrada para cada ejecución y guardado de datos hacia una única salida.

- **TT**—El archivo objetivo o *target* de Tornado, genera código para la ejecución en **VxWorks**.

Todos los archivos objetivo incluidos en el RTW, soportan modelos de ejecución en *uni-tarea*, *multi-tarea* e híbridos de tiempo continuo. Como se mencionó anteriormente estos archivos objetivo, se pueden extender para crear interfaces en tiempo de ejecución y manejadores o archivos *drivers* para cargar a una plataforma en particular o a un ambiente de trabajo especial o *host*. Una vez visto la clasificación general de los *targets*, en la siguiente sección se mostrará cual es la arquitectura general de la herramienta RTW.

2.4. Estructura TLC

El *Target Language Compiler* es una herramienta que permite ajustar la generación automática de código realizada por el Real Time Workshop a las necesidades del usuario. De este modo, se puede producir código dependiente de la plataforma o mejorar las características de rendimiento y de tamaño de código del programa creado. Para la generación de código C, el *Target Language Compiler* utiliza un conjunto de "`Archivos de bloque objetivo`"que especifican el tipo particular de código que se debe añadir por cada bloque o modelo. Habitualmente este tipo de archivos se usa para mejorar las características del código que será generado[17]. Existen dos razones para usar estos archivos:

- Reducir el tamaño del programa creado. En principio, el *Real Time Workshop* llama a todas las posibles funciones que puedan aparecer en un bloque de este tipo, aunque estén vacías. Con estos archivos se evita esta situación, de forma que sólo aparezcan llamadas a las funciones que vayan a desempeñar alguna

[1] Los *solvers* o ejecutores son funciones que ejecutan algoritmos que se utilizan para resolver sistemas de control.

labor. Hay que tener en cuenta que en sistemas empotrados es de vital importancia que el ejecutable final sea reducido, ya que en ellos los recursos son normalmente limitados.

- Generar código según algún parámetro de una función S. Si se desea que un determinado parámetro dentro de un modelo simulink genere una porción de código, no queda otra alternativa que codificar un archivo tlc para el bloque correspondiente. En caso contrario el programa creado contemplará todas las posibles opciones para ese parámetro, con lo cual el ejecutable tendrá un tamaño innecesariamente grande.

A continuación se enumeran las principales funciones que deben aparecer en los archivos tlc creados:

1. **BlockInstanceSetup**: es ejecutada previamente a la generación de código para todos los bloques de un determinado tipo que tienen definida esta función en su archivo tlc. Por ejemplo, si hay 10 bloques de un tipo, su función es llamada 10 veces, una por cada bloque del modelo. Habitualmente se usa para añadir código específico de un bloque determinado en el archivo de encabezado (`model.h`) o en el comienzo del archivo fuente (`model.c`). En el archivo de encabezado se escribirán etiquetas, variables globales y prototipos de funciones, mientras que en el archivo fuente se añadirán las definiciones de dichas funciones.

2. **BlockTypeSetup**: es ejecutada una sola vez antes de que comience la generación de código. En el ejemplo anterior, esta función sólo sería ejecutada una vez para todos los bloques de un tipo del modelo. Normalmente es utilizado para añadir código específico de un tipo especial de bloque.

3. **Start**: el código que se incluya en esta función irá a parar a la función mdlStart del archivo fuente.

4. **Outputs**: el código que se incluya en esta función irá a parar a la función mdlOutputs del archivo fuente.

5. **Update**: el código que se incluya en esta función irá a parar a la función mdlUpdate del archivo fuente.

6. **Terminate**: el código que se incluya en esta función irá a parar a la función mdlTerminate del archivo fuente.

2.4.1. Archivos de enlace compatibles (*hook files*)

El RTW contiene archivos de enlace o *hook files*, que son ejecutados en puntos específicos durante la generación de código. Por lo que se pueden usar para añadir funciones como agregar una variable especifica, seleccionar un sistema objetivo específico o *target*, o bien cambiar del proceso de construcción de código. Estos archivos se empezaron a utilizar desde la versión 6 de MatLab, aunque ya se contemplaban en la versión 5 (R13). Esta sección describe la importancia de los archivos M de enlace o *hook files* especialmente referidos al archivo `STF_make_rtw_hook.m`, ya que se utiliza en el proceso de generación de código que usa el módulo implementado en el trabajo de tesis. Este archivo de enlace implementa una función `STF_make_rtw_hook` que envía a una acción específica, dependiendo de un argumento `hookMethod` pasado a la cadena.

El funcionamiento de estos archivos se presenta, cuando el proceso de construcción comienza. Este, en el momento que inicia, llama automáticamente al archivo `STF_make_rtw_hook` pasándole el argumento `hookMethod`, para definir el archivo objetivo y establecer la estructura del código.

Así, cuando un modelo es creado y no se específica ninguna característica de configuración en particular, el RTW usa el módulo de generación y ejecución de código por defecto que contiene archivos de enlace que no realizan ninguna función. Por lo que cuando se genera un modelo, el RTW busca si hay algún archivo de estos en el directorio, o en el path con el nombre del archivo objetivo (por ejemplo ⟨*mytarget*⟩`_rtw_info_hook.m`, donde *mytarget* identifica el nombre del archivo objetivo utilizado). Si el archivo es encontrado, este puede anular la configuración por defecto. Entonces el usuario puede subsecuentemente volver a especificar, cualquier característica del proceso de construcción.

Para asegurar, que el archivo de enlace `STF_make_rtw_hook` sea llamado correctamente por el proceso construcción, se debe asegurar que la siguiente condición sea encontrada:

- El nombre del archivo debe tener el nombre del archivo objetivo (STF), añadido a la cadena `_make_rtw_hook.m`. Por ejemplo, para el generador propuesto en la tesis, se genera código vía un archivo STF de nombre `generador.tlc`, por lo que, el archivo de enlace `STF_make_rtw_hook.m` se llama `generador_make_rtw_hook.m`. De la misma forma, la función de enlace implementada dentro del archivo se establece del mismo modo.

La función de enlace en los archivos sigue una función prototipo descrita en la siguiente sección.

Función de prototipo y argumentos de `STF_make_rtw_hook.m`

La función prototipo para `STF_make_rtw_hook.m` es:

```
function STF_make_rtw_hook(hookMethod, modelName, rtwRoot,
         templateMakefile, buildOpts, buildArgs)
```

El argumento es definido como:

- `hookMethod`: Cadena que específica el estado del proceso de construcción desde el cual la función `STF_make_rtw_hook` es llamada. El diagrama de flujo de la figura 2.3, resume el proceso de construcción, destacando los puntos de enlace. Los valores válidos para `hookMethod` son `'entry'`, `'before_tlc'`, `'before_make'`, y `'exit'`.
- `rtwRoot`: Cadena reservada para RTW.
- `modelName`: Cadena que específica el nombre del modelo. Validad para todas las fases del proceso de construcción.
- `templateMakefile`: Nombre de la plantilla `makefile`.
- `buildOpts`: Estructura de MatLab que contiene los campos de la siguiente lista. Válido solamente para las fases `'before_make'` y `'exit'`. Los campos `buildOpts`son:
  - `modules`: Arreglo de carácteres que específica la lista de los archivos C generados, tal como: `model.c`,`model_data.c`, etc.
  - `codeFormat`: Arreglo de caracteres que contiene el formato de código que específica el archivo objetivo.
  - `noninlinedSFcns`: Arreglo de celdas que especifican la lista de las funciones S en el modelo.
  - `compilerEnvVal`: Cadena que específica el ambiente del valor de la variable del compilador (por ejemplo, (`C:/Applications/BorlandC`).
- `buildArgs`: Arreglo de caracteres que contienen el argumento `make_rtw`. Cuando se invoca el proceso de construcción, el argumento `buildArgs` es copiado desde la cadena del argumento seguido de `make_rtw` en el campo del comando `Make` del archivo objetivo de configuración de opciones del RTW.

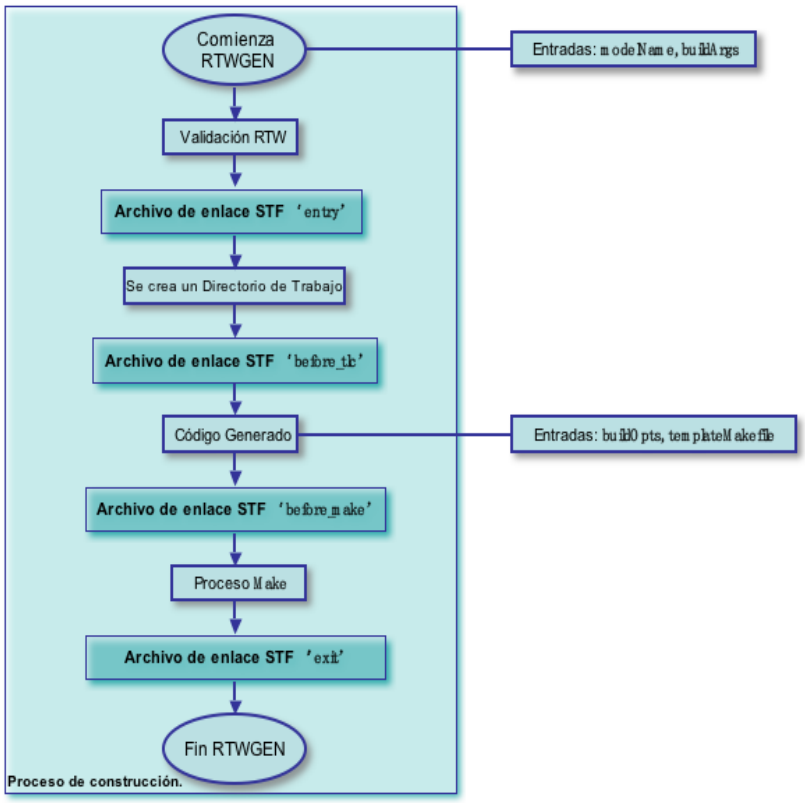

Figura 2.3: Diagrama de flujo del Proceso de construcción (Puntos de enlace importantes).

2.4.2. Funciones del RTW

En todo código genenerado, independientemente del modelo, el RTW establece funciones que se asocian a una estructura de ejecución del modelo. A continuación, se muestran las principales funciones que debe incluir un archivo generado.

1. **MdlInitializeSizes**: Simulink llama a esta rutina al editar el modelo para determinar el número de puertos de entrada y salida, así como su tamaño. También le llama al comienzo de la simulación para otro tipo de información, como puede ser el número de parámetros que recoge la S-function o el número de estados del algoritmo que se implementa.

2. **MdlInitializeSampleTimes**: Esta función es llamada para establecer el periodo de la S-Function.

3. **MdlStart**: Esta función lleva a cabo tareas que se deben realizar en el comienzo de la ejecución del programa.

4. **MdlOutputs**: Esta función calcula las salidas. Es llamada durante el lazo de ejecución cada vez que hay que actualizar la salida.

5. **MdlUpdate**: Normalmente esta función se usa para actualizar estados discretos, pero puede utilizarse para realizar cualquier tarea que deba tener lugar una vez por cada paso del ciclo de ejecución.

6. **MdlDerivatives**: Esta función se usa en sistemas contínuos para calcular el siguiente estado.

7. **MdlTerminate**: Esta función lleva a cabo tareas al fín de la ejecución del programa.

8. **MdlRTW**: En el caso de que se vaya a generar un fichero tlc para la S-function, esta rutina es la encargada de pasar los parámetros a dicho programa para que se pueda llevar a cabo la generación de código según establezcan tales valores.

Una vez generado el código con las funciones que describen su comportamiento, este podrá ser utilizado sin necesidad de una larga codificación y depuración manual. Con las características fundamentales de la herramienta RTW descritas, en las siguientes secciones se describirá el desarrollo del bloque generador de código usado en la aplicación y el proceso que se sigue para generar el código como tarea de control, el cual es la parte fundamental de esta tesis.

2.5. Bloque Generador de tareas (Bkernel)

El bloque generador de código, es un módulo de Simulink que se construyó con la finalidad de agilizar la configuración del generador RTW. Esto permite tener en un solo bloque las configuraciones necesarias para producir el código de cada modelo simulink, teniendo como base un bloque de Simulink el módulo generador de tareas. El módulo nombrado “Bkernel”, se implementó como una librería precompilada de Simulink, la cual se agrega al modelo construído. Mediante la activación de este, configura toda la herrameinta RTW y produce el código de acuerdo al archivo objetivo que se utiliza; en las siguientes secciones, se describe el proceso de selección del archivo objetivo y la configuración de los archivos TLC.

De esta manera el módulo generador “Bkernelrealiza las siguentes funciones:

- Configura el generador de código para el archivo objetivo empleado.
- Adapta el proceso de generación al modelo Simulink dentro del ambiente de desarrollo RTW.

El bloque generador "Bkernel" (figura 2.4) funciona como una libreria de Simulink diseñada para la generación de código C. Este módulo tiene las siguientes características:

1. El bloque es un archivo de instrucciones *fixed-step* de MatLab escrita en lenguaje C.
2. Ejecuta y configura todos los parámetros de los modelos .
3. Este bloque puede ejecutar un archivo de instrucciones de configuración independientemente del proceso de generación de código y los cambios son visibles en la GUI y pueden ser salvados con el modelo.

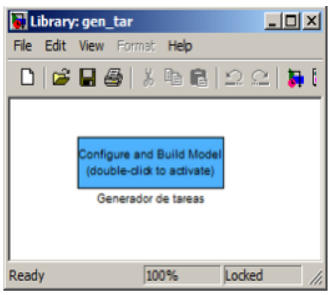

Figura 2.4: Bloque Generador.

2.5.1. Archivo de instrucciones del módulo generador

El archivo de instrucciones o *script* del módulo generador, describe una sola función que no regresar ningun valor y toma un argumento simple `cs` para invocar la funcion **m** que ejecuta la configuración del RTW. Esta función es de la forma:

función: `function gen_tar(cs);`

Esta función **m**, contiene instrucciones mismas de RTW que configuran la herramienta para evitar la configuración manual, como por ejemplo la instrucción de configuración `'SupportAbsoluteTime'`, la cuál inhabilita o deshabilita la función de tiempo absoluto, que se usa normalmente en aplicaciones embedidas.

```
setparam(cs,'SupportAbsoluteTime','off');
```

El argumento `cs` es un elemento que se puede manejar como un objeto que contiene información sobre los modelos activos en Simulink. Esto lo toma Simulink y lo pasa dentro de su función de configuración cuando el usuario acciona el bloque generador. Así, al módulo generador le es asociado el: archivo de instrucciones `function gen_tar(cs)` que en realidad es un archivo **M** (M-file[2]), que se adaptó al bloque para invocar el proceso de construcción depués de configurar el modelo Simulink. Cabe resaltar que la creación de módulo de configuración solo trabaja para las verciones R13.5 y R14 de MatLab, debido a que estas versiones del sotfware cuentan con la característica de personalizar las librerias de Simulink o crear nuevas librerías. Para las versiones anteriores es necesario incluir en el `path` de MatLab el conjunto de archivos modificados y realizar la configuración normal (*archivo objetivo*, parámetros, compilador, comentarios, etc). En el anexo B, se incluye una guía rápida para el uso de estos archivos para las versiones anteriores a las mencionadas.

En la siguiente sección, se describirá como se realiza la selección del archivo objetivo, los cuales son dos de los principales elementos de la herramienta RTW, ya que por medio de estos se define el proceso de generación de código.

2.5.2. Archivo objetivo (*target*)

El bloque generador define la configuración del archivo objetivo o *target*. El archivo de instrucciones usa el archivo objetivo de default para generar código. Este archivo de instrucciones primero almacena la cadena de variables que corresponden al `System target file, a la plantilla makefile,` y al comando `Make`, en la siguiente forma:

- `stf = 'generador.tlc';`
- `tmf = 'generador.tmf';`

[2]Los Archivos M o *M-file*, son archivos tipo texto que contiene código MatLab, que sirve para escribir funciones y archivos de instrucciones.

- `mc = 'make_rtw';`

El STF es seleccionado pasando la función `cs` como objeto y la cadena `stf` al switch de la función *target* o función objetivo, quedando:

- `switchTarget(cs,stf,[ ]);`

```
function gen_tar(cs)
        % Bloque generador de tareas para el kernel de Tiempo Real
        %       Generador base ERT
        %       Compilador: Borland C
        %       Ver.: Matlab R14, R13
        %       Descripcion: Modulo de configuracion y generacion de
        %                    codigo a traves de generador.tmf
        %       Archivos:   nombremodelo.c, nombremodelo.h,
        %                   nombremodelo_types.h y nombremodelo_data.h
% Seleccion de Archivos TLC
    stf = 'generador.tlc';
    tmf = 'generador.tmf';
    mc = 'make\_rtw';

% Seleccion del Target
    switchTarget(cs, stf, {});
    set_param(cs,'TemplateMAkeFile',tmf);
    set_param(cs,'MakeCommand',mc);

    isERT = strcmp(get_param(cs,'IsERTTarget'),'on');

% Op. de linea de comandos de TLC
    set_param(cs, 'TLCOptions','-p0');
```

Figura 2.5: Archivo de instrucciones `gen_tar.m` deconfiguración

En la figura 2.5 se muestra la sección del archivo de instrucciones `gen_tar.m`, donde se incluye la selección del *target* y *host*.

2.6. TLC Generador

Como ya se había mencionado en la sección 2.5.1, el RTW generará código de un modelo Simulink, donde el formato de código C o ADA cambia de acuerdo con el archivo objetivo seleccionado y tiene la característica básica de ofrecer la posibilidad de personalizar el código generado por Real-Time Workshop. Los archivos TLC son estructuras ASCII que controlan explícitamente el modo en que Real-Time Workshop genera el código. Editando un fichero TLC, se puede alterar el modo en que se genera el código para un bloque particular, o incluir código escrito manualmente. Como parte de la generación de código de este trabajo, se utiliza la opción de MatLab escribir archivos TLC para alterar la forma en la que el código es generado y este sea utilizado como una aplicación independiente.

El nombre del programa es `generador.tlc`; este archivo tiene la función de invocar toda la estructura TLC y generar un código más sencillo que el generado normalmente por la herramienta RTW como se mencionó anteriormente. A continuación, se describe la estructura del TLC generado.

2.6.1. Función del TLC en la generación de código

La respuesta del proceso de generación de código la da el *system target file* (STF), este sistema de archivos decide como el código debe aparecer, si hay comentarios, donde deben ir las funciones o donde deben ir las variables, etc. Por lo tanto la librería de funciones del STF contiene estructuras que soportan el proceso de generación de código. De esta manera, la salida del compilador TLC es una versión en código fuente del diagrama de bloques de Simulink. Para el caso del trabajo presentado, se propone una nueva estructura para los archivos del TLC, así, por cada bloque en el modelo de Simulink habrá una estructura que específica como traducir dicho bloque en el código del archivo objetivo que se esta usando.

2.6.2. Mapeo del Archivo de Bloque Objetivo

El mapeo del *block target file* (BTF) especifica cuál archivo objetivo se debe usar para la generación de código. El mapeo original del BTF que esta por default en MatLab, viene en *matlabroot/rtw/c/tlc/genmap.tlc*; también contiene un conjunto de archivos TLC que son comunes para todos las plataformas de archivos objetivo, junto con otro grupo de archivos que son específicos para archivos objetivo especiales[3].

[3]El listado de los archivos objetivo o *targets* especiales se presenta en el anexo A

Todos los archivos modificados para la aplicación de la presente tesis, se manejan en una carpeta externa. Por medio del cambio de este mapeo se modifica la manera en que el código es generado por el TLC para un modelo Simulink. El conjunto completo de estos archivos, invocan al archivo TLC de aplicación el cual llama al STF y al BTF para hacer la descripción en código de cada elemento del modelo Simulink. En la siguiente sección se describe la estructura de archivo TLC que se emplea para la generación de código utilizada en este trabajo de tesis.

