3 Funcionamiento desde Real Time Workshop Embedded Coder.

3.1 Introducción.

Existen paquetes de software que constituyen una herramienta de traducción de algoritmos plasmados mediante lenguajes gráficos de especificación formal al lenguaje *C++*, permitiendo, además, la especificación de restricciones de tiempo en el diseño de dichos métodos. Los lenguajes de descripción formal basados en técnicas de prototipado rápido utilizan elementos gráficos, como pueden ser los diagramas de bolas o de bloques, los cuales se encuentran muy extendidos en ámbitos como el control e integración de sistemas.

Gracias a la capacidad que nos ofrecen estas herramientas, es posible automatizar la generación de código máquina partiendo de lenguajes de especificación a muy alto nivel. Para ello se ha de cubrir ciertos pasos, entre los cuales existen interfaces de adaptación, haciéndose necesaria la configuración de cada uno de ellos para los niveles concretos que comunica.

Para el caso que nos ocupa, se eligió el traductor de lenguajes gráficos de diseño [B18] *Real Time Workshop*[1], el cual construye el código *C++* que se compilará mediante *MetroWerks CodeWarrior*, permitiendo la generación automática de código ejecutable para el microcontrolador de *ARM*.

El objetivo de este capítulo consiste en el estudio de la forma en la que se ha de especificar, bajo *RTW*, la interfaz de comunicaciones con el compilador. Para ello se plantean varios apartados, cuya temática abarca desde la descripción de esta herramienta de trabajo hasta la generación de un ejemplo sencillo que pretende ilustrar todo el proceso.

3.2 Descripción de Real Time Workshop.

Se trata, como *Matlab*, de un producto comercial de la empresa *MathWorks, Inc*, la cual se dedica a elaborar herramientas de trabajo asistido por ordenador para multitud de campos asociados al diseño, como son las ramas de la ingeniería, desde el ámbito de la ingeniería financiera hasta el diseño de estrategias de control, pasando por el tratamiento digital de imágenes. Cada una de estas herramientas se agrupa en lo que *MathWorks, Inc* denomina *Toolbox's*, es decir, en *Cajas de Herramientas* que, junto con el núcleo de la aplicación, forman el software *Matlab*.

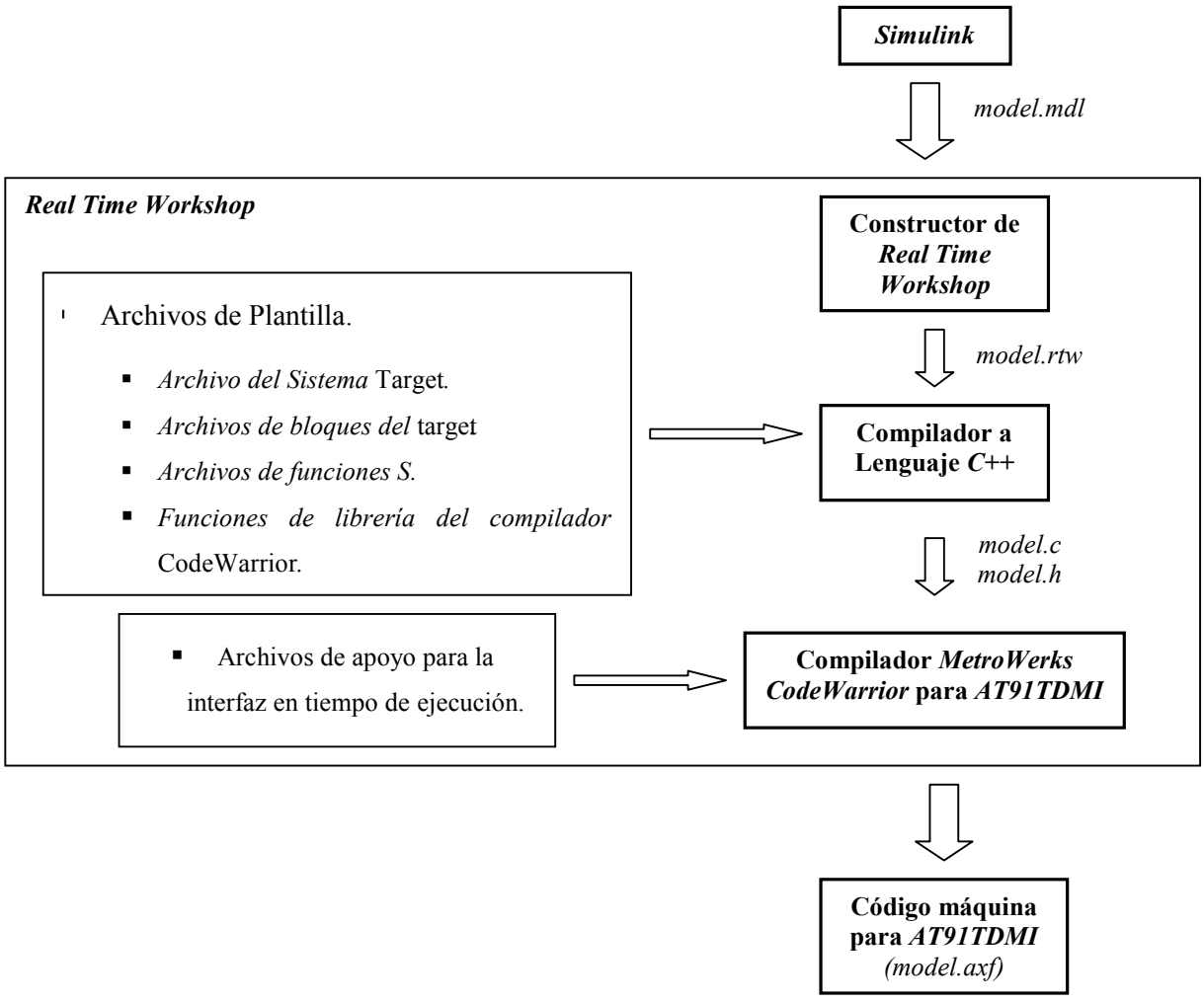
[1] Por simplicidad nos referiremos a *Real Time Workshop* como *RTW*