2.6.3. Estructura del programa TLC

Como se describió anteriormente, un conjunto de archivos TLC constituyen un programa TLC y el punto de entrada para este programa es un archivo STF llamado `codegen.tlc`. Este archivo es como un *main* en un programa en lenguaje C. Este archivo TLC llama a otros 3 archivos TLC que son:

1. `genmap.tlc.` Este archivo hace el mapeo del SFT, por ejemplo: este archivo indica cuál archivo TLC va a ser usado para la generación de código de un bloque o para la adición de alguna función S o S-Function[4] de MatLab.

2. `commomsetup.tlc.` Este archivo invoca las variables globales del TLC necesarias para la generación de código.

3. `commonentry.tlc.` Este archivo es el punto de entrada común para la generación de código. Este archivo TLC decide la selección del STF basado en el archivo objetivo establecido para la aplicación.

El archivo `commonentry.tlc` es el responsable de la generación de archivos mencionados en la estructura de código. Estos archivos TLC llaman a la estructura `*wide.tlc` que activa el formato de código (C o ADA), este archivo se tomo cómo base, ya que contiene la información que necesita el TLC para definir el formato de código generado. Por ejemplo si el archivo objetivo selecionado es **ert**, entonces se llama al `ertwide.tlc` para generar código embebido. En la aplicación este archivo se renombra a `gen_wide.tlc`, el cuál llama a los siguientes archivos:

1. `gen_body.tlc.` Este archivo genera el código fuente del archivo `modelo.c`.[5]

[4]Las S-Functions son funciones desarrolladas en lenguaje C, que posibilitan la creación de controladores complejos de una forma sencilla.

[5]Sí un usuario quisiera posteriormente hacer alguna modificación o incluir alguna función en este archivo, tendría que cambiar directamente el archivo `gen_body.tlc`.

2. `gen_hdr.tlc`. Este archivo genera el código fuente del archivo `modelo.h`.

3. `gen_reg.tlc`.. Este archivo genera el código fuente del archivo `modelo_reg.h`.

4. `gen_export.tlc`.. Este archivo genera el código fuente del archivo `modelo_export.h`.

5. `gen_common.tlc`.. Este archivo genera el código fuente del archivo `modelo_common.h`.

6. `gen_prm.tlc`.. Este archivo genera el código fuente del archivo `modelo_prm.h`.

2.7. Selección del Archivo objetivo

En todo proceso de generación de código con la herramienta RTW, la selección del archivo objetivo o *target* es la parte fundamental y en la aplicación de esta tesis, el *target* que se implementa, toma el papel más importante dentro del proceso de generación, ya que por medio de este, se cambia la configuración inicial del RTW y se establece una nueva interfase para la aplicación. La siguiente sección describe como se implementa este archivo y se muestra la estructura general del mismo.

2.7.1. Archivo generador.

El archivo objetivo o *target* que se utiliza, toma como referencia el generador ERT (*Embedded Real Time Coder*), el cuál es una herramienta complementaria del RTW de MatLab. Este generador provee un marco de trabajo para la producción de código que puede ser optimizado en velocidad, uso de memoria y simplicidad. El uso de esta herramienta se enfoca principalmente en la generación código para sistemas embebidos o empotrados[6]. A continuación se describe el procedimiento que se sigue para utilización del archivo objetivo.

2.7.2. Implementación del archivo `generador.tlc`.

El archivo objetivo que se implementa se llama `generador.tlc` y usa como base la plantilla del archivo `ert.tlc` del generador ERT. Como se mencionó en la sección 2.2.1, el compilador que se usa para la generación del código es el compilador Borland C (el cual previamente se configuró como compilador inicial para todo proceso de

[6]En el anexo A, se presentan también las características fundamentales del ERT

construcción de código[7]), por lo que se utiliza el archivo `ert_bc.tmf`, el cual se renombra como `generador.tmf`, esto se hace por si se requiere utilizar el generador en otra computadora. La estructura general del archivo `generador.tlc`, se presenta en la figura 2.6, en el se muestra el pseudocódigo de las funciones que necesita el archivo `tlc` para la configuración del proceso de generación.

```
%%  SYSTLC: especificación de los archivos  *.tlc y  *.tmf
%   inclusión de path: D:/carpeta/archivo.tlc
%   Asignaci\'on de formato de código: "Empotrado" o  "Rsim"
%   Asignaci\'on de tipo de target: "RT" o "Rsim"
%   Asignaci\'on de lenguaje: "C" o "ADA"
%   Activación del proceso del construcción: "AutoBuildProcedure"
%   inclusión de puntos de entrada: "codegenentry.tlc"

BEGIN_RTW_OPTIONS
 Configuraciones Personalizables ....
 Cambio de Configuraciones de Inicio ....
 Combinaci\'on de Configuraciones Est\'andar y Personalizables ....
END_RTW_OPTIONS
```

Figura 2.6: Estructura del archivo `tlc`

Para poder usar el archivo `generador.tmf`, se establece en éste, el archivo objetivo, de la forma

```
SYS_TARGET_FILE = generador.tlc
```

De igual manera, el archivo `tlc`, se adiciona el archivo `generador.tmf` como plantilla *makefile*, de modo

También dentro del archivo `generador.tlc`, se incluyen las opciones del archivo objetivo `ert.tlc` como sigue,

Debido a que los archivos `generador.tlc` y `generador.tmf` son archivos de instrucciones de MatLab, se pueden visualizar en la herramienta de selección del archivo

[7]En el anexo C (Configuraciones) se describen los pasos para configurar el compilador de MatLab y las etapas que se deben seguir para utilizar el archivo `generador.tlc` como archivo objetivo inicial

```
%% SYSTLC: Generador ERT TMF: generador.tmf MAKE: make_rtw EXTMODE:
%%         ext_comm
%% Archivo: Generador.tlc (real-time system target file.)
%%
%selectfile NULL_FILE
```

```
BEGIN_RTW_OPTIONS
% modificaciones del generador: archivo = fullfile(matlabroot, 'rtw',
% 'c', 'ert', 'ert.tlc');
propsObj = tlc.rtwoptions(file); props = propsObj.getOptions;
rtwoptions = propsObj.combineCategories(props,rtwoptions);
END\_RTW\_OPTIONS
```

objetivo cuando se incluyen en path de trabajo de MatLab. En la figura 2.7, se muestra la pantalla del selección del archivo objetivo, donde se muestra incluído el archivo `generador.tlc` como parte de la selección.

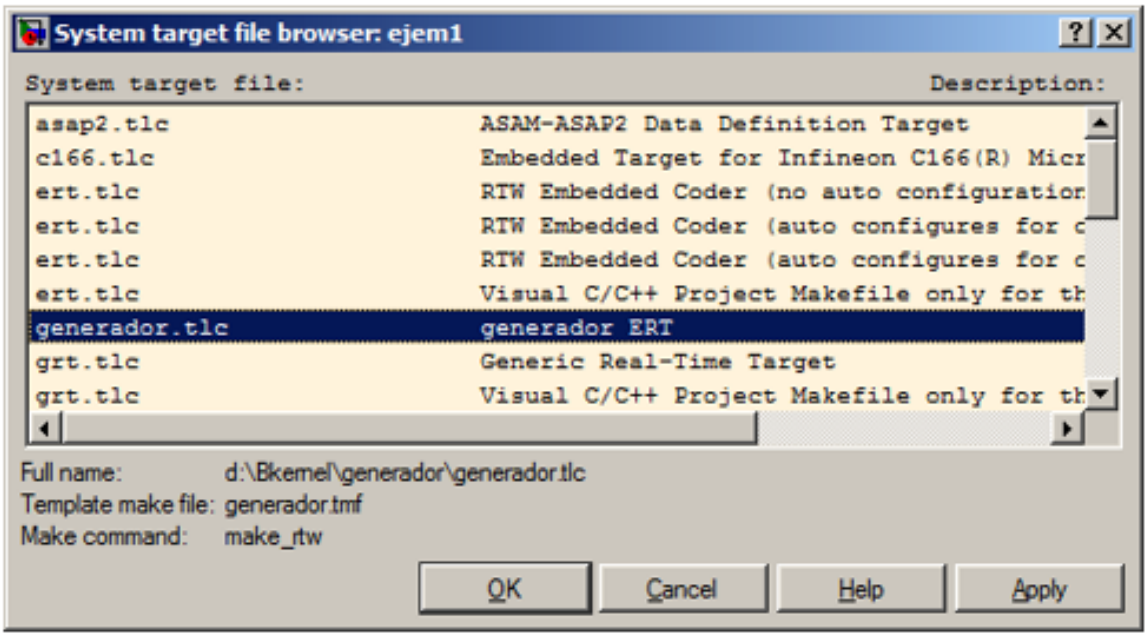

Figura 2.7: Selección del archivo objetivo *target* `generador.tlc`.

En el proceso de generación, también se incluyeron dos archivos `tlc` más, `gen_c_banner.tlc` y `gen_h_banner.tlc`. Estos archivos se utilizan para establecer el encabezado de los archivos generados e incluir funciones al código (`*.c` o `*.h`), por lo que se incluyen en el archivo `generador.tcl` con la sentencia `addincludepath`.

Finalmente para la aplicación, en el archvio `generador.tlc` se muestra a continuación. En este archivo de instrucciones, se muestra como están estructuradas las etapas de:

- `Configuraciones Personalizables`
- `Cambio de Configuraciones de Inicio`
- `Combinación de Configuraciones Estándar y Personalizables`

Etapa de **Configuraciones Personalizales**: en esta etapa se incluyen los comandos de invocación del archivo objetivo `generador.tlc` y los mensajes hacia la linea de comando de MatLab.

Etapa de **Cambio de Configuraciones de Inicio**: en esta etapa se incluyen los comandos de invocación para las opciones del ERT y los archivos objetivo de generación del proceso de construcción de código del archivo objetivo implementado. Los comandos como `ERTSrcFileBannerTemplate` y `ERTCreateCustomFiles`, sirven como puntos de entrada para los archivos *banner*, los cuales son archivos de configuración de código como se mencionó anteriormente.

Etapa de **Combinación de Configuraciones Estandar y Personalizables**: en esta etapa se incluyen los comandos de invocación para el archivo objetivo.

2.8. Sumario

Como se mencionó en este capítulo, el proceso se enfoca principalmente en la generación de código fuente a partir de un diagrama de control generado en Simulink, utilizando el bloque desarrollado que simplifica el proceso de construcción de código.

La generación de código se realiza por medio de la implementación de archivos TLC (Target Languaje Compiler), los cuales son utilizados en el proceso de generación. El archivo archivo objetivo `generador.tlc`, se utliza para configurar el proceso

```
BEGIN_RTW_OPTIONS

% Configuraciones Personalizables

oIdx = 1; rtwoptions(oIdx).prompt = 'Proceso Generador';
rtwoptions(oIdx).type = 'Category'; rtwoptions(oIdx).enable ='on';
rtwoptions(oIdx).default = 2; % numero de items bajo esta categoria

rtwoptions(oIdx).popupstrings = ''; rtwoptions(oIdx).tlcvariable =
''; rtwoptions(oIdx).tooltip = ''; rtwoptions(oIdx).callback = '';
rtwoptions(oIdx).opencallback = ''; rtwoptions(oIdx).closecallback
= ''; rtwoptions(oIdx).makevariable = '';

oIdx = oIdx  1; rtwoptions(oIdx).prompt = 'Incluye Generador.tlc';
rtwoptions(oIdx).type = 'Checkbox';
rtwoptions(oIdx).default = 'on';
rtwoptions(oIdx).tlcvariable = 'genHdr';
rtwoptions(oIdx).tooltip ='Activa proceso de generacion',...sprintf('\n'),

oIdx = oIdx  1; rtwoptions(oIdx).prompt = 'Incluye Generador Pragma';
rtwoptions(oIdx).type = 'Checkbox';
rtwoptions(oIdx).default = 'on';
rtwoptions(oIdx).tlcvariable = 'genPragmas';
rtwoptions(oIdx).tooltip =('Incluye gen. pragma code',...sprintf('\n'),
                                    'manejo de errores');
```

de generación y producir un código más sencillo y manejable.

Así, de acuerdo a lo mencionado, se puede resaltar los siguientes puntos:

- El uso de la herramienta ERT como generador de código fuente, desde un diagrama Simulink, permite que el código pueda ser utilizado para el diseño de sistemas de control con características de tiempo real y procesos que involucren procesos de aplicación específica como la de los sistemas empotrados.
- El módulo generador de Simulink desarrollado, se construyó con la finalidad de agilizar la configuración del generador RTW, permitiendo tener en un solo bloque las configuraciones necesarias para producir el código de cada modelo

```
% Continuacion.... BEGIN_RTW_OPTIONS
% Cambio de Configuraciones de Inicio
oIdx = oIdx  1; rtwoptions(oIdx).default = 'on';
rtwoptions(oIdx).tlcvariable = 'ERTCustomFileBanners';

oIdx = oIdx  1; rtwoptions(oIdx).default = 'gen_c_banner.tlc';
rtwoptions(oIdx).tlcvariable = 'ERTSrcFileBannerTemplate';

oIdx = oIdx  1; rtwoptions(oIdx).default = 'gen_h_banner.tlc';
rtwoptions(oIdx).tlcvariable = 'ERTHdrFileBannerTemplate';

oIdx = oIdx  1; rtwoptions(oIdx).default = 'on';
rtwoptions(oIdx).tlcvariable = 'ERTCreateCustomFiles';

oIdx = oIdx  1; rtwoptions(oIdx).default = 'generador_file_process.tlc';
rtwoptions(oIdx).tlcvariable = 'ERTCustomFileTemplate';
```

```
% Continuacion.... BEGIN_RTW_OPTIONS
% Combinacion de Configuraciones Estandar y Personalizables

file = fullfile(matlabroot, 'rtw', 'c', 'ert', 'ert.tlc');
propsObj = tlc.rtwoptions(file); props = propsObj.getOptions;
rtwoptions = propsObj.combineCategories(props,rtwoptions);

%-----------------------------------------------------%
% Configuracion de ajustes de la generacion de codigo %
%-----------------------------------------------------%
rtwgensettings.BuildDirSuffix = '_generador_rtw';

END_RTW_OPTIONS
```

simulink. Además, se pudo obtener un código más manejable, el cual se describirá en el siguiente capítulo. Este código puede utilizarse fuera del ambiente de MatLab con el menor de modificaciones. Así mismo, el bloque generador define la configuración del archivo objetivo. El archivo de instrucciones que utiliza, usa el *target* de default para generar código. El archivo de instrucciones invoca al

`System target file, a la plantilla makefile,` y al comando `Make` para configurar y activar el proceso de generación.

- Como parte de la generación de código de este trabajo, se utiliza la opción de MatLab escribir archivos TLC para alterar la forma en la que el código es generado y este sea utilizado como una aplicación independiente. El nombre del programa es `generador.tlc`; este archivo tiene la función de invocar toda la estructura TLC y generar un código más sencillo que el generado normalmente por la herramienta RTW como se mencionó anteriormente.

- El archivo *target* que se implementa se llama `generador.tlc` y usa como base la plantilla del archivo `ert.tlc` del generador ERT y el compilador que se usa para la generación del código es el compilador Borland C con el archivo `ert_bc.tmf`, el cual se renombra como `generador.tmf`, esto se hace por si se requiere utilizar el generador en otra computadora.

En el siguiente capítulo, se presentará como se realizó la interfaz para poder usar el código generado en otro ambiente, en específico en un micro-kernel de tiempo real, así como también, se describirá las características principales de este, desde su arquitectura hasta el manejo de procesos.

Capítulo 3

Descripción del sistema

3.1. Introducción

Como ya se mencionó anteriormente, con Simulink es posible generar código C correspondiente a un modelo por medio de la herramienta RTW, lo cual es sencillo con tan solo configurar el menú de parámetros de simulación de la herramienta y seleccionando un *target* específco. Sin embargo, el uso de la aplicación no resulta tan simple como el proceso de generación, ya que sí el usuario coloca simplemente todo el código generado dentro de un compilador, esté no funciona, debido a las funciones especializadas y al número de dependencias de MatLab que se generan para el código.

En este capítulo, se presentará como se realizó la configuración del archivo *target* `generador.tlc`, la estructura del código generado y la implementación de la interfaz para poder usar el código producido por el módulo generador de tareas en otro ambiente, en específico en un micro-kernel de tiempo real, así como también, se describirán las características principales de este, desde su arquitectura hasta el manejo de procesos. De igual manera, se describe la arquitectura general del sistema a partir del uso del módulo generador hasta la inclusión del código como tárea de control dentro del micro-kernel para así finalmente en el capítulo 5, se presenten los resultados obtenidos de la apliación.

3.2. Configuración de Archivos

El proceso de construcción permite suministrar archivos de enlace (*hook files*) opcionales, que se ejecutan en puntos específicos en la generación de código y en el proceso *make* o adicionando acciones específicas en los *targets* en el proceso de construcción.

En esta sección, se describe el archivo de enlace M, de nombre `gen_make_rtw_hook.m`. Este archivo implementa una función que envía a una acción específica dependiente de un método que modifica la generación de código. El proceso construcción llama automáticamente al archivo `gen_make_rtw_hook.m`, el cual pasa los argumentos del método de construcción al proceso de generación. A continuación se describen el método y los argumentos que se utilizan en el proceso de construcción.

3.2.1. Configuración y Especificación del Archivo objetivo

La función de enlace implementada en el archivo M, tiene la siguiente estructura:

```
function STF_make_rtw_hook(hookMethod, modelName, rtwRoot, templateMakefile, buildOpts, buildArgs)
```

Para trabajar con el archivo M se especificó la información del *target*, donde fue necesario primero especificar el tamaño de la palabra para el tipo de datos de clase entero (**short**, **int** y **long**) y la propiedad específica de la implementación en C para el *target* que se utilizó (tal como el comportamiento de sobreflujo de enteros), esto es debido a que en el lenguaje C no se específica esta información. En el caso del tamaño de palabra, este se especificó de acuerdo a las características del compilador **BorlandC**; en la tabla 3.1 se muestra esta especificación para ambos casos.

Sentencia valor (tamaño de palabra)	Sentencia valor (tamaño de palabra)

value.CharNumBits = 8;
value.ShortNumBits = 16;
value.IntNumBits = 32;
value.LongNumBits = 32;
	value.ShiftRightIntArith = true;
	value.Float2IntSaturates = false;
	value.IntPlusIntSaturates = false;
	value.IntTimesIntSaturates = true;

Tabla 3.1: Tamaño de palabra definido

En conjunto para automatizar el proceso de generación, se utilizan dos archivos

M adicionales, junto con el archivo `gen_make_rtw_hook.m`; estos son `uset_param.m` y `uget_param.m`, que sirven para establecer la configuración del modelo generado en simulink, que es parte del proceso de generación. Estos archivos configuran todas las opciones de la generación de código del RTW, eliminando la necesidad de hacer una configuración manual cada vez que se hace un modelo, ahorrando tiempo y eliminando errores. Así, dentro del archivo `gen_make_rtw_hook.m`, se implementa la siguiente función:

```
function EntryConfiguration(model)
```

La función `function Entry` llama a la función `uset_param.m`, para establecer todos los parámetros del modelo. La lista entera de las opciones de configuración, se muestra en el anexo C. Esta función es llamada desde la etapa de entrada '`entry`' del proceso de construcción, mientras que el punto de salida guarda las características del modelo anterior.

```
switch hookMethod case 'entry'
    % Proceso llamado al emperzar la generacion de codigo.
    disp('Configuracion del modelo para generador.tlc.');
    uset_param(modelName,'BackupSettings');
    try
        EntryConfiguration(modelName);
    catch
        uset_param(modelName,'RestoreSettings');
        error(lasterr)
    end
case 'exit'
    % Proceso llamado al finalizar la generacion de codigo.
    % Todos los argmentos son validos en esta etapa
    disp('Restoring model configuration.');
    uset\_param(modelName,'RestoreSettings');
end
```

Figura 3.1: Configuración del modelo para el *target* `generador.tlc`

La función `function Entry`, esta implementada dentro de un bloque *try/catch*, de modo que en caso de que se produzca un error en el proceso de construcción, los

ajustes de los modelos sean restaurados. En la figura 3.1, se muestra la parte del punto de entrada y salida. En este se muestra el punto de entrada donde se llama a la función `function Entry`.

Finalmente, en la figura 3.2, se muestra un fragmento de código del archivo de configuración `gen_tar_file_process.tlc`, donde se establece la forma de la salida con un mensaje para la consola. Esto se hace con ayuda de la función `MdlOutput`.

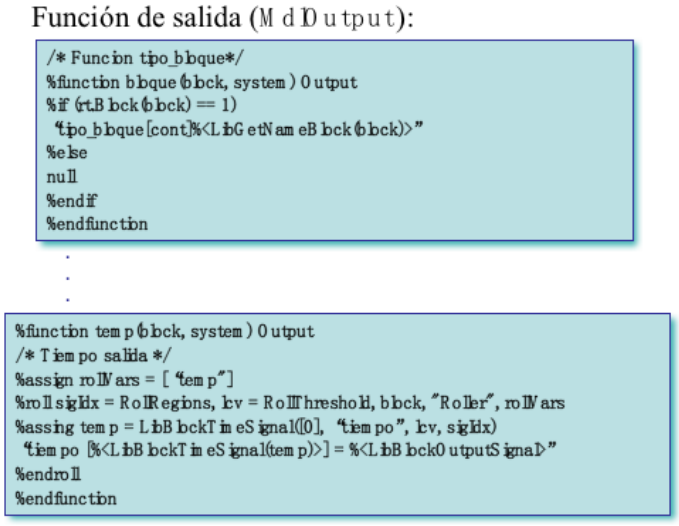

Figura 3.2: Configuración de la función `MdlOutput`

3.2.2. Archivos modificados

A continuación se enlistan todos los archivos que se modificaron para configurar el proceso de generación para el archivo *target* `generador.tlc`.

- Archivos (`ert.tlc`) → `generador.tlc`.
- Archivos (`ert_bc.tlc`) → `gen_tar_bc.tlc`.
- Archivos (`ert_file_process.tlc`) → `gen_tar_file_process.tlc`.
- Archivos (`ert_c_banner.tlc`) → `gen_tar_c_banner.tlc`.
- Archivos (`ert_h_banner.tlc`) → `gen_tar_h_banner.tlc`.

- Archivos (`#.tmf`) → `generador.tmf`

3.3. Código Generado

Cada vez que se inicia el proceso de generación RTW con el módulo generador, este prepara el modelo Simulink, en varias etapas para la generación de código. A continuación se enumeran cada una de estas etapas.

1. Primero, se identifica las variables de las señales existentes en el modelo.
2. Se identifican las líneas de las señales asociadas con esas variables.
3. Se toman las propiedades de las señales.
4. Se asigna un identificador y una clase de almacenamiento[1] (*Storage class*).
5. Se asigna una clase de almacenamiento para cada señal leída y se exporta como variable global.
6. Identifica las operaciones y los parámetros en el modelo. Estos son parámetros en el modelo que pueden ser cambiados desde el programa principal por el usuario. (por ejemplo: `input_parameters` al modelo).
7. Crea los parámetros del modelo en el espacio de trabajo de MatLab.
8. Por ultimo invoca al proceso TLC y el *target* `generador.tlc` para la generación de código.

3.3.1. Formato del código generado

El RTW crea un directorio en el directorio de trabajo (*definido por el usuario*), donde coloca el código generado del modelo Simulink. Este directorio de trabajo también contiene los archivos objetos (`*.obj`), un archivo `makefile` y otros archivos más que se crean durante el proceso de generación. El nombre de inicio del directorio de trabajo corresponde al nombre el modelo Simulink con la extensión `_rtw`.

[1]Las tipos de clase de almacenamiento para las señales se clasifican en 4; en el Anexo B, se presentan cada una de ellas.

Una vez generado el código, este contiene 5 archivos de cabecera (*.h) y un archivo C, el cuál describe el algoritmo del modelo[2]. A continuación, se presenta la descripción de cada uno de estos archivos.

- `modelo.c`. Este archivo contiene todos los puntos de entrada del algoritmo implementado en el modelo (`Mdlinitialize`, `MdlStart`, etc.) y define el algoritmo de implementación del modelo en lenguaje C.
- model_private.h. Este archivo contiene las macros locales y los datos locales que son requeridos por el modelo y subsistemas que contenga. Este archivo esta incluído en los archivos fuente generados en el modelo, por lo que no hubo necesidad de adicionar este archivo al código generado.
- `model.h`. Archivo principal de cabecera. Este archivo declara las estructuras de datos del modelo y define una interfaz pública para los puntos de entrada del modelo y las estructuras de datos. Normalmente en este archivo se incluye una interfaz en tiempo real para el modelo, pero con el bloque generador, se elimina la generación de esta parte de código, debido a que la estructura de tiempo real la maneja el micro-kernel.
- `model_types.h`. Este archivo proporciona declaraciones para la estructura de datos del modelo en tiempo real y la estructura de datos de los parámetros. Éstos son necesarios para la declaración de las funciones reutilizables. El archivo `model_types.h` esta incluído en todos los archivos de cabecera generados en el modelo.
- `modelo_prm.h`. Contiene todos los parámetros utilizados en el modelo.
- `modelo_export.h`. Contiene las declaraciones externas para toda variable que valla a utilizarse en la aplicación externa.

Código generado por el RTW

La principal característica de los archivos generados con el RTW de MatLab, consiste en la implementación de estructuras que simulan el manejo de interrupciones para dispositivos de tiempo real. Estas estructuras son comentarios en el código que indican donde deberían ir estas funciones, como la muestra el siguiente código.

[2]En este caso, para fines de identificación 'modelo' corresponderá al nombre del diagrama de bloques contruído en Simulink

```
#include "model.h"
#include <stdio.h>
#include "model1_dt.h"
    .
    .
/* Real-time model */
    rtModel_model1 model1_M_;
    rtModel_model1
    *model1_M = &model1_M_;

/* Model output function */
     .
     .
```

Figura 3.3: Código producido por el ERT.

Tomemos como ejemplo el código anterior de la figura 3.3, el cual corresponde a un fragmento del archivo `*.c` generado por el ERT de inicio. En este archivo, se muestra como se establece parte de la estructura que indica el manejo de interrupciones y el cambio de contexto (`rtModel`), pero no define la función completa, a menos que se usen los *targets* Tornado o xPC [3]. Estas funciones, no están completadas debido a que son definidas así por defecto para que el usuario implemente sus propias funciones.

En el caso de la aplicación presentada en esta tesis, se modifica la inclusión de estas funciones de manejo de interrupciones y cambio de contexto, de manera que estas se agregan como comentarios del código. Para ejemplificar esto, se selecciona un archivo `C` (figura 3.4), generado con el proceso de construcción utilizando el archivo `generador.tlc`.

En este archivo, se generan todas las funciones que describen el comportamiento del modelo como se mencionó en la sección 2.4.2, pero las funciones que indican el manejo de interrupciones y el cambio de contexto se agregan comentadas con el fín de indicar al usuario como podrían implementarse si se quisiera utilizar el código en otro sistema o kernel de tiempo real, o bien, en algón tipo especial de procesador.

[3]Tornado y xPC, son *targets* ya definidos en MatLab que funcionan exclusivamente en estos ambientes, donde se pueden trabajar con un solo modelo de Simulink en tiempo Real.

```
#include "model.h" #include "model_private.h"

#include "model_prm.h" #include <stdio.h> #include <limits.h> . .
/*Funciones de tiempo real comentadas*/

/*rtModel_model model_M_;*/ /*rtModel_model *model_M =&model_M_;*/
. . /* Function de salida */

void model_output(int_Ttid) { /*Algoritmo del modelo*/
  real_T rtb_SineWave;
  /* Sin: '<Root>/Sine Wave' */
  rtb_SineWave = model_P.SineWave_Amp *
    sin(model_P.SineWave_Freq * model_M->Timing.t0
    model_P.SineWave_Phase)  model_P.SineWave_Bias;
  /* Gain: '<Root>/Gain' */
  model_B.Gain = rtb_SineWave * model_P.Gain_Gain;
  /* Gain: '<Root>/Gain1' */
  model_B.Gain1 = rtb_SineWave * model_P.Gain1_Gain;
}
```

Figura 3.4: Código modificado

Esto se logra modificando el arhivo `generador.tlc`, en la parte de "configuraciones personalizables" en el archivo *target* como se mostró en el capítulo 2. Con las demás modificaciones hechas al generador se logra un código más manejable y comentado, facilitando su lectura. Además, el código generado incluye nombres de bloques y etiquetas de señales del modelo Simulink, permitiendo seguir con el código el modelo. Las etiquetas del código generado incluyen un prefijo que corresponde al nivel de jerarquía donde se sitúa el bloque específico[4].

En la siguiente sección se presenta como se realiza el modelo del sistema para el proceso de generación donde se describe la arquitectura de la implementación de la interfaz para poder usar el código generado producido por el módulo generador de

[4]En el capítulo 5 se muestra un proceso de generación y los códigos generados se incluyen en el anexo D

tareas en otro ambiente, en específico en un micro-kernel de tiempo real.

3.4. Modelo del Sistema

Muchos de los procesos de control con características de tiempo real, consisten de aplicaciones embebidas, donde un sistema embebido (a veces traducido del inglés como embebido, empotrado o incrustado) es un sistema informático de uso específico construído dentro de un dispositivo mayor. Estos sistemas integrados se utilizan para usos muy diferentes de los usos generales para los que se emplea procesadores específicos. Por lo que la combinación de estos elementos (*software y hardware*) depende de la estructura del código y del tipo de procesador que se utilice. El simular este tipo de procesos, resulta entonces fundamental para poder desarrollar aplicaciones de control más funcionales.

La aplicación en esta tesis presenta una alternativa de simulación utilizando la combinación de MatLab y un micro-kernel de tiempo real desarrollado como trabajo de investigación escolar. Utilizando MatLab, el usuario puede diseñar, simular y generar código de un modelo o sistema de control y utilizarlo en una plataforma, ambiente o procesador específico. Para el caso de esta aplicación se utiliza MatLab para producir un código más manejable, simplificado y sin dependencias con la ayuda módulo generador **Bkernel** (cap 2), el cual lo conforman varios archivos (sección 2.6.3) de configuración, que establecen las condiciones de generación de código en base al archivo *target* `generador.tlc` (sección 2.7.2).

El modelo del sistema, lo constituye el generador de código y el micro-kernel de tiempo real, el cual se detallará en las siguientes secciones. En la figura 3.5, se muestra un diagrama a bloques del modelo del sistema. La aplicación contempla la generación del código y la ejecución de este se realiza fuera del ambiente, específicamente en un micro-kernel como un proceso más del mismo.

El código una vez generado, se maneja como tarea de control, esto es debido a que lo que pretende es simular un proceso de control como una tarea de tiempo real, dentro de un sistema de control mayor con características de tiempo real, proporcionadas por el micro-kernel de tiempo real. El micro-kernel puede manejar más de un proceso o tarea.

La función principal de micro-kernel es la de permitir la creación y ejecución concurrente de varios procesos. Proporcionando funciones “primitivas”del sistema) que

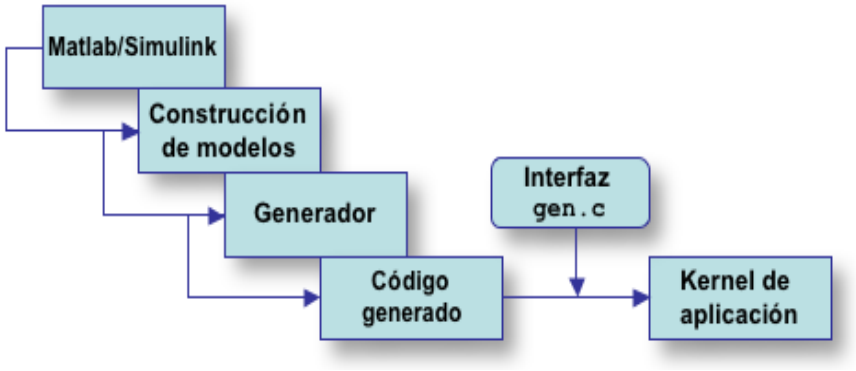

Figura 3.5: Modelo del Sistema.

permitan la creación, eliminación, sincronización y comunicación entre procesos.

El micro-kernel fue desarrollado para funcionar "sobre" MS-DOS. Se le ha adicionado una librería para el uso de un entorno gráfico independiente al Sistema Operativo. Las funciones del sistema operativo MS-DOS no son usadas durante la ejecución del micro-kernel, las funciones de bajo nivel están escritas en ensamblador.

Este en conjunto con el módulo generador ofrecen una alternativa de simulación para el estudio de sistemas de control con características de tiempo real. Los modelos Simulink se ejecutan en el micro-kernel con una política de planificación RR (*Round Robin*), permitiendo visualizar como se comportaría el modelo de control en conjunto con otros procesos de Simulink que se ejecutan concurrentemente como tareas de control. Además, el sistema de aplicación ofrece la posibilidad de obtener un código m·s accesible para utilizarlo en alguna otra plataforma, es decir, otra plataforma diferente a la que se ejecuta MatLab. En la siguiente sección, se describe como esta constituída la arquitectura de la aplicación, donde se muestra todos los elementos que conforman el sistema completo, desde la generación de código hasta el micro-kernel.

3.5. Arquitectura del Sistema

La arquitectura general de la aplicación se muestra en la figura 3.6. La estructura muestra un diagrama de control realizado en Simulink, al cual se le adiciona el bloque generador de tareas, el cual genera código en C utilizando el RTW (Real Time Workshop) toolbox de MatLab. La herramienta RTW permite generar código o un

programa ejecutable (C, .exe, .dll) desde un diagrama de Simulink, donde el diagrama de Simulink funciona como la interfaz de usuario (GUI) [16]. Posteriormente este se exportará al micro-kernel, donde se ejecutará por medio de una API como una tarea de control o proceso; permitiendo ejecutarse junto con varios procesos de forma concurrente.

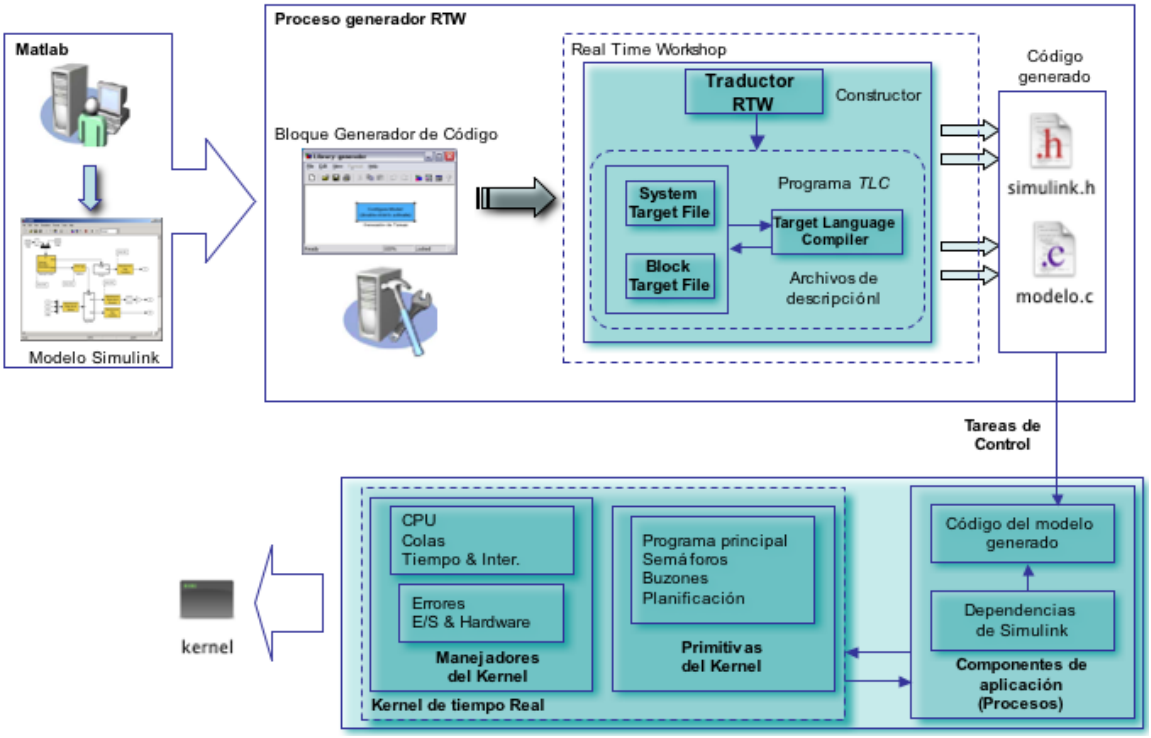

Figura 3.6: Arquitectura del Sistema.

El esquema de la arquitectura esta compuesta de los siguientes elementos:

- Módulo generador. Esta etapa constituye el proceso de construcción del código.
- Código generado. Esta etapa constituye el código generado, construído para trabajar sobre la plataforma MS-DOS.
- Micro-kernel e interfaz de aplicación. Esta etapa corresponde a la interfaz de aplicación donde se ejecuta el código producido por el módulo generador.

- Ejecución de proceso. Esta es la última etapa donde se visualiza la ejecución de las tareas de control o los procesos del micro-kernel.

1. **Módulo generador.**
   Este módulo es el más importante del sistema de aplicación. El usuario utiliza MatLab/Simulink para diseñar el sistema o módulo de control por medio de la simple unión de bloques funcionales, la mayoría disponibles en varias bibliotecas preconfiguradas, dentro de una sencilla interfaz gráfica de usuario (GUI). Una vez generado el modelo se le adiciona la librería preconfigurada `gen_tar.mdl`, el cual es el módulo generador **Bkernel** descrito en la sección 2.5. Este módulo al activarlo invoca al proceso de construcción del código con el *target* `generador.tlc`, donde se establece la estructura de configuración y se modifica el formato inicial en el cual la herramienta RTW está configurada. La construcción de código se realiza en base al generador ERT que perimite desarrollar código para sistemas embebidos sin dependencias de MatLab.

2. **Código generado.**
   Una vez generado el código, este contiene 5 archivos de cabecera (*.h) que contienen las macros locales y los datos locales que son requeridos por el modelo y subsistemas que contenga y un archivo que describe el algoritmo del modelo (*.c). Posteriormente, el RTW crea un directorio en el directorio de trabajo, donde coloca el código generado del modelo Simulink. Este directorio de trabajo también los archivos objetos (`*.obj`), un archivo `makefile`. El nombre de inicio del directorio de trabajo corresponde al nombre el modelo Simulink con la extensión `_rtw`.

   En el código, se generan todas las funciones que describen el comportamiento del modelo, pero las funciones que indican el manejo de interrupciones y el cambio de contexto se generan comentadas. Con las demás modificaciones hechas al generador se logra un código más manejable y comentado, facilitando su lectura. Además, el código generado incluye nombres de bloques y etiquetas de señales del modelo Simulink, permitiendo seguir con el código el modelo. Las etiquetas del código generado incluyen un prefijo que corresponde al nivel de jerarquía donde se sitúa el bloque específico.

3. **Micro-krenel e interfaz de aplicación.**
   El código una vez generado, se maneja como tarea de control, esto es debido a que lo que pretende es simular un proceso de control como una tarea de tiempo real, dentro de un sistema de control mayor con características de tiempo real,

proporcionadas por el micro-kernel de tiempo real. El micro-kernel puede manejar más de un proceso o tarea. La función principal de micro-kernel es la de permitir la creación y ejecución concurrente de varios procesos. Proporcionando funciones ("primitivas"del sistema) que permitan la creación, eliminación, sincronización y comunicación entre procesos.

La capa principal, llamada Micro-kernel, consta de la definición e inicialización de las variables y estructuras principales del sistema. Es la primera parte que se ejecuta del sistema, después de inicializar las variables y estructuras, quita el control a MS-DOS y pone en funcionamiento los manejadores del sistema.

El micro-kernel se encarga de crear el primer proceso a ejecutar dentro del kernel, de este se desprenden los demás procesos del sistema. Cuando el sistema debe terminar, el micro-kernel se encarga de regresar el control a MS-DOS y reestablecer el estado original de la pantalla.