Dentro del paquete que éste constituye se encuentra la herramienta *Simulink*; ésta conforma un paquete de modelado, simulación y análisis de sistemas dinámicos. Posee una interfaz *GUI*, es decir, *Graphics User Interface*; con lo que admite el uso de diagramas de bloques, sin más que conectar bloques predefinidos o de usuario, para realizar la descripción de sistemas dinámicos.

RTW construye aplicaciones, o sea, código fuente compatible con el lenguaje *C++* a partir de los diagramas descritos por *Simulink* para la edición, seguimiento y depuración de algoritmos sobre multitud de plataformas de computación, yendo desde sistemas con muchos recursos hasta entornos embebidos o *embedded* con especificaciones de tiempo, es decir, permite la computación del tiempo real.

3.2.1 Descripción de la Funcionalidad e Interrelación: Capas.

En este punto se busca la descripción de cada una de las fases [B20] que intervienen en la generación de código máquina. Con este fin, se hará uso del siguiente esquema de bloques:



Partiendo del entorno *Simulink* se genera un diagrama de bloques que se almacena en un archivo con la extensión *mdl*, en él se guarda toda la información relativa al modelo del sistema dinámico de trabajo. Gracias al constructor *RTW* se logra una imagen ejecutable para el sistema *hardware* que soportará la aplicación; para ello se requiere de una serie de pasos. El propio constructor genera, a partir del archivo *model.mdl*, su correspondiente fichero con la extensión *rtw*; éste se utiliza por el compilador interno de *Matlab*, que elabora un conjunto de archivos compatibles con la normativa *ANSI C++*, es decir, que se compone de archivos de código fuente *c* y de librerías *h*.

Gracias a los archivos de configuración de la interfaz [B12] que comunica el generador de código *C++* de *Matlab* con el compilador y depurador concretos, se permite que se les invoque con los parámetros adecuados de forma automática. Estos ficheros de configuración se denominan, por la nomenclatura de *MathWorks Inc*, plantillas o *templates*.

Debido a que *Simulink* permite la descripción funcional de los bloques constructivos de sus diagramas mediante la inclusión de código descrito en lenguaje *C++*, también se hace necesario la especificación de la ruta de directorio de los archivos fuente *c*, a parte de la definición del encapsulado de las funciones que en ellos se definen. Las funciones, utilizadas de este modo, son denominadas funciones *S* [B10], puesto que se construyen en los diagramas de *Simulink*.

3.2.2 Arquitectura de Programa.

[B20] El código *C++* se genera en dos estilos, según el objetivo esté embebido o no; su estructura se ve afectada por si se permite ejecutar un entorno multitarea o por qué sistemas o módulos deben ser incorporados. Las secciones siguientes describen estas distinciones estructurales.