La interfaz de aplicación donde se ejecutan los procesos de Simulink, se llama `gen.c`. Este archivo contiene todas las dependencias (*librerías*) necesarias y las funciones de ejecución, que necesitan todos los códigos generados por la herramienta ERT. Dentro del archivo `gen.c`, se pasa la cadena del nombre del modelo o los modelos de Simulink, dentro de las funciones de activación, implementadas para incluir y ejecutar el código generado en el micro-kernel. El código se inserta manualmente dentro del diretorio del micro-kernel y al compilar toda la aplicación, se ejecutan todos los procesos habilitados en el micro-kernel. En la siguente sección se presenta como esta definido el archivo `gen.c` y como se maneja la ejecusión de los procesos.

4. **Ejecución de proceso.**
   Esta etapa consta de 2 capas del micro-kernel de aplicación: la capa de **Primitivas del Sistema** y la capa de la **Librería Gráfica**. La capa de las *primitivas del Sistema*, es la parte accesible del kernel desde los procesos. Proporciona una interfaz libre de detalles de implementación de las acciones que se le permiten realizar a un proceso. Las funciones reciben solo los argumentos necesarios para la realización de la tarea solicitada y no da pie a configuración ni a opciones de manejo de los recursos disponibles. Las primitivas que el sistema proporciona son:

- **Primitivas de procesos.** Creación (activa) y eliminación (elimina) de procesos.
- **Primitivas de sincronización.** Creación de semáforos (creaSemaforo), wait (espera) y signal (señal).
- **Primitivas de tiempo.** retrasa, la cual permite la suspensión de un proceso durante un lapso de tiempo determinado.
- **Primitivas de comunicación.** Creación de buzones (creaBuzon), envío (envía) y recepción (recibe) de mensajes.

La capa de la *librería gráfica*, es la parte del sistema que tiene un conjunto de funciones, casi todas accesibles para el usuario. Al no formar parte del control del procesador y solo manejar el formato de despliegue, esta no forma parte del sistema. Se llama librería por que es un conjunto complementario de funciones, que se consideró importante, para evitar el uso de las funciones de despliegue en modo texto proporcionadas por el MS-DOS a través de la interrupción 21H.

La librería está estructurada de manera de que se pueda cambiar el número y tamaño de las ventanas, así como el uso de distintos tipos de fuentes. La velocidad de este entorno gráfico es la mayor posible, dado que accede directamente la memoria de video para realizar cualquier despliegue. De esta manera en pantalla se despliegan 6 procesos que se ejecutan de manera simultánea, en donde se muestra el comportamiento de cada modelo que se incluye en el archivo `gen.c`.

3.6. Interfaz de Aplicación

La interfaz de aplicación, es la parte del sistema donde se realiza la conexión funcional entre MatLab/Simulink y el micro-kernel. Esto es, a partir de un diagrama de Simulink (elaborado por el usuario), el módulo `gen_tar` construye el código del modelo desarrollado. Este código es la descripción del modelo en cuestión y es manejado por la interfaz de aplicación, para que se pueda utilizar en el micro-kernel de tiempo real.

Esta interfaz es un programa en C que contiene dependencias y funciones que necesitan los códigos generados por el ERT. El archivo se llama `gen.c`, y su función es la de integrar y ejecutar el código de Simulink en el micro-kernel de tiempo real. Una vez construído el código fuente se seleccionan los 6 archivos que describen el

Tabla 3.2: Archivos de aplicación

Archivos	
`prefijo.c`	`prefijo_types.h`
`prefijo.h`	`prefijo_prm.h`
`prefijo_private.h`	`prefijo_export.h`
*prefijo = nombre del modelo Simulink	

comportamiento del modelo. En el cuadro 3.2 se enlistan los archivos que se utilizan para incluirlos al micro-kernel. Estos archivos son generados para cada modelo que se construya en Simulink, con la diferencia de que el nombre del archivo comienza con un prefijo que indica el nombre del modelo.

Los archivos de descripción, se copian de la carpeta generada del proceso de contrucción, a una carpeta de archivos llamada `codsim` creada dentro de la estructura de archivos del micro-kernel para almacenar todos los archivos de los modelos que se utilizen en la aplicación. En la figura 3.7, se ilustra este paso. A continuación se describe la estructura del archivo `gen.c`, donde se muestran las dependencias y funciones que se necesitan para ejecutar el código de Simulink, así mismo, se describe el archivo que maneja las tareas de control como procesos del kernel.

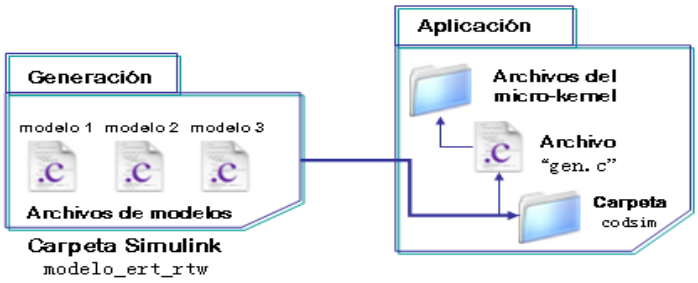

Figura 3.7: Carpetas de aplicación.

3.7. Estructura del archivo `gen.c`

El archivo `gen.c`, esta generado de modo que el usuario solo ingrese el nombre del modelo en el código para luego después activar la tarea de control. El archivo `gen.c` consta de 3 etapas: 1) Registro de modelos, 2) Funciones de ejecución y 3) Funciones de activación.

Registro de modelo.

En esta etapa, es donde se incluyen los archivos `*.h` correspondientes al código generado de los modelos, que se necesitan para ejecutarse, como los presentados en tabla 3.2. También en esta etapa se incluyen las definiciones y librerías necesarias, como es el caso de la libreria `rt_sim.h`, que contiene todas las estructuras, parámetros y definiciones de las funciones de ejecución que se incluyen en este archivo.

```
/*************************************************
    Archivo         :gen.c
    DESCRIPTION     :Aplicacion Simuinlink kernel
    Plataformas     :Windows, Unix
*************************************************/
#define RT
#include "rt_sim.h"
#include <stdio.h>

/**** INCLUIR MODELOS *****/

#include "impulso.h"
#include "seno.h"
#include "filtro1.h"
#inluce  "filtro2.h"
//#include ".h"
/*****************************************/
```

Figura 3.8: Archivo `gen.c`(a)

Como se mecionó anteriormente, en esta etapa se incluye el nombre del archivo `*.h` del modelo Simulink. El cuadro 3.8, muestra como están incluídos los nombres de

los archivos *.h del modelo generado en Simulink. De esta manera el usuario puede agregar los archivos *.h que valla a utilizar en el micro-kernel.

Funciones de ejecución.

En esta etapa se utiliza las funciones de ejecución que se generan para ejecutar el código del modelo. Estas funciones son generadas en el código y están contenidas en el archivo *.c del modelo. La utilización de estas funciones es similar al uso que se aplica en el archivo modelo_main.c[5]. Este archivo muestra de manera descriptiva, como ejecutar el código construído por la herramienta. Las funciones de ejecución se describieron en la sección 2.4.2 y estas son:

- rt_OneStep
- MdlInitializeSizes
- MdlInitializeSampleTimes
- MdlStart
- MdlOutputs
- MdlUpdate
- MdlTerminate

Función de activación.

Esta es la etapa donde se llama al modelo Simulink. La función de activación se llama "void modelo()"; en esta función se define la variable S, donde se pasa la cadena del nombre del modelo Simulink de la forma

```
S = MODELNAME(tfStr, "nombre", tmpStr2);
```

La posición del argumento "nombre", indica que ahí, se debe agregar el nombre del modelo Simulink para completar la función. En esta etapa se inicializa la estructura del reloj, que se define en el rt_sim.h y en un clclo For se dibuja la respuesta del sistema de acuerdo al tiempo de muestreo del modelo generado. En el cuadro 3.9, se muestra la función modelo. El ciclo for de la función modelo muestra como se

[5]El archivo modelo_main.c se genera de manera opcional durante el proceso de contrucción del código. Esto se logra configurando el panel del control de la herramienta RTW

despliega la respuesta, por medio de la función `dibpixel`, implementada dentro del micro-kernel para realizar las representaciones gráficas. En el capítulo 5, se describe la función `dibpixel`.

```
void modelo() {
   S = MODELNAME(tfStr, "nombre_modelo", tmpStr2);
   int y, i;
   UploadCheckTrigger();
   if NUMST==1
        UploadBufAddTimePoint(S, 0);
   elseif
    {
        int i;
        for (i=0; i<NUMST; i++) {
            if (ssIsSampleHit(S, i, unused)) {
                UploadBufAddTimePoint(S, i);
            }
        }
    }
   else{
    for(i=0;i<=360;i++) {
        ssIsSampleHit = rt_UpdateDiscreteTaskSampleHits(S)*P/100.0;
        y=ssIsSampleHit*CONTINUOUS_SAMPLE_TIME;
        dibPixel(30*(NUMST) + v->x +angulo, 85 + v->y - y, 14);

    }
```

Figura 3.9: Archivo `gen.c` (b)

3.7.1. Proceso Simulink

El proceso Simulink, es la parte donde se incluye la función que invoca al código generado por la herramienta para su ejecución como tarea de control dentro del kernel. El proceso Simulink se incluye dentro del archivo `proceso.c`, el cual contiene todos los procesos que se vallan a ejecutar dentro del kernel.

```
/*====================================
=====  PROCESO SIMULINK 1     ==========
======================================*/
void proc1(void) {

    Ventana *v = asignaVentana();   /* Solicita una ventana. */
    dibgraf(v);

    while(1){                       /* Ejecucion del modelo */
     modelo1(v);
    };
    elimina();                      /* Termina proceso */
}
.
.
.
/*====================================
=====  PROCESO SIMULINK n    ==========
======================================*/

void proc_n(void) {

    Ventana *v = asignaVentana();   /* Solicita una ventana. */
    dibgraf(v);

    while(1){modelo2(v);};          /* Ejecucion del modelo */
    elimina();                      /* Termina proceso */
}
```

Figura 3.10: Procesos Simulink

Para definir los procesos Simulink dentro del kernel, se definieron 5 funciones (`proc1`,...,`proc5`). Para el caso de la aplicación de esta tesis, solo se difinieron 5 funciones, pero también se tiene la posibilidad de agregar más funciones de forma que sí el ususario desea agregar más funciones para la ejecución, lo puede hacer con tan solo copiar estas funciones el número de procesos que se vallan a utilizar. Estas funciones llaman a las funciones del archivo `gen.c`. En el cuadro 3.10, se muestra el

código de las funciones `porc #`. A estas funciones se les asigna una ventana dentro del micro-kernel y se llama a la función del modelo que se va a ejecutar. Esta operación se realiza dentro de una estructura `while`, seguida de la función `elimina`, que sirve para terminar el proceso cuando se indique el final de la operación.

3.8. Sumario

En este capítulo, se presentó como se realizó la configuración del archivo *target* `generador.tlc`, la estructura del código generado y la implementación de la interfaz para poder usar el código generado en otro ambiente, en específico en un micro-kernel de tiempo real.

Utilizando MatLab, el usuario puede diseñar, simular y generar código de un modelo o sistema de control y utilizarlo en una plataforma, ambiente o procesador específico. Para el caso de esta aplicación se utiliza MatLab para producir un código más manejable, simplificado y sin dependencias con la ayuda módulo generador **Bkernel**, el cual lo conforman varios archivos de configuración, que establecen las condiciones de generación de código en base al archivo *target* `generador.tlc`. El proceso de construcción permite suministrar archivos de enlace (*hook files*) opcionales, que se ejecutan en puntos específicos en la generación de código y en el proceso *make* o adicionando acciones específicas en los *targets* en el proceso de construcción.

El modelo del sistema, está formado del generador de código y el micro-kernel de tiempo real. La aplicación contempla la generación del código y la ejecución de este se realiza fuera del ambiente, en el micro-kernel como un proceso más del mismo. El código una vez generado, se maneja como tarea de control, dentro de un sistema de control con características de tiempo real, proporcionadas por el micro-kernel. El micro-kernel puede manejar más de un proceso o tarea, donde la función principal de este, es la de permitir la creación y ejecución concurrente de varios procesos.

El micro-kernel fue desarrollado para funcionar "sobre"MS-DOS y contiene una librería para el uso de un entorno gráfico independiente al Sistema Operativo y los modelos Simulink se ejecutan en el micro-kernel concurrentemente como tareas de control.

En el capítulo siguiente se presentará como está constituída la estructura del

micro-kernel, junto con la activación de los procesos como tareas de control. Así mismo, se presentará las pruebas y simulaciones que se realizaron para verificar el comportamiento de los modelos Simulink como tareas de control dentro del kernel de tiempo real.

Capítulo 4

Ejecución de modelos de control

4.1. Introducción

Como se ha mencionado en los capítulos anteriores, la aplicación propuesta en esta tesis, presenta el desarrollo de una interfaz de programación como una alternativa de simulación, utilizando la combinación de MatLab (ambiente de diseño) y un micro-kernel de tiempo real (ambiente externo). Esto es, utilizando MatLab/Simulink, el usuario puede diseñar y simular sistemas de control y con la ayuda del módulo generador, se construye el código del modelo diseñado, de manera que este pueda ser utilizado en una plataforma, ambiente o procesador específico. Para el caso de esta aplicación se utiliza MatLab/Simulink para producir un código más manejable, simplificado y sin dependencias de MatLab con la ayuda del módulo generador **Bkernel** (cap 2), el cual lo conforman varios archivos (sección 2.6.3) de configuración, que establecen las condiciones de generación de código en base al archivo *target* `generador.tlc` (sección 2.7.2).

El sistema lo constituye el generador de código y el micro-kernel de tiempo real, por lo que la aplicación contempla la construcción del código y la ejecución de este se realiza fuera del ambiente de MatLab, en específico en un micro-kernel. El código una vez generado, se maneja como tarea de control; esto es debido a que lo que pretende es simular un proceso de control como una tarea de tiempo real, teniendo la posibilidad de manejar más de un proceso o tarea de control, dentro del micro-kernel.

Así, el módulo generador junto con el micro-kernel ofrecen una alternativa de simulación para el estudio de sistemas de control con características de tiempo real. Los modelos Simulink se ejecutan en el micro-kernel permitiendo visualizar como se comporta el modelo de control en conjunto con otros procesos de Simulink que se ejecutan concurrentemente.

En el presente capítulo se describe como está constituída la estructura del micro-kernel donde se ejecutan las tareas de control. De igual manera, se describe también la activación de estos procesos. Así mismo, se presentan las pruebas y simulaciones llevadas a cabo, para verificar el comportamiento de los modelos Simulink como tareas de control dentro del micro-kernel. Para las pruebas, se presentan las condiciones de la simulación y los resultados obtenidos de cada uno de los casos de estudio.

4.2. Micro-kernel de Tiempo Real

El micro-kernel es la etapa final de la aplicación, ya que en él se ejecutan las tareas de control generadas para simular su comportamiento como base de prueba para una implementación real. A continuación, se describe las características del micro-kernel (el cual fue desarrollado sobre MS-DOS) que se utiliza en el aplicación de la tesis.

La finalidad del kernel, es la de permitir la creación y ejecución concurrente de varios procesos. Proporcionando funciones ("primitivas"del sistema) que permitan la creación, eliminación, sincronización y comunicación entre procesos.

El sistema está desarrollado en forma de capas. En el diagrama (figura 4.1), se muestra la arquitectura simplificada del Kernel. A continuación se explica de manera breve los componentes de cada una de las capas del sistema.

4.2.1. Arquitectura del micro-kernel

La capa principal, llamada micro-kernel, consta de la definición e inicialización de las variables y estructuras principales del sistema. Es la primera parte que se ejecuta del sistema, después de inicializar las variables y estructuras, reemplaza el control de MS-DOS y pone en funcionamiento los manejadores del sistema.

El micro-kernel se encarga de crear el primer proceso a ejecutar dentro del kernel (figura 4.2); de este se desprenden los demás procesos del sistema. Cuando el sis-

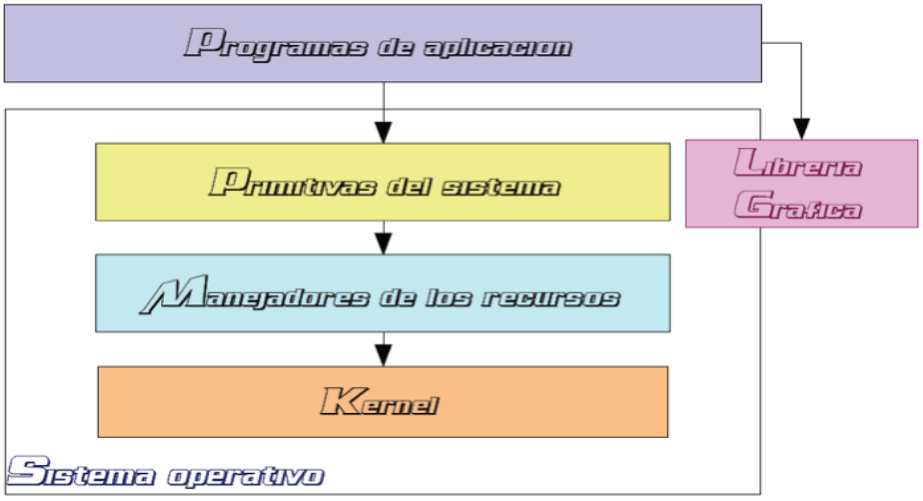

Figura 4.1: Arquitectura del microkernel.

tema debe terminar, el micro-kernel se encarga de regresar el control a MS-DOS y reestablecer el estado original de la pantalla.

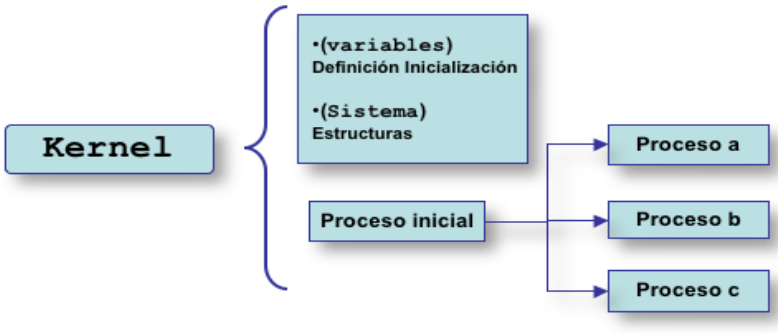

Figura 4.2: Proceso inicial.

4.2.2. Manejadores de los recursos

La segunda capa llamada Manejadores de los recursos, contiene todas las partes del sistema que son agregadas a MS-DOS, para agregar estas funcionalidades se han definido varios submódulos dentro de esta capa. Los módulos se describen a continuación, y su agrupación es en función de los recursos que ellos administran.

- **Manejador de colas de procesos.** Este manejador, define el recurso más auxiliado y utilizado de todo el micro-kernel, las colas de procesos. En ellas se establece la estructura que define a los procesos del micro-kernel. Esta, define los tipos de datos abstractos, colas simples, colas de prioridad y cola delta. Para cada una se tienen las funciones de: crear una cola, crear un nodo, liberar un nodo, agregar un elemento a la cola, quitar un elemento de la cola, verificar si la cola se encuentra vacía y obtener una referencia al nodo almacenado al principio de la cola.[1] Sus definiciones y su implementación se encuentran en los archivos `mancolas.h` y `mancolas.c`.

- **Manejador del CPU.** En este módulo se engloban todas las funciones necesarias para la creación, inicialización, eliminación y planificación de los procesos. Aquí se encuentran los mecanismos para tomar el control y la planificación del procesador. Los procesos tienen 8 niveles de prioridad y son planificados con base a esas prioridades. Sus definiciones y su implementación se encuentran en los archivos `mancpu.h` y `mancpu.c`.

- **Manejador de Semáforos.** En este módulo se encuentran las funciones para la creación, eliminación y uso de semáforos para la sincronización de procesos. El signal (señal) y el wait (espera) requieren de varias funciones que forman parte de los manejadores de colas y del CPU para la suspensión y la rehabilitación de los procesos. Sus definiciones y su implementación se encuentran en los archivos `mansem.h` y `mansem.c`.

- **Manejador del reloj.** En el manejador de reloj, se tienen funciones que permiten a los procesos realizar acciones en función del tiempo. En este manejador, se define al reloj del sistema como un recurso más, el cual se puede administrar para los procesos que requieran esperar por algún suceso temporal. Sus definiciones y su implementación se encuentran en los archivos `manreloj.h` y `manreloj.c`.

- **Manejador de buzones.** En esta sección se tienen funciones y definiciones para la comunicación entre procesos a través de Buzones. La creación y eliminación de buzones, así como el envío y la recepción de mensajes a través del buzón. Sus definiciones y su implementación se encuentran en los archivos `manmsg.h` y `manmsg.c`.

[1] Esto solo disponible para las colas de prioridad.

4.2.3. Primitivas del Sistema

Esta capa es la parte accesible del micro-kernel desde los procesos. Proporciona una interfaz libre de detalles de implementación de las acciones que se le permiten realizar a un proceso. Las funciones reciben solo los argumentos necesarios para la realización de la tarea solicitada y no da pie a configuración ni a opciones de manejo de los recursos disponibles. Las primitivas que el sistema proporciona son:

- **Primitivas de procesos.** Creación (activa) y eliminación (elimina) de procesos.
- **Primitivas de sincronización.** Creación de semáforos (creaSemaforo), wait (espera) y signal (señal).
- **Primitivas de tiempo.** retrasa, la cual permite la suspensión de un proceso durante un lapso de tiempo determinado.
- **Primitivas de comunicación.** Creación de buzones (creaBuzon), envío (envía) y recepción (recibe) de mensajes.

4.2.4. Librería Gráfica

En esta parte del sistema se tienen un conjunto de funciones, casi todas accesibles para el usuario. Al no formar parte del control del procesador y solo manejar el formado de despliegue, esta no es parte del sistema. Se llama librería por que es un conjunto complementario de funciones, que se consideró importante, para evitar el uso de las funciones de despliegue en modo texto proporcionadas por el MS-DOS a través de la interrupción 21H.

La librería está estructurada de manera de que se pueda cambiar el número y tamaño de las ventanas, así como el uso de distintos tipos de fuentes. La velocidad de este entorno gráfico es la mayor posible, dado que accede directamente la memoria de video para realizar cualquier despliegue. Las funciones de mayor utilidad proporcionadas por esta librería para los programas de aplicación son:

- **asignaVentana.** Permite que la librería le proporcione una estructura de datos denominada “ventana”, que le ayudará al proceso manejar su propio sistema de coordenadas y realizar despliegues en una ventana sin interferencias por parte de otro proceso usuario.
- **dibCar.** Permite dibujar un carácter en una posición y color especificado.

- **dibCadena.** Permite dibujar una cadena de caracteres en una posición y color especificado.
- **dibLinea.** Permite dibujar una linea recta con un color especificado.
- **dibCuadro.** Permite dibujar un rectángulo iluminado de un color especificado.
- **dibCirculo.** Permite dibujar un circulo iluminado de un color especificado.

4.3. Activación de los procesos

El proceso de activación de las tareas de control se encuentra en el archvio `run.c`. El micro-kernel cuenta con 6 procesos de prueba que se pueden ejecutar si se activan; estos procesos se nombraron *procesos naturales del kernel* y solo se utilizan cuando se desea ejemplificar la ejecución concurrente de los procesos. Estos son:

1. proceso pelota (activa una pelota que rebota en pantalla),
2. proceso reloj (activa un reloj con la hora del sistema),
3. proceso Editor (activa un editor de texto sencillo),
4. proceso Filósofos (resuleve el problema de los filósofos comensales),
5. proceso Servidor/Cliente (activa un proceso de envío de mensajes)

Como se mencionó en la sección anterior, una función principal de nombre `proceso1`, esta encargada de activar los procesos y las tareas de control. Para el caso de la aplicación en la tesis se agrega la llamada al proceso Simulnik dentro de la estructura de la función `proceso1`, de forma que el usuario pueda activar o desactivar los procesos naturales del kernel y las tareas de control o bien combinarlas.

En el cuadro 4.3, se muestra el fragmento de código para la función `proceso1`, que activa los procesos naturales del kernel y las tareas de contol de Simulink. En el fragmento de código, se ejemplifica como estan activados 5 procesos (*1 proceso natural y 4 procesos Simulink*) y cabe mencionar que aqui también se incluye la función "`elimina()`", que sirve para terminar la ejecución del micro-kernel. En la siguiente sección, se describirá la descripción de los modelos y las pruebas de aplicación.

```
void proceso1(void){
    /* PROCESOS NATURALES DEL KERNEL */
            // activa(proc1, "pelota", 5);
            activa(proc2, "reloj", 5);
            //activa(p_simulink, "modelo", 5);
            //activa(proc3, "Editor", 5);
            //activa(aplFilosofos, "aplicfil", 5);
            //activa(aplServidor, "Servidor", 5);
            //activa(aplCliente, "Cliente", 5);

    /* Procesos Simulink */
            activa(proc5, "p_simulink1", 5);
            activa(proc6, "p_simulink2", 5);
            activa(proc7, "p_simulink3", 5);
            activa(proc8, "p_simulink4", 5);
            // activa(proc9, "p_simulink5", 5);
    elimina();
}
```

Figura 4.3: Función *proceso1*

4.4. Procesos Simulink

Los modelos Simulink desarrollados, se construyeron para probar la aplicación con los bloques más comúnmente usados para desarrollar modelos de control, por ejemplo bloques de constantes, ganancias, sumadores, integradores, funciones, etc.

Con estos modelos Simulink se genera el código de cada uno de ellos y después se ingresa al micro-kernel para su ejecución. En las siguientes subsecciones se muestra cada uno de los modelos generardos, los archivos generados y finalmente la ejecución de estos como tareas de control.

4.4.1. Modelos de Prueba

Para las pruebas en el micro-kernel se contruyen 5 modelos Simulink, los cuales son:

1. Modelo "control1" (muestra la operación de una función de tranferencia).
2. Modelo "sse" (muestra un generador de onda senoidal).
3. Modelo "filtro" (muestra la operación de un filtro discreto).
4. Modelo "filtro2" (muestra la operación con un bloque `zero-pole` y un integrador).
5. Modelo "conv" (convertidor $C°/F°$).

En la aplicación se plantea la generación de código de C a partir del diagrama de control de Simulink y utilizarlo a través de la interfaz de aplicación `gen.c` en el micro-kernel de tiempo real como proceso o tarea control. El modelo de tareas de control se ejecuta como un proceso más del micro-kernel, sin recursos compartidos, compartiendo la posibilidad de ejecutar varios procesos de forma concurrente.

Modelo de prueba I

Este primer modelo es una función de transferencia sencilla, compuesta de tres blouqes Simulink: un bloque "`Step`", un bloque "`Transfer Fcn`" y un bloque "`Scope`". La función que realiza este modelo, es la de obtener la respuesta de la operación *H(s)* por medio de un generador *Step*.

$$H(s) = \frac{1}{s+0{,}005}$$

La figura 4.4, muestra el modelo Simulink y su respuesta con el bloque `Scope`.

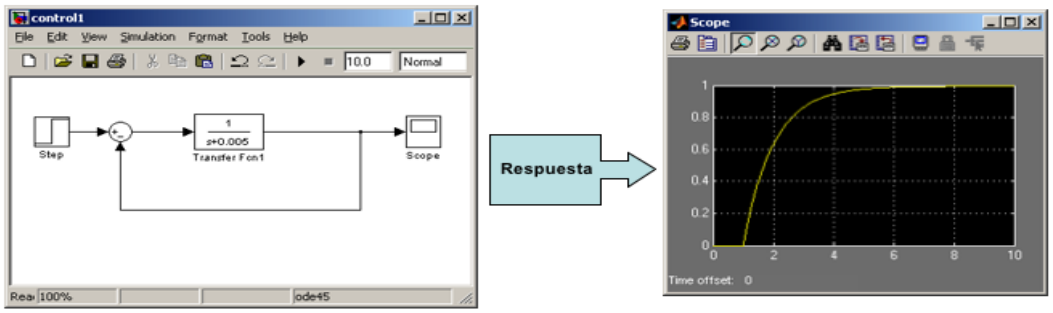

Figura 4.4: Modelo de control1.

Una vez simulado el modelo, se construye el código con el módulo generador. Esta operación se realiza integrando el bloque librería *gen_tar* al modelo, como la muestra

la figura 4.5.

El archivo del modelo Simulink, se llama "control1.mdl" y al activar el módulo generador, se construyen los archivos que describen el comportamiento del modelo Simulink; estos archivos se guardan en un subdirectorio creado al inicio del proceso. Los archivos se muestran en la tabla 4.1 y son los que se agregan a la carpeta de respaldo `condsim` dentro del micro-kernel.

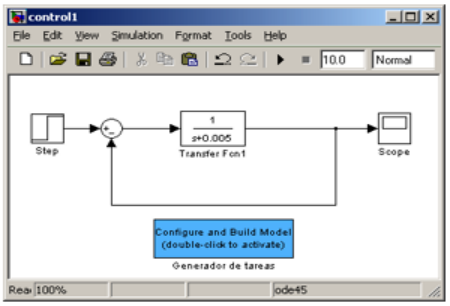

Figura 4.5: Modelo control1.

Tabla 4.1: Archivos del modelo *control1*

Archivos	
`control1.c`	`control1_types.h`
`control1.h`	`control1_prm.h`
`control1_private.h`	`control1_export.h`

Ejecución del modelo "control1"

El siguiente paso en la apliación es integrar el modelo al archivo `gen.c` dentro de la estructura del micro-kernel. Como se mostró en el cuadro 3.8 de la sección 3.7; es importante resaltar que solo se ingresa la cadena del nombre del modelo Simulink. Una vez integrado el código al micro-kernel de tiempo real, se compila toda la aplicación.

La ejecución del modelo, se realiza junto con un proceso natural del micro-kernel. El proceso natural que se ejecuta, es el proceso "`aplicfil`"; por lo que en pantalla,

se despliega la ejecución de dos procesos (el proceso `aplicfil` y el proceso Simulink "`control1`"). En la figura 4.6, se muestra la ejecución de ambos procesos.

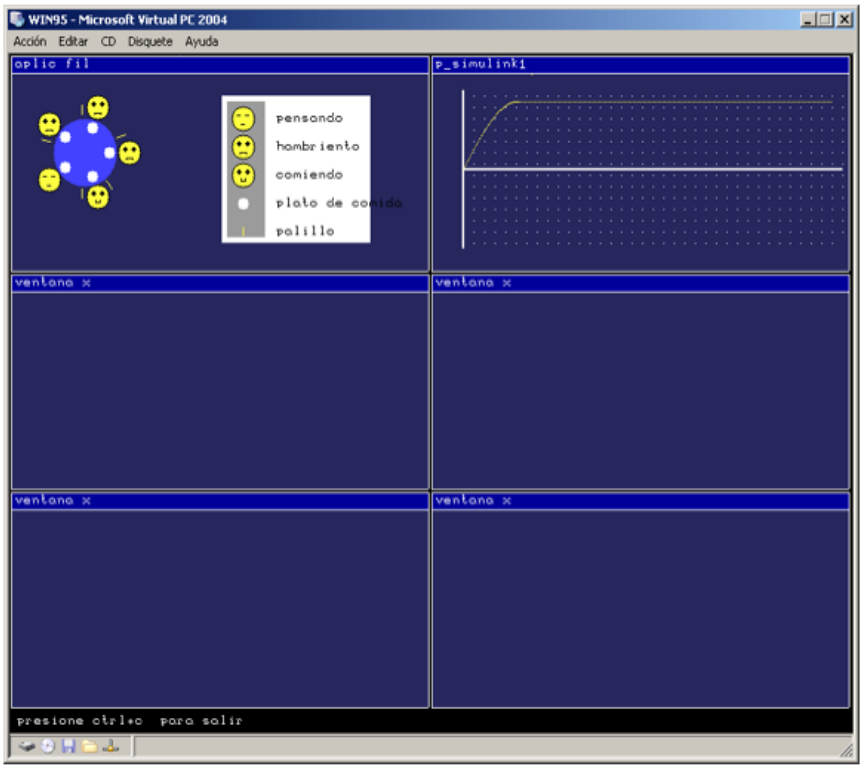

Figura 4.6: Respuesta del micro-kernel.

Modelo de prueba II

Este modelo Simulnik es el más sencillo de todos, debido a que presenta un arreglo de dos bloques Simulink: un bloque "`Sine Wave`" y un bloque "`Scope`". La función que realiza este modelo, es la de desplegar la respuesta del generador de onda senoidal en un intervalo de tiempo infinito. La figura 4.7, muestra el modelo Simulink "`ssen.mdl`" y su respuesta con el bloque `Scope`.

Con el modelo simulado, se construyó el código con el módulo generador. Esta operación se realiza de igual manera, integrando el bloque *gen_tar* al modelo, como la muestra la figura 4.8.

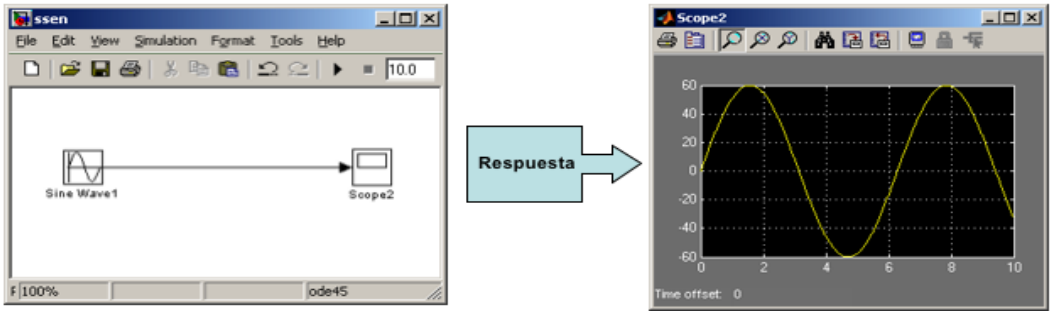

Figura 4.7: Modelo de control `ssen`.

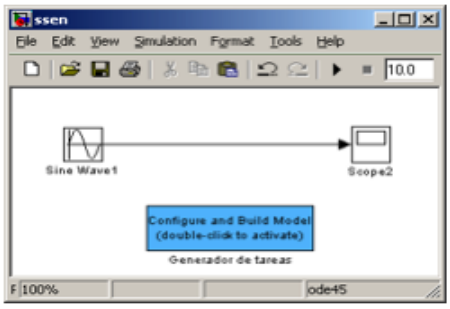

Figura 4.8: Modelo `ssen`.

Los archivos que describen el comportamiento del modelo Simulink se generan al activar el módulo `gen_tar`. Estos archivos se construyen dentro de otro subdirectorio creado al inicio del proceso de nombre `ssen_ert_rtw`. Los archivos se muestran en la tabla 4.2 y se agregan a la carpeta de respaldo `condsim` dentro del micro-kernel.

Tabla 4.2: Archivos del modelo *ssen*

Archivos	
`ssen.c`	`ssen_types.h`
`ssen.h`	`ssen_prm.h`
`ssen_private.h`	`ssen_export.h`

Ejecución del modelo "ssen"

Una vez integrado el nombre del modelo al archvio `gen.c` dentro de la estructura del micro-kernel, se compila la aplicación. La ejecución del modelo, se realiza junto con el modelo anterior (control1); por lo que en pantalla, se despliega la ejecución de 2 tareas de tiempo real. En la figura 4.9, se muestra la ejecución de los procesos p_simulink1 y p_simulink2.

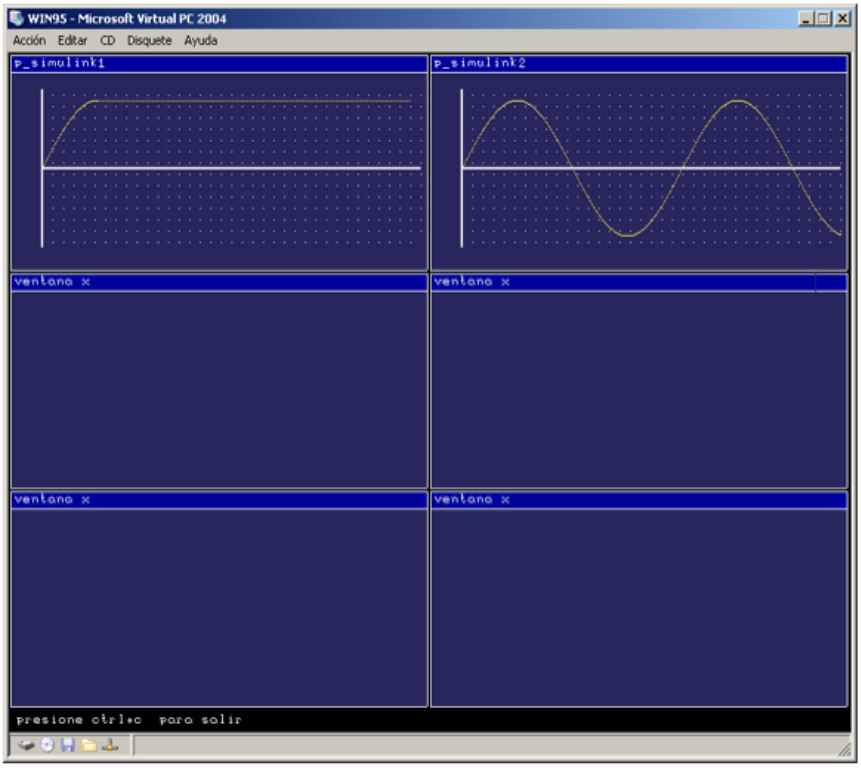

Figura 4.9: Ejecución de las tareas control1 y ssen.

Modelo de prueba III

El modelo Simulink que se presenta a continuación es un filtro discreto compuesto de 4 bloques Simulink: un bloque "`Step`",un bloque "`Sum`", un bloque "`Zero-Pole`" y un bloque "`Scope`". La función que realiza este modelo, es la de obtener la respuesta de la operación:

$$H(s) = \frac{1}{s(s-1)(s-1)}$$

La figura 4.10, muestra el modelo Simulink y su respuesta con el bloque `Scope`.

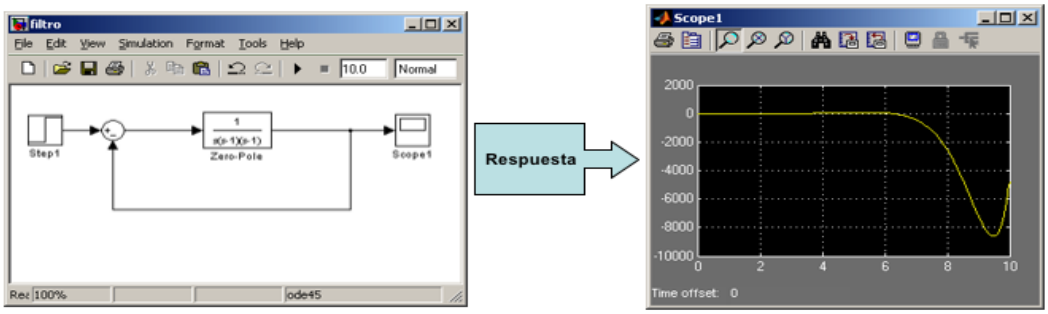

Figura 4.10: Modelo de prueba 3.

El archivo del modelo Simulink, se llama "`filtro.mdl`" y los archivos que describen el comportamiento del modelo Simulink se generan al activar el módulo `gen_tar`; estos archivos se agregan al subdirectorio creado al inicio del proceso de nombre `filtro_ert_rtw`. Los archivos se muestran en la tabla 4.3 y se agregan a la carpeta de respaldo `condsim` dentro del micro-kernel.

Tabla 4.3: Archivos del modelo *filtro*

Archivos	
`filtro.c`	`filtro_types.h`
`filtro.h`	`filtro_prm.h`
`filtro_private.h`	`filtro_export.h`

Ejecución del modelo "filtro"

Una vez integrado el nombre del modelo al archvio `gen.c` dentro de la estructura del micro-kernel, se compila la aplicación. La ejecución del modelo, se realiza junto con los procesos "`Editor`", "`control1`" y "`ssen`".