3.2.2.1 Introducción.

Uno de los estilos del código está pensado para el prototipado rápido, así como para la simulación a través de la generación de código. En cambio, el otro estilo está dirigido a aplicaciones imbuidas. Esta tabla clasifica qué objetivos son compatibles con *RTW*. Para mayor información relativa entre el estilo del código y las características del objetivo, véase el apartado *3.2.4.1 Elegir un Formato de Código para la Aplicación*.

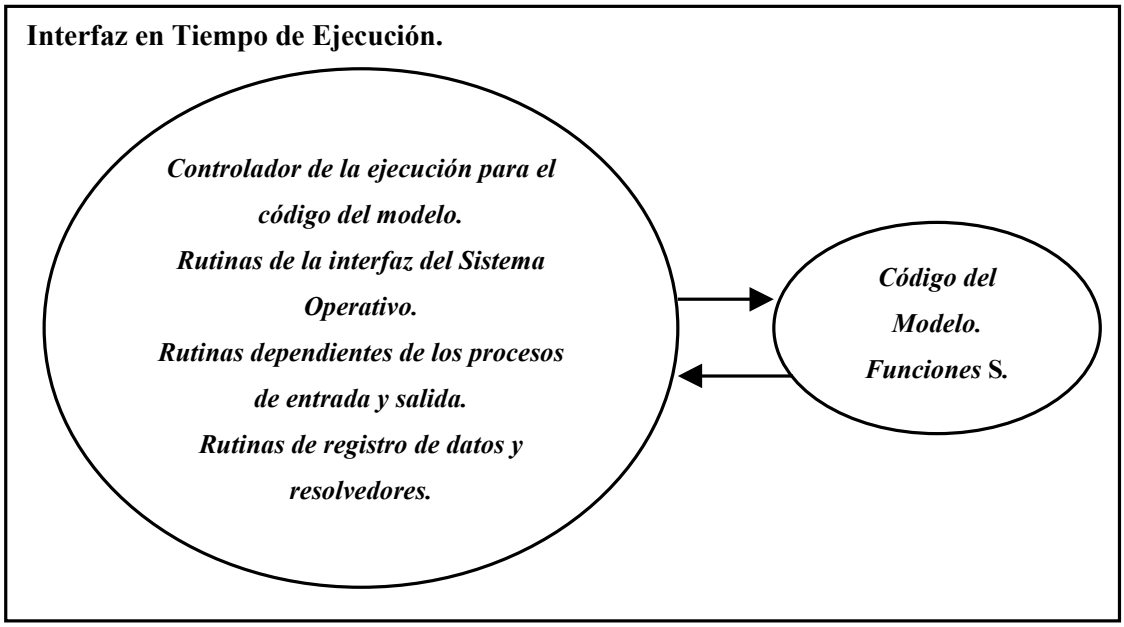
Sistema Objetivo	Estilo del Código
RTW Embedded Coder	*Embebido*: útil como punto de comienzo cuando se va a usar el código generado en una aplicación imbuida.
Tiempo Real Genérico (GRT)	*Prototipado rápido*: simulación sin tiempo real en una estación de trabajo. Resulta útil cuando como inicio del desarrollo de un prototipo en un objetivo en tiempo real, pero con primitivas de un sistema operativo sin tiempo real; además permite validar el correcto funcionamiento del código generado.
GRT dinámico	*Prototipado rápido*: es análogo al caso anterior, salvo que la asignación de memoria se lleva a cabo de forma dinámica.
Función S	*Prototipado rápido*: genera una función *S* en un archivo *C-MEX* facilitando la simulación del modelo de trabajo dentro de otro modelo de *Simulink*.
VxWorks Tornado	*Prototipado rápido*: ejecuta un modelo en tiempo real utilizando las primitivas de gestión de tareas del sistema operativo en tiempo real de nombre *VxWorks*.
Windows en tiempo real	*Prototipado rápido*: ejecuta modelos en tiempo real mediante interrupción cuando se utiliza en sistema operativo *Microsoft Windows*.

Existen otros modos de funcionamiento, pero éstos son los que hemos creído de mayor interés siendo, por tanto, los que se usarán en la generación de código con *RTW*.

Hay un conjunto de fabricantes, independientes de *MathWorks Inc*, que facilitan otros sistemas objetivos utilizando las interfaces de *Matlab*; entre ellos destacaremos a *Motorola* puesto que ha desarrollado una serie de funciones que permite trabajar con su microcontrolador *MPC555*, a través del compilador cruzado *MetroWerks CodeWarrior*, desde *Simulink*.

3.2.2.2 Ejecución del Modelo.

Previo al estudio de los diferentes estilos de generación de código se requiere conocer