Por lo que en pantalla, se despliega la ejecución de 2 tareas de tiempo real y un proceso natural del micro-kernel. En la figura 4.11, se muestra la ejecución concurrente de los procesos.

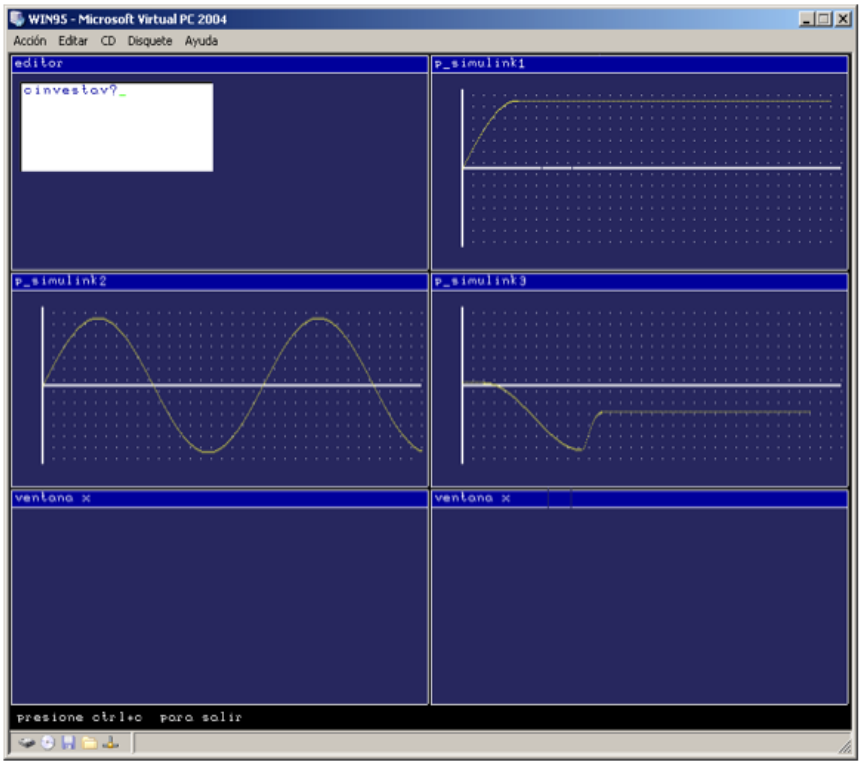

Figura 4.11: Respuesta del micro-kernel para el modelo "filtro".

Modelo de prueba IV

El modelo Simulnik que se presenta a continuación esta compuesto de 5 bloques Simulink: un bloque "`Step`",un bloque "`sum`", un bloque "`Integrator`", un bloque "`Zero-Pole`" y un bloque "`Scope`". La función que realiza este modelo, es la de obtener la respuesta de la operación:

$$H(s) = \frac{1}{s} \cdot \frac{1}{s(s-1)(s-1)}$$

En la figura 4.12, se muestra el modelo Simulink y su respuesta con el bloque `Scope`. El archivo del modelo Simulink, se llama "filtro2.mdl" y los archivos que describen el comportamiento del modelo Simulink se generan al activar el módulo `gen_tar` (figura 4.13). Estos archivos de igual forma, se agregan en un subdirectorio creado al inicio del proceso de nombre `filtro2_ert_rtw`. Los archivos se muestran en la tabla 4.4 y se agregan a la carpeta de respaldo `condsim` dentro de la estructura del micro-kernel.

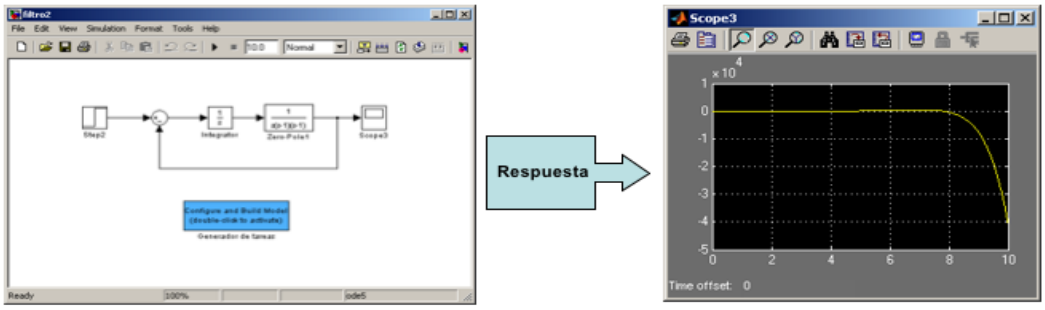

Figura 4.12: Modelo de prueba 4.

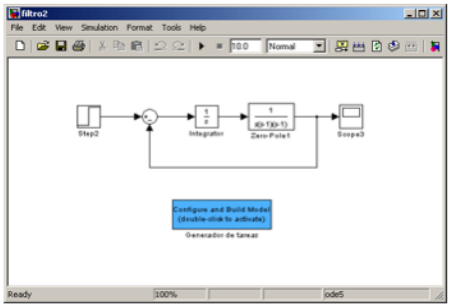

Figura 4.13: Modelo `filtro2` con el módulo Bkernel

Tabla 4.4: Archivos del modelo *filtro*

Archivos	
`filtro.c`	`filtro_types.h`
`filtro.h`	`filtro_prm.h`
`filtro_private.h`	`filtro_export.h`

Ejecución del modelo "filtro2"

Una vez integrado el nombre del modelo al archvio `gen.c` dentro de la estructura del micro-kernel, se compila toda la apliación. La ejecución del modelo, se realiza junto con los procesos anteriores , el proceso "`Editor`" y las tareas de tiempo real "`control1`", "`ssen`" y la tarea "`filtro`". En la ejecución gráfica del micro-kernel, se despliega la ejecución de 4 tareas y un proceso natural del micro-kernel. En la figura 4.14, se muestra la ejecución concurrente de estos procesos.

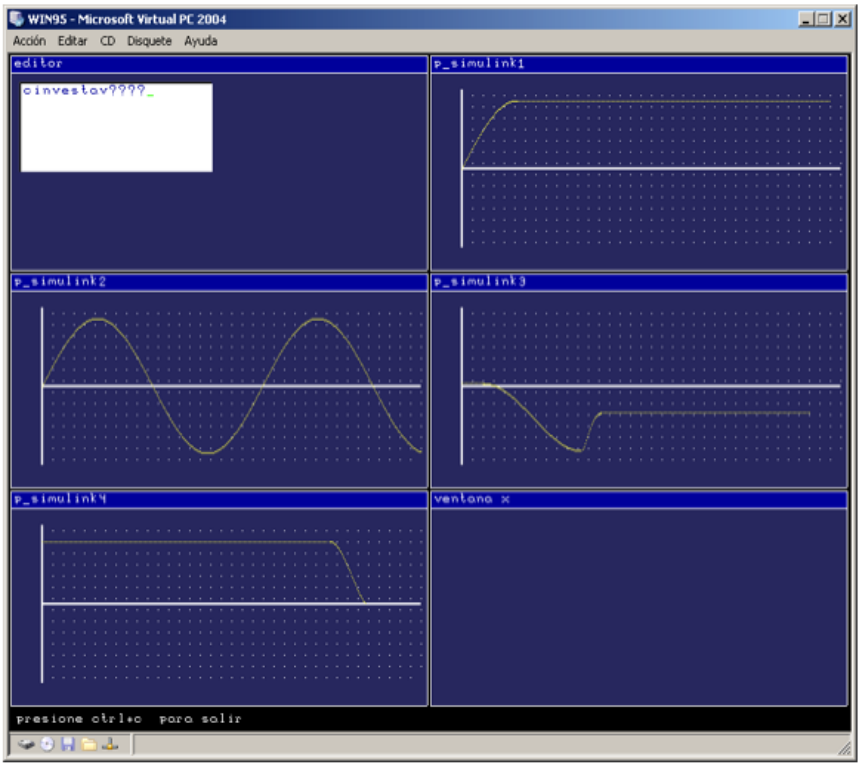

Figura 4.14: Respuesta del micro-kernel para la prueba IV

Modelo de prueba V

El último modelo Simulnik que se presenta es un sistema sencillo para representar gráficamente la conversión de grados °F en grados °C. Este modelo esta compuesto de 5 bloques Simulink: un bloque "`Sine Wave`",un bloque "`Gain`", un bloque "`Sum`", un bloque "`Constant`" y un bloque "`Scope`". La función que realiza este modelo, es la de obtener la respuesta de la operación: $°F = \frac{9}{5}°C + 32$. La figura 4.15, muestra el modelo Simulink y su respuesta con el bloque `Scope`.

El archivo del modelo Simulink, se llama "conv.mdl" y los archivos que describen el comprotamiento del modelo Simulink se construyen al activar el módulo `gen_tar`. Estos archivos se generan en un subdirectorio de nombre `conv_ert_rtw` creado al inicio del proceso de construcción de código. Los archivos se muestran en la tabla 4.5 y se agregan a la carpeta de respaldo `condsim` dentro de la estructura del micro-kernel.

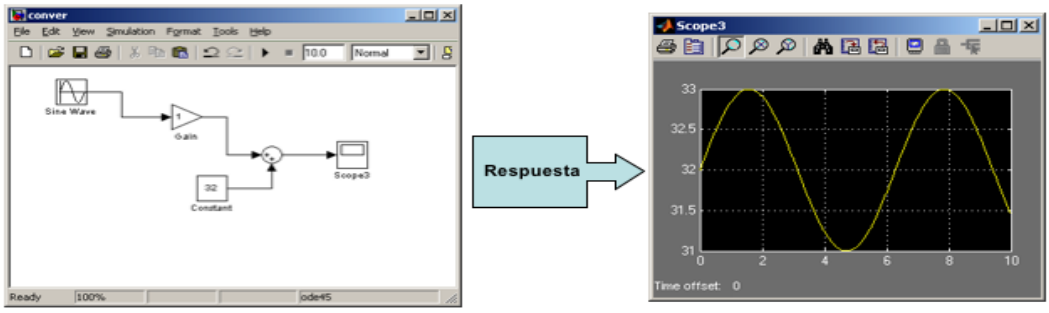

Figura 4.15: Modelo de prueba V.

Tabla 4.5: Archivos del modelo *conv*

Archivos	
`conv.c`	`conv_types.h`
`conv.h`	`conv_prm.h`
`conv_private.h`	`conv_export.h`

Ejecución del modelo "conv"

Una vez integrado el código al archvio `gen.c` dentro de la estructura del micro-kernel, se compila la aplicación. La ejecución del modelo, se realiza junto con 5 procesos, el proceso "`aplicfil`" y las tareas "`control1`", "`ssen`", "`filtro`" y la tarea "`filtro2`".

En la aplicación, se despliega la ejecución de 5 tareas de control y un proceso natural del micro-kernel (para este caso el proceso de los filósofos). En la figura 4.16, se muestra la ejecución concurrente de todos procesos (*aplicfil*, *p_simulink1*, *p_simulink2*, *p_simulink3*, *p_simulink4* y *p_simulink5*).

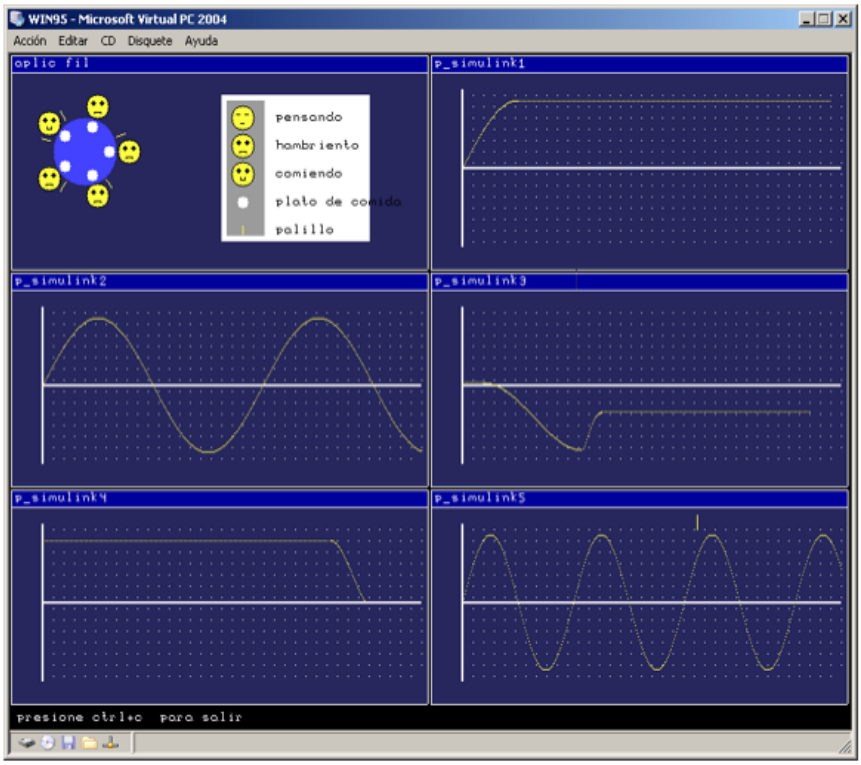

Figura 4.16: Respuesta del micro-kernel(d).

4.5. Sumario

Los modelos Simulink presentados, se generaron con el fin de construir modelos sencillos con los bloques más básicos de Simulink y obtener la descripción en código de bloques empleados. La salida de estos modelos de control fue representada de manera gráfica por un bloque "`Scope`", ya que por medio de este tipo de representación resulta más didáctico observar la evolución de los sistemas que se generen.

El modelo de tareas de control se ejecuta dentro del micro-kernel sin recursos compartidos, teniendo la posibilidad de correr varios procesos de forma concurrente. El método de planificación utilizado en la aplicación es uno de prioridad fija, en particular se emplea la política de RR (Round Robin).

Se utiliza este algoritmo, ya que es uno de los algoritmos más empleados, sencillos y equitativos para compartir recursos entre los procesos. En este caso, cada proceso tiene asignada la misma prioridad con un intervalo de tiempo de ejecución, llamado *cuantum*. Si el proceso agota su *cuantum* de tiempo, se elige a otro proceso para ocupar el recurso o el procesador. Si el proceso se bloquea o termina antes de agotar su *cuantum* también se alterna el uso del recurso o del procesador. La política RR (*Round Robin*) se implementa como planificador de incio del micro-kernel.

El código generado de los modelos Simulink, opera como un proceso más, dentro del micro-kernel; lo que abre la posibilidad de crear más procesos o tareas de control desde los modelos Simulink y probar varios procesos de control para obtener un estudio más detallado del comportamiento de sistemas complejos.

Capítulo 5

Conclusiones y Trabajo a Futuro

5.1. Conclusiones

Este trabajo presenta el desarrollo de un módulo generador de código simple a partir de un diagrama de Simulink y una interfaz de manejo de código. El código generado se utiliza como una aplicación dentro de un micro-kernel de tiempo real por medio de la interfaz, comportándose como un proceso más del mismo; esta herramienta puede ser utilizada para el diseño e implementación de sistemas de control, integrando así la funcionalidad de la herramienta MatLab/Simulink/*Embedded Real Time Worshop* con otra aplicación fuera del ambiente de MatLab (micro-kernel de tiempo real).

El proceso se enfoca principalmente en la generación de código fuente a partir de un diagrama de control generado en Simulink, utilizando el bloque desarrollado que simplifica el proceso de construcción de código. La generación de código se realiza por medio de la modificación de archivos TLC (ít Target Languaje Compiler), utilizados en el proceso de generación.

La aplicación desarrollada es fácil de usar y servirá de base para futuras investigaciones de tipo aplicado, ya que no será necesario implementar en un lenguaje de tiempo real los diseños obtenidos de Simulink (un proceso tedioso y de mucho cuidado sobre todo en el caso de sistemas con subsistemas). En su lugar se generará automáticamente el código listo para una aplicación externa, utilizando el ERT del *Real Time*

WorkShop.

De acuerdo a los resultados obtenidos, cabe resaltar los siguientes puntos:

1. Con el módulo de Simulink desarrollado, se pudo obtener un código más manejable, plano y sencillo, el cual puede utilizarse fuera del ambiente de MatLab con el menor de modificaciones. Este módulo tiene un alto grado de generalización para los esquemas que se contruyan de Simulink, en específico para la librería de control.

2. MatLab es un software de reconocido prestigio en todo el mundo y se ha convertido prácticamente en el estándar para los sistemas de control. Se tienen programas alternativos como "`Wincon`" y "`LabVIEW`", pero muy especializados en su funcionalidad. La selección de MatLab/Simulink ha sido un acierto para el cumplimiento de los objetivos de este proyecto y el trabajo futuro.

3. El código generado de los modelos Simulink, opera como un proceso más, dentro del micro-kernel; lo que abre la posibilidad de crear más procesos o tareas de control desde los modelos Simulink y probar varios procesos de control para obtener un estudio más detallado del comportamineto de sistemas complejos.

4. El código como proceso o tarea de control dentro el kernel, se ejecutó por medio de un mecanismo de planificación de prioridad fija, en particular se empleó la política de RR (Round Robin), ya que es uno de los algoritmos más empleados, y sencillos para compartir recursos entre los procesos. En este caso, cada proceso tiene asignada la misma prioridad de tiempo de ejecución.

5. Aunque la implementación de los ejemplos de prueba se realizó en el entorno Windows, es posible realizar la simulación de tareas de control en otros entornos, incluyendo aquellos en los cuales el diagrama de simulación se encuentre en otra PC(*host*) y el programa de control en otro equipo (*target*), debido a las características de la herramienta.

6. Debido a las características de la herramienta, se puede obtener código para diferentes plataformas (UNIX, MS-DOS, WINDOWS) y manejarlo como una aplicación independiente, de acuerdo a las necesidades del usuario.

7. La aplicación desarrollada es una herramienta en permamente desarrollo y busca incluir en futuras etapas una interfaz de usuario más independiente y con elementos interactivos para el diseño de sistemas de control más completos.

5.2. Trabajo futuro

El diseño de control en tiempo real es difícil y es muy sensible a errores. Con ello el diseño del código de control por lo general desatiende las limitantes de tiempo que hacen difícil la integración de control y aplicaciones de tiempo real en un marco de trabajo integrado.

Por lo que en un trabajo a futuro, se tendría que contemplar la implementación de otras políticas de planificación, como los algoritmos presentados en el capitulo 2. Esto es, para considerar como se comportarian los modelos bajo políticas de manejo de recursos. Es claro que los resultados de las simulaciones realizadas en este trabajo de tesis, entregan información parcial acerca del desempeño, ya que los modelos presentados solo se crearon para obtener la descripción en código de los bloques más utilizados de Simulink. Además, existen otros aspectos de la simulación que no se tomaron en cuenta para el modelado, como los tiempos de simulación dentro Simulink o la consideración de subsistemas dentro de los sistemas de control diseñados. Es necesario mejorar el diseño de la interfaz para tomar en cuenta estos aspectos, así como de contemplar más mecanismos de planificación para el ambiente externo (microkernel).

Debido a las características de la aplicación presentada es posible obtener el código para plataformas diferentes (UNIX, MS-DOS, etc) por lo que en un trabajo a futuro se plantea el manejar el código en un kernel de aplicación como RTAI u otro de base en Linux, para probar los alcances de la herramienta en lo que respecta a la generación de código.

Con lo implementado en la tesis, es conveniente también incluir a la aplicación lo siguiente:

- **Manejo de S-function especiales en modelos dinámicos.** Una de las pricipales características de Simulink es el generar bloques con `S-function` para conectar bloques simulink con elementos externos de hardware, por lo que obtener el código de descripción de estos modelos mejoraría el alcance de la aplicación planteada en lo que respecta al manejo de modelos Simulink.
- **Automatización del manejo de código.** Uno de los principales incovenientes en la aplicación, es que no es tan directo el paso del código generado al microkernel, por lo que para un trabajo a futuro, el agilizar este paso extendería la funcionlidad de la aplicación.

Finalmente en trabajos futuros podría incluirse también ejemplos de aplicación con sistemas reales utilizando otras políticas de planificación, manejo de mensajes entre ellos, manejo de errores y una interfaz gráfica más completa.

Bibliografía

[1] J. Eker, "A matllab toolbox for real-time and control systems co-desing," *6ht International Conference on Real-Time Computing Systems and Applications*, December 1999.

[2] K.-E. Årzen and A. Cervin, "An introduction to control and scheduling co-desing," *39th IEEE Conference on Descision and Control*, December 2000.

[3] R. Saco, E. Pires, and C. Godfrid., "Real time controlled laboratory plant for control," *32nd ASEE/IEEE Frontiers in Educantion Conference*, November 2002.

[4] M. Inc., "Simulink user's guide." Third printing Revised for Version 6.1 (Release 14SP1), Tech. Rep., Octubre 2004.

[5] R. Kirner, R. Lang, P. Puschner, and C. Temple., "Integrating WCET Analysis into a Matlab/Simulink Simulation Model," *16th IFAC Workshop on Distributed Computer Systems*, November 2000.

[6] O. M. Gómez, "Kernel de tiempo real para el control de procesos," Tesis de Maestria, Departamento de Ingeniería Eléctrica – Sección de Computación.,CINVESTAV, Marzo 2004.

[7] A. Cervin, "Integrated control and real-time scheduling," Masters Thesis, Department of Automatic Contorl Lund Institute of Technology, April 2003.

[8] P. M. G. Quaranta, "Using matlab/simulink rtw to build real time control applications in user space with rtia-lxrt," *IEEE Control Systems Magazine*, 2003.

[9] A. Cervin, D. Henriksson, and B. Lincoln., "Jitterbug and true time: Analysis tools for real-time control systems," *2nd Workshop on Real-Time Tools*, August and 2002.

[10] M. Inc., "Real-time windows target user's guide," Third printing Revised for Version 6.1 (Release 14SP1), Tech. Rep., Octubre 2004.

[11] Quanser, "Wincon: Build, start, control," venturcom), Tech. Rep., Octubre 2005.

[12] G. C. Butazzo, "Hard real-time computing systems," *Predictable scheduling algorithms and applications*, 1997.

[13] D. Henriksson and K.-E. Årzen., "On dynamic real-time scheduling of model predictive controllers," *41st IEEE Conference on Decision and Control*, December 2002.

[14] C. Liu and J. Layland, "Scheduling algorithm for multiprogramming in hard real-time enviroment," *Journal of the ACM*, pp. 20:46–61, 1973.

[15] K. Ogata, *Sistemas de control en tiempo discreto*, 2nd ed. Prentice-Hall, 1996.

[16] T. M. Inc.., "Real-time workshop user's guide," Third printing Revised for Version 6.1 (Release 14SP1), Tech. Rep., Octubre 2004.

[17] T. M. Inc., "Tlc for rtw user's guide," Third printing Revised for Version 6.1 (Release 14SP1), Tech. Rep., Octubre 2004.

Apéndice A

A.1. *Real Time Workshop Embedded Coder*

Real Time Workshop Embedded Coder permite generar código C de producción a partir de modelos de tiempo discreto de Simulink. Este código está optimizado para minimizar el uso de memoria RAM y ROM y maximizar el rendimiento de la CPU.

El formato de este código difiere del estándar generado por RTW el cual requiere una mayor complejidad para poder soportar sistemas contínuos. El formato de código embebido soporta sistemas *single-rate* y *multi-rate* así como modos de tarea única y multitarea.

Se puede usar este formato de código para reducir el tamaño del código eliminando código de inicialización el cual puede ser innecesario para una aplicación en particular. El código generado es independiente del target y puede ser ejecutado con o sin sistema operativo.

Además también es posible ajustar el estilo, las estructuras y el tamaño del código generado en función de las necesidades del usuario.

Los targets embebidos que soporta Real-Time Workshop son los siguientes:

- Embedded Target para microcontroladores Infineon C166. Este producto permite ejecutar código en tiempo real de una aplicación para ejecutarse en los microcontroladores Infineon C166.

- Embedded Target para Motorola MPC555. Embedded Target for Motorola MPC555 permite implantar código de producción generado desde el Real-Time Workshop Embedded Coder directamente en el microcontrolador MPC555.
- Embedded Target para TI C6000 DSP. Embedded Target for C6000 DSP Platform permite el desarrollo rápido de software en tiempo real para DSP Texas Instruments (TI) C67x de coma flotante y C62x de coma fija.
- Embedded Target for OSEK/VDX. Este producto permite crear aplicaciones para el sistema operativo de tiempo real OSEK/VDX a partir de los modelos de Simulink.

A.2. Tipos de Archvios objetivo

Los cuadros A.1, A.2 y A.3, listan todos los archivos objetivo o*targets* soportados por la herramienta RTW, junto con los formatos de código, plantillas `makefile` y comandos `make` asociados con estos *targets*.

Tabla A.1: Targets disponibles del *System Target File* de Matlab

Target/formato del código	**STF**	**Plantilla** `Makefile`	**Comando Make**	**Generador de Plataforma**
RTW Embebido T	ert.tlc	ert_default_tmf	make_rtw	PC y Unix
RTW Embebido W	ert.tlc	ert_watc.tmf	make_rtw	Watcom
RTW Embebido VC	ert.tlc	ert_vc.tmf	make_rtw	Visual C/C++
RTW Embebido MSVC	ert.tlc	ert_msvc.tmf	make_rtw	VC/P. Makefile
RTW Embebido BC	ert.tlc	ert_bc.tmf	make_rtw	Borland
RTW Embebido LCC	ert.tlc	ert_lcc.tmf	make_rtw	LCC
RTW Embebido Unix	ert.tlc	ert_unix.tmf	make_rtw	Unix

Tabla A.2: Targets disponibles del *System Target File* de Matlab (Continuación)

Target/formato del código	**STF**	**Plantilla `Makefile`**	**Comando Make**	**Generador de Plataforma**
RTW Genérico T	grt.tlc	grt_default_tmf	make_rtw	PC o Unix
RTW Genérico W	grt.tlc	grt_watcom.tmf	make_rtw	Watcom
RTW Genérico VC	grt.tlc	grt_vc.tmf	make_rtw	Visual C/C++
RTW Genérico MSVC	grt.tlc	grt_msvc.tmf	make_rtw	VC/P. Makefile
RTW Genérico BC	grt.tlc	grt_bc.tmf	make_rtw	Borland
RTW Genérico LCC	grt.tlc	grt_lcc.tmf	make_rtw	LCC
RTW Genérico Unix	grt.tlc	grt_unix.tmf	make_rtw	Unix
RTW Genérico Dinámico T	grt_malloc.tlc	grt_malloc_default_tmf	make_rtw	PC o Unix
RTW Genérico Dinámico W	grt_malloc.tlc	grt_malloc_wat.tmf	make_rtw	Watcom
RTW Genérico Dinámico VC	grt_malloc.tlc	grt_malloc_vc.tmf	make_rtw	Visual C/C++
RTW Genérico Dinámico MSVC	grt_malloc.tlc	grt_malloc_msvc.tmf	make_rtw	VC/C++
RTW Genérico Dinámico BC	grt_malloc.tlc	grt_malloc_bc.tmf	make_rtw	Borland
RTW Genérico Dinámico LCC	grt_malloc.tlc	grt_malloc_lcc.tmf	make_rtw	LCC
RTW Genérico Dinámico Unix	grt_malloc.tlc	grt_malloc_unix.tmf	make_rtw	Unix

Tabla A.3: Targets disponibles del *System Target File* de Matlab (Continuación)

Target/formato del código	**STF**	**Plantilla `Makefile`**	**Comando Make**	**Generador de Plataforma**
ADARTM GNAT	rt_ada_tasking.tlc	gnat_tasking.tmf	make_rtwada	GNAT
S-function T	rtwsfcn.tlc	rtwsfcn_default_tmf	make_rtw	PC o Unix
S-function W	rtwsfcn.tlc	rtwsfcn_watc.tmf	make_rtw	Watcom
S-function VC	rtwsfcn.tlc	rtwsfcn_vc.tmf	make_rtw	Visual C/C++
S-function BC	rtwsfcn.tlc	rtwsfcn_bc.tmf	make_rtw	Borland
S-function LCC	rtwsfcn.tlc	rtwsfcn_lcc.tmf	make_rtw	LCC
Tornado RTW T	tornado.tcl	tornado.tmf	make_rtw	VxWorks
RTWT W	rtwin.tlc	rtwin_watc.tmf	make_rtw	Windows 95/98/NT
RTWT VC	rtwin.tlc	rtwin_vc.tmf	make_rtw	Windows 95/98/NT/Visual C/C++
EVM67 T	evm67.tlc	evm67.tmf	make_rtw	Texas Inst.
CCS Target	ccs.tlc	ccs.tmf	make_rtw	Texas Inst.
XPC Target	xpctarget.tlc	xpctarget.tmf	make_xpc	Watcom y Visual C/C++
DOS (4GW)	drt.tlc	drt_watc.tmf	make_rtw	Watcom C
RTW LE/O	osek_leo.tlc	osek_leo.tmf	make_rtw	Lynx Embedded OSEK
RTSim T	rsim.tlc	rsim_default_tmf	make_rtw	PC y Unix
RTSim W	rsim.tlc	rsim _watc.tmf	make_rtw	Watcom
RTSim VC	rsim.tlc	rsim _vc.tmf	make_rtw	Visual C/C++
RTSim MSVC	rsim.tlc	rsim _msvc.tmf	make_rtw	VC/Project Makefile
RTSim BC	rsim.tlc	rsim _bc.tmf	make_rtw	Borland
RTSim LCC	rsim.tlc	rsim _lcc.tmf	make_rtw	LCC
RTSim Unix	rsim.tlc	rsim _unix.tmf	make_rtw	Unix
ADAST GNAT	rt_ada_sim.tlc	gnat_sim.tmf	make_rtwada	GNAT

Apéndice B

B.1. Configuraciones

A continuación, se describen los pasos para configurar el compilador de Matlab y las etapas que se deben seguir para utilizar el módulo generador y los archivos `*.tlc` para configurar el *target* de la herramienta RTW.

Para realizar esta configuración se creó la carpeta Bkernel que contine todas las plantillas `*.tlc` que se generaron y el módulo generador junto con su script de configuración. El módulo generador, solo es compatible con la versión R13.5 y R14 de Matlab, ya que requiere de la versión 3.1 del *Real-Time workshop Embeded Coder*.

B.1.1. Configuración del compilador

El proceso de construcción de código del RTW depende de la instalación correcta de uno o varios de los compiladores soportados. Cabe señalar que el compilador, en este contexto, se refiere a un ambiente de desarrollo que contiene un *linker* y una utilidad *make*, además de un compilador de lenguaje de alto nivel.

El proceso de construcción también requiere de la selección de una plantilla *makefile*. La plantilla *makefile* determina qué compilador será ejecutado, durante la fase de construcción para compilar el código generado.

Sobre Windows, se debe instalar uno o varios de los compiladores soportados. Además, se debe definir una variable de ambiente asociada con cada compilador.

Se puede seleccionar una plantilla *makefile* que es específica para el compilador utilizado. Por ejemplo, la plantilla `grt_bc.tmf` designa el compilador Borland C/C++ y la palntilla `grt_vc.tmf` designa el compilador Visual C/C++.

Alternativamente, se puede elegir una plantilla *makefile* de inicio que seleccionará el compilador inicial para el sistema que se utilize. El compilador de inicio es el compilador que usa Matlab para construir archivos `mex`. Para establecer el compilador de inicio se utiliza la sentencia:

```
mex -setup
```

Esta opción despliega los compiladores que tiene soportado Matlab, los cuales son:

1. MSDevDir o DEVSTUDIO (definen el path para el compilador de Microsoft Visual C/C++).
2. WATCOM (definen el path para el compilador Watcom C/C++).
3. BORLAND (definen el path para el compilador Borland C/C++)

Para el caso de la aplicación en la tesis, se utiliza el compilador BorlandC 5.2 y la plantilla `ert_bc.tmf`, por lo solo hay que seleccionar la opción del compilador y establecer la ruta. Esta operación configura automaticamente el compilador de la herramienta y lo utiliza para todas las operciones que realize el RTW.

B.1.2. Configuración del módulo generador y de las plantillas `*.tlc`

Dentro de la carpeta Bkernel, se encuetran los archivos del módulo generador; estos son:

- `gen_tar.m`. Script de configuración del módulo generador.
- `generador.mdl`. Libreria de Simulink que contiene el módulo generador.
- `gen_tar.asv` Archvio de Simulnik de vinculación.

De igual manera, dentro de la carpeta se encuetran también los archivos de configuración de la herramienta RTW; estos son:

- `carpeta genrador`. Contiene los archivos `generador.tlc` y `generador.tmf`
- `carpeta hooks`. Contiene los archivos *hook* de confiuración.
- `carpeta plantillas`. Contiene los archivos `banner` y el archivo `generador_file_process`.

Para poder utilizar tanto el moódulo generador como las plantillas `tlc`, la carpeta Bkernel, se debe incluir dentro del path de Matlab sin colocarla dentro de su mismo directorio. Esto es debido a que sí se requiere instalar una nueva versión de Matlab, esta reemplazará todo los archivos dentro del directorio de Matlab anterior. Para incluir en el path de Matlab la carpeta se utiliza la sentencia "`SET PATH`", de forma:

`C:/ SET PATH` *D:/Bkernel*

Finalmente, para verificar que la carpeta de archivos este en el path de Matlab, se utiliza la sentencia "`PATH`". Una vez incluida la carpeta al path de Matlab, los archvos se incluyen automaticamente en las opciones de selección de *target*, de tal forma que al aceder el panel de configuración del RTW se puede seleccionar el target desarrollado, como lo muestra la figura B.1.

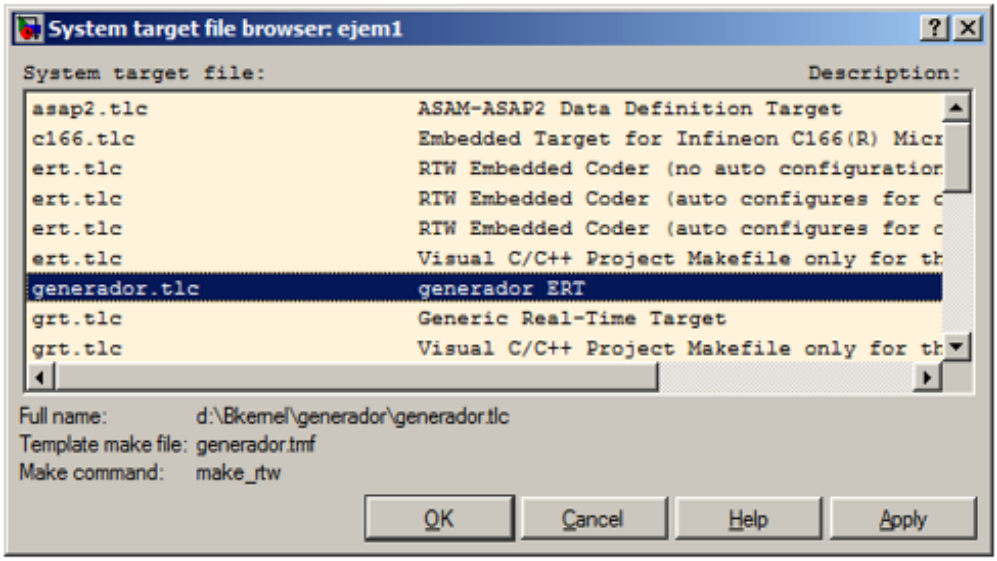

Figura B.1: Selección del archivo *target*

Al seleccionar el target, la herramienta queda totalmente configurada y se puede activar el proceso precionando `Ctrl B` o bien, utilizando el módulo generador.

Para usar el módulo generador, solo se tiene que abrir el archivo `generador.mdl` y arrastrar el bloque dentro del modelo generado. Si el modelo simulink no tiene errores, se puede activar el generador haciendo doble "`click`.[en] el módulo. Una vez accionado el módulo generador, este configura el RTW y genera el código de forma automatica, para que este sea utilizado fuera del ambiente de Matlab.

B.2. Clases de Almacenaniento para archivos objetivos

Los cuadros B.1 y B.2, listan los 4 tipos de clases de almacenamiento o *Storage Class*, y también muestran los tipos de señales que se manejan.

Tabla B.1: Clases de almacenamiento (*Storage class*)

Tipo	Descripción
Auto-If storage class	Este tipo de variables son las más comunes y no son visibles en el código fuente.
Exported global	Estas variables se definen en el código de C generado como variables globales. Las variables de declaración externa se definen en el archivo `model_export.h`. Usando estas variables se puede leer el valor de las señales en el programa generado.

Tabla B.2: Clases de almacenamiento (*Storage class*)

Tipo	Descripción
Imported extern	Éstas son variables globales que son declaradas como variables externas en el código generado. La declaración externa la contiene el archivo `model_common.h`. La definición de la variable debe ser puesta en el código principal.
Imported extern pointer	Este tipo de variables son manejadas como las variables de importación externa del espacio de trabajo de Matlab.

Apéndice C

C.1. Código Generado

A continuación se presenta el código generado para el ejemplo filtro. En particular se muestran los archivos `filtro.c` y `filtro.h`. Estos archivos muestran las funciones que se generan para el modelo construido en simulink.

```
1  /* filtro.c
2   *
3   * Real-Time Workshop code generation for Simulink model "filtro.mdl".
4   *
5   * Model Version              : 1.5
6   */
7  #include "filtro.h"
8  #include "filtro_private.h"
9
10 /* Block signals (auto storage)*/
11 BlockIO_filtro filtro_B;
12
13 /*Continuous states */
14 ContinuousStates_filtro filtro_X;
15
16 /* Solver Matrices */
17
18 /* A and B matrices used by ODE5 fixed-step solver */ static const
19 real_T rt_ODE5_A6 = {
20   1.0/5.0, 3.0/10.0, 4.0/5.0, 8.0/9.0, 1.0, 1.0
21 }; static const real_T rt_ODE5_B66 = {
22   {1.0/5.0, 0.0, 0.0, 0.0, 0.0, 0.0},
23   {3.0/40.0, 9.0/40.0, 0.0, 0.0, 0.0, 0.0},
24   {44.0/45.0, -56.0/15.0, 32.0/9.0, 0.0, 0.0, 0.0},
25   {19372.0/6561.0, -25360.0/2187.0, 64448.0/6561.0, -212.0/729.0, 0.0, 0.0},
26   {9017.0/3168.0,-355.0/33.0,46732.0/5247.0,49.0/176.0,-5103.0/18656.0,0.0},
27   {35.0/384.0, 0.0, 500.0/1113.0, 125.0/192.0, -2187.0/6784.0, 11.0/84.0}
28 };
29
30 continua .....
```

```
static void rt_ertODEUpdateContinuousStates(RTWSolverInfo *si ,
int_T tid) {
  time_T t = rtsiGetT(si);
  time_T tnew = rtsiGetSolverStopTime(si);
  time_T h = rtsiGetStepSize(si);
  real_T *x = rtsiGetContStates(si);
  ODE5_IntgData *id = rtsiGetSolverData(si);
  real_T *y = id->y;
  real_T *f0 = id->f0;
  real_T *f1 = id->f1;
  real_T *f2 = id->f2;
  real_T *f3 = id->f3;
  real_T *f4 = id->f4;
  real_T *f5 = id->f5;
  real_T hB6;
  int_T i;
  int_T nXc = 3;

  rtsiSetSimTimeStep(si,MINOR_TIME_STEP);

  /* Save the state values at time t in y, we'll use x as ynew. */
  (void)memcpy(y, x, nXc*sizeof(real_T));

  /* Assumes that rtsiSetT and ModelOutputs are up-to-date */
  /* f0 = f(t,y) */
  rtsiSetdX(si, f0);
  filtro_derivatives();

  /* f(:,2) = feval(odefile, t + hA(1), y + f*hB(:,1), args(:)(*)); */
  hB0 = h * rt_ODE5_B00;
  for (i = 0; i < nXc; i++) {
    xi = yi + (f0i*hB0);
  }
  rtsiSetT(si, t + h*rt_ODE5_A0);
  rtsiSetdX(si, f1);
  filtro_output(0);
  filtro_derivatives();

  /* f(:,3) = feval(odefile, t + hA(2), y + f*hB(:,2), args(:)(*)); */
  for (i = 0; i <= 1; i++) hBi = h * rt_ODE5_B1i;
  for (i = 0; i < nXc; i++) {
    xi = yi + (f0i*hB0 + f1i*hB1);
  }
  rtsiSetT(si, t + h*rt_ODE5_A1);
  rtsiSetdX(si, f2);
  filtro_output(0);
  filtro_derivatives();

  /* f(:,4) = feval(odefile, t + hA(3), y + f*hB(:,3), args(:)(*)); */
  for (i = 0; i <= 2; i++) hBi = h * rt_ODE5_B2i;
  for (i = 0; i < nXc; i++) {
    xi = yi + (f0i*hB0 + f1i*hB1 + f2i*hB2);
  }
  rtsiSetT(si, t + h*rt_ODE5_A2);
  rtsiSetdX(si, f3);
  filtro_output(0);
  filtro_derivatives();

  /* f(:,5) = feval(odefile, t + hA(4), y + f*hB(:,4), args(:)(*)); */
  for (i = 0; i <= 3; i++) hBi = h * rt_ODE5_B3i;
  for (i = 0; i < nXc; i++) {
    xi = yi + (f0i*hB0 + f1i*hB1 + f2i*hB2 +
      f3i*hB3);
  }
  rtsiSetT(si, t + h*rt_ODE5_A3);
  rtsiSetdX(si, f4);
  filtro_output(0);
  filtro_derivatives();

  /* f(:,6) = feval(odefile, t + hA(5), y + f*hB(:,5), args(:)(*)); */
  for (i = 0; i <= 4; i++) hBi = h * rt_ODE5_B4i;
  for (i = 0; i < nXc; i++) {
    xi = yi + (f0i*hB0 + f1i*hB1 + f2i*hB2 +
      f3i*hB3 + f4i*hB4);
  }
  rtsiSetT(si, tnew);
  rtsiSetdX(si, f5);
  filtro_output(0);
  filtro_derivatives();

continua .....
```

```
/* Model output function */
void filtro_output(int_T tid) {

  /* local block i/o variables */

  real_T rtb_Step1;

  /* Update absolute time of base rate at minor time step */
  if (rtmIsMinorTimeStep(filtro_M)) {
    filtro_M->Timing.t0 = rtsiGetT(&filtro_M->solverInfo);
  }

  if (rtmIsMajorTimeStep(filtro_M)) {
    /* set solver stop time */
    rtsiSetSolverStopTime(&filtro_M->solverInfo,
     ((filtro_M->Timing.clockTick0+1)*filtro_M->Timing.stepSize0));
  }                                    /* end MajorTimeStep */

  /* ZeroPole Block: <Root>/Zero-Pole */
  {
    filtro_B.ZeroPole = 1.0*filtro_X.ZeroPole_CSTATE2;
  }

  /* Step: '<Root>/Step1' */
  {
    real_T currentTime = filtro_M->Timing.t0;
    if (currentTime < 1.0) {
      rtb_Step1 = 0.0;
    } else {
      rtb_Step1 = 1.0;
    }
  }

  /* Sum: '<Root>/Sum1' */
  filtro_B.Sum1 = rtb_Step1 - filtro_B.ZeroPole;
}

/* Model update function */ void filtro_update(int_T tid) {

  if (rtmIsMajorTimeStep(filtro_M)) {
    rt_ertODEUpdateContinuousStates(&filtro_M->solverInfo, 0);
  }

  /* Update absolute time for base rate */

  if(!(++filtro_M->Timing.clockTick0)) ++filtro_M->Timing.clockTickH0;
  filtro_M->Timing.t0 = filtro_M->Timing.clockTick0 *
    filtro_M->Timing.stepSize0 + filtro_M->Timing.clockTickH0 *
    filtro_M->Timing.stepSize0 * 4294967296.0;

  if (rtmIsMajorTimeStep(filtro_M) &&
   filtro_M->Timing.TaskCounters.TID1 == 0) {
    /* Update absolute timer for sample time: 0.2s, 0.0s */

    if(!(++filtro_M->Timing.clockTick1)) ++filtro_M->Timing.clockTickH1;
    filtro_M->Timing.t1 = filtro_M->Timing.clockTick1 *
      filtro_M->Timing.stepSize1 + filtro_M->Timing.clockTickH1 *
      filtro_M->Timing.stepSize1 * 4294967296.0;
  }

}

/* Derivatives for root system: '<Root>' */ void
filtro_derivatives(void) {
  /* simstruct variables */
  StateDerivatives_filtro *filtro_Xdot = (StateDerivatives_filtro*)
    filtro_M->ModelData.derivs;

  /* ZeroPole Block: <Root>/Zero-Pole */
  {

    filtro_Xdot->ZeroPole_CSTATE0 = 1.0*filtro_B.Sum1;
    filtro_Xdot->ZeroPole_CSTATE0 += (1.0)*filtro_X.ZeroPole_CSTATE0;

    filtro_Xdot->ZeroPole_CSTATE1 = (1.0)*filtro_X.ZeroPole_CSTATE0 +
      (1.0)*filtro_X.ZeroPole_CSTATE1;

    filtro_Xdot->ZeroPole_CSTATE2 = (1.0)*filtro_X.ZeroPole_CSTATE1;
  }
}

continua .....
```

```
/* Model initialize function */
void filtro_initialize(boolean_T
firstTime) {

  if (firstTime) {
    /* registration code */
    /* initialize real-time model */
    (void)memset((char_T *)filtro_M, 0, sizeof(rtModel_filtro));

    {
      /* Setup solver object */

      rtsiSetSimTimeStepPtr(&filtro_M->solverInfo,
       &filtro_M->Timing.simTimeStep);
      rtsiSetTPtr(&filtro_M->solverInfo, &rtmGetTPtr(filtro_M));
      rtsiSetStepSizePtr(&filtro_M->solverInfo, &filtro_M->Timing.stepSize0);
      rtsiSetdXPtr(&filtro_M->solverInfo, &filtro_M->ModelData.derivs);
      rtsiSetContStatesPtr(&filtro_M->solverInfo,
       &filtro_M->ModelData.contStates);
      rtsiSetNumContStatesPtr(&filtro_M->solverInfo,
       &filtro_M->Sizes.numContStates);
      rtsiSetErrorStatusPtr(&filtro_M->solverInfo, &rtmGetErrorStatus(filtro_M));

      rtsiSetRTModelPtr(&filtro_M->solverInfo, filtro_M);
    }
    rtsiSetSimTimeStep(&filtro_M->solverInfo, MAJOR_TIME_STEP);
    filtro_M->ModelData.intgData.y = filtro_M->ModelData.odeY;
    filtro_M->ModelData.intgData.f0 = filtro_M->ModelData.odeF0;
    filtro_M->ModelData.intgData.f1 = filtro_M->ModelData.odeF1;
    filtro_M->ModelData.intgData.f2 = filtro_M->ModelData.odeF2;
    filtro_M->ModelData.intgData.f3 = filtro_M->ModelData.odeF3;
    filtro_M->ModelData.intgData.f4 = filtro_M->ModelData.odeF4;
    filtro_M->ModelData.intgData.f5 = filtro_M->ModelData.odeF5;
    filtro_M->ModelData.contStates = ((real_T *) &filtro_X);
    rtsiSetSolverData(&filtro_M->solverInfo, (void
      *)&filtro_M->ModelData.intgData);
    rtsiSetSolverName(&filtro_M->solverInfo,"ode5");

    rtmSetTFinal(filtro_M, 10.0);
    filtro_M->Timing.stepSize0 = 0.2;
    filtro_M->Timing.stepSize1 = 0.2;

      /* block I/O */
      void *b = (void *) &filtro_B;
      filtro_M->ModelData.blockIO = (b);

      {

        int_T i;
        b =&filtro_B.ZeroPole;
        for (i = 0; i < 2; i++) {
          ((real_T*)b)i = 0.0;
        }
      }
    }
    /* states */
    {
      int_T i;
      real_T *x = (real_T *) &filtro_X;
      filtro_M->ModelData.contStates = (x);
      for(i = 0; i < (int_T)(sizeof(ContinuousStates_filtro)/sizeof(real_T));
       i++)
      {
        xi = 0.0;
      }
    }

/* Model terminate function */ void filtro_terminate(void) { }

/* Fin de archivo filtro.c*/
```

```
/*
 * filtro.h
 *
 * Real-Time Workshop code generation for Simulink model "filtro.mdl".
 */

#ifndef _RTW_HEADER_filtro_h_
#define _RTW_HEADER_filtro_h_

#include <limits.h>
#include <math.h>
#include <float.h>
#include<string.h>
#include "rtwtypes.h"
#include "simstruc.h"
#include"fixedpoint.h"
#include "rt_logging.h"

#include "rt_nonfinite.h"
#include "rtlibsrc.h"

#include "filtro_types.h"

/* Macros for accessing real-time model data structure  */

#ifndef rtmGetBlkStateChangeFlag
#define
rtmGetBlkStateChangeFlag(rtm)((rtm)->ModelData.blkStateChange)
#endif

#ifndef rtmGetBlockIO
#define
rtmGetBlockIO(rtm)((rtm)->ModelData.blockIO)
#endif

#ifndef rtmSetBlockIO
#define
 rtmSetBlockIO(rtm, val)((rtm)->ModelData.blockIO = (val))
#endif

#ifndef rtmGetChecksums #define
rtmGetChecksums(rtm)((rtm)->Sizes.checksums)
#endif

#ifndef rtmGetConstBlockIO #define
rtmGetConstBlockIO(rtm)((rtm)->ModelData.constBlockIO)
#endif

#ifndef rtmSetConstBlockIO
#define
rtmSetConstBlockIO(rtm,val)((rtm)->ModelData.constBlockIO = (val))
#endif

#ifndef rtmGetContStates
# define
rtmGetContStates(rtm)((rtm)->ModelData.contStates)
#endif

#ifndef rtmSetContStates
# define
rtmSetContStates(rtm,val)((rtm)->ModelData.contStates = (val))
#endif

/* Definition for use in the target main file */

#define filtro_rtModel                  rtModel_filtro

/* Block signals (auto storage) */
typedef struct _BlockIO_filtro
{
  real_T ZeroPole;                      /* '<Root>/Zero-Pole' */
  real_T Sum1;                          /* '<Root>/Sum1' */
} BlockIO_filtro;

/* Block states (auto storage) for system: '<Root>' */

typedef struct D_Work_filtro_tag {
  struct {
    void *LoggedData;
  } Scope1_PWORK;                       /* '<Root>/Scope1' */
} D_Work_filtro;

continua .....
```

```
/* Continuous states (auto storage) */

typedef struct _ContinuousStates_filtro {
  real_T ZeroPole_CSTATE3;            /* '<Root>/Zero-Pole' */
} ContinuousStates_filtro;

/* State derivatives (auto storage) */

typedef struct _StateDerivatives_filtro {
  real_T ZeroPole_CSTATE3;            /* '<Root>/Zero-Pole' */
} StateDerivatives_filtro;

  /*
   * ModelData:
   * The following substructure contains information regarding
   * the data used in the model.
   */
  struct {
    void *blockIO;
    const void *constBlockIO;
    real_T *defaultParam;
    ZCSigState *prevZCSigState;
    real_T *contStates;
    real_T *discStates;
    real_T *derivs;
    real_T *nonsampledZCs;
    void *inputs;
    void *outputs;
    real_T odeY3;
    real_T odeF63;
    ODE5_IntgData intgData;
  } ModelData;

  /*
   * Sizes:
   * The following substructure contains sizes information
   * for many of the model attributes such as inputs, outputs,
   * dwork, sample times, etc.
   */
  struct {
    uint32_T checksums4;
    uint32_T options;
    int_T numContStates;
    int_T numU;
    int_T numY;
    int_T numSampTimes;
    int_T numBlocks;
    int_T numBlockIO;
    int_T numBlockPrms;
    int_T numIports;
    int_T numOports;
    int_T numNonSampZCs;
    int_T sysDirFeedThru;
    int_T rtwGenSfcn;
  } Sizes;

  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    time_T stepSize;
    uint32_T clockTick0;
    uint32_T clockTickH0;
    time_T stepSize0;
    uint32_T clockTick1;
    uint32_T clockTickH1;
    time_T stepSize1;
    int_T *sampleTimeTaskIDPtr;
    int_T *sampleHits;
    int_T *perTaskSampleHits;
    time_T *t;
    time_T sampleTimesArray2;
    time_T offsetTimesArray2;
    int_T sampleTimeTaskIDArray2;
    int_T sampleHitArray2;
    int_T perTaskSampleHitsArray4;
    time_T tArray2;
  } Timing;

#endif
/* Fin de archivo filtro.h*/
```

Glosario de Términos

ADA - Lenguaje de programación estructurado y fuertemente tipado de forma estática que fue diseñado por Jean Ichbiah de CII Honeywell Bull por encargo del Departamento de Defensa de los Estados Unidos (DoD).

ANSI - El Instituto Nacional Estadounidense de Estándares (ANSI, por sus siglas en inglés: American National Standards Institute) es la principal organización encargada de promover el desarrollo de estándares tecnológicos en Estados Unidos. ANSI es miembro de la Organización Internacional para la Estandarización (ISO) y de la Comisión Electrotécnica Internacional (International Electrotechnical Commission, IEC).

API - (acrónimo inglés de Application Programming Interface, Interfaz de Programación de Aplicaciones) es un conjunto de especificaciones de comunicación entre componentes software. Representa un método para conseguir abstracción en la programación, generalmente (aunque no necesariamente) entre los niveles o capas inferiores y los superiores del software.

ASCII - El código ASCII (acrónimo inglés de American Standard Code for Information Interchange -Código Estadounidense Estándar para el Intercambio de Información), es un código de caracteres basado en el alfabeto latino tal como se usa en inglés moderno y en otras lenguas occidentales.

BTF - (acrónimo inglés de Block Target File), especifica cuál archivo objetivo se debe usar para la generación de código.

CCL - (acrónimo inglés de Custom Code Library), librería de código personalizable que permite insertar código personalizado dentro del código fuente generado.

CPU - (acrónimo inglés de Central Processing Unit) o Unidad Central de Proceso (UCP) a la unidad donde se ejecutan las instrucciones de los programas y se controla el funcionamiento de los distintos componentes de una computadora.

DS - (acrónimo inglés de Data Segment), se refiere a una particular sección de un segmento de código.

DLL - Del acrónimo de Dynamic Linking Library (Bibliotecas de Enlace Dinámico), término con el que se refiere a los archivos con código ejecutable que se cargan bajo demanda del programa por parte del sistema operativo.

DSP - (acrónimo inglés de Digital Signal Processing), Procesamiento Digital de Señales (PDS) es una área de la ingeniería que se dedica al análisis y procesamiento de señales (audio, voz, imágenes, video) que son discretas en el tiempo.

EDF - (acrónimo inglés de *Earliest Deadline First*), es un planificador dinámico que selecciona tareas de acuerdo a sus plazos de respuesta absolutos. Las tareas con plazos más cercanos tendrán mayor prioridad.

ERT - (acrónimo inglés de *Embebed Real Time Target*). Archivo objetivo de instrucciones para sistemas empotrados o embebidos.

GRT - (acrónimo inglés de *Generic Real Time*). Archivo objetivo de instrucciones para sistemas genéricos.

GRTT - (acrónimo inglés de *Generic Real Time Target*). Archivo objetivo o *target* de tiempo real genérico, genera c-digo para ajuste interactivo de parámetros de los modelos, despliege y registro de los resultados en tiempo real de la simulaci-n y asigna datos estáticamente (para la ejecuci-n en tiempo real eficiente).

GRTTM - (acrónimo inglés de *Generic Real Time Target Malloc*). Archivo objetivo o *target* de tiempo real genérico de asignación, utiliza la asignaci-n de memoria dinámica en el c-digo generado, permitiéndole incluir múltiples instancias de un modelo o múltiples modelos.

IP - (acrónimo inglés de *Instrucction Pointer*). Estructura que direcciona a la estructura de una instrucción.

MS-DOS - MS-DOS es un sistema operativo de Microsoft perteneciente a la familia DOS. Fue un sistema operativo para el IBM PC que alcanzó gran difusión. MS-DOS significa Micro-Soft Disk Operating System.

PCB - (acrónimo inglés de *Process Control Block*). En esta estructura de datos se guarda, el estado del proceso, la dirección y tamaño asignado de su código, datos y stack, los recursos que tiene asignados, y posibles manejadores de excepción asignados al proceso.

Quantum - Cantidad máxima de tiempo de CPU, se le asigna a cada proceso para que cumpla con su ejecución.

RR- (acrónimo inglés de *Round Robin*). Este es uno de los algoritmos más sencillos y equitativos en el reparto de recursos entre los procesos, muy válido para entornos de tiempo compartido.

RM - (acrónimo inglés de *Rate Monotonic*). Algoritmo de planificación que se basa en una simple regla que asigna las prioridades de las tareas de acuerdo a su frecuencia. RSIM - (acrónimo inglés de *Rapid Simulation Target*). Archivo objetivo de instrucciones para sistemas de simulación rápida.

RS232 - RS-232 (también conocido como EIA RS-232C) es una interfaz que designa una norma para el intercambio serie de datos binarios entre un DTE (Equipo terminal de datos) y un DCE (Equipo de terminación del circuito de datos), aunque existen otras situaciones en las que también se utiliza la interfaz RS-232.

RTOS - (acrónimo inglés de *Real Timer Operating System*). Designación de sistema operativo de tiempo real.

RTW - (acrónimo inglés de *Real Time Workshop*). Herramienta de MatLab/Simulink para apliaciones de tiempo real.

RTWT - (acrónimo inglés de *Real Time Windows Target*). Herramienta de MatLab/Simulink para apliaciones de tiempo real sobre la plataforma Windows.

Sobretiro - (acrónimo inglés de ít overshoot) Cuando una variable tiende a un valor estable superior existe una velocidad de variación crítica por encima de la cual la variable irá más allá del valores normales. Este pico más allá de valor de consigna se llama sobretiro.

Solver - Los *solvers* o ejecutores son funciones de MatLab que ejecutan algoritmos que se utilizan para resolver sistemas de control.

Stack - Es una zona de memoria (no diferente del resto de memoria) requerida por todo programa (la misma cpu lo requiere) para un uso especial. Su función, sintéticamente, es la de servir para el intercambio dinámico de datos durante la ejecución de un programa, principalmente para la reserva y liberación de variables locales y paso de argumentos entre funciones.

SS - (acrónimo inglés de *Stack Segment*). Caracteriza la sección de un segmento de código, en la pila de almacenamiento de información.

STF - (acrónimo inglés de *System Target File*). Archivo que permite seleccionar el *target*. Estos archivos corresponden a los bloques de Simulink. Por cada bloque en el modelo de Simulink hay un archivo de este tipo que especifica como traducir dicho bloque en el código de un *target* específico.

SFT - (acrónimo inglés de *S-Function Target*). Archivo objetivo o *target* para Funciones S, convierte los modelos en DLL's de Simulink, permitiéndole compartir modelos para simulaciones sin el compromiso de la propiedad intelectual.

TCP/IP - familia de protocolos de Internet. Son un conjunto de protocolos de red que implementa la pila de protocolos en la que se basa Internet y que permiten la transmisión de datos entre redes de computadoras. En ocasiones se la denomina conjunto de protocolos TCP/IP, en referencia a los dos protocolos más importantes que la componen: Protocolo de Control de Transmisión (TCP) y Protocolo de Internet (IP), que fueron los dos primeros en definirse, y que son los más utilizados de la familia.

TLC - (acrónimo inglés de *Target Language Compiler*). El Compilador de lenguaje objetivo construye la estructura de datos para los bloques Simulink.

TT - (acrónimo inglés de *Tornado Target*). archivo objetivo o *target* de Tornado, genera c-digo para la ejecuci-n en **VxWorks**.

VHLL - (acrónimo inglés de *Very High Language Level*), Lenguaje de alto nivel, para la descripción de forma grafica de instrucciones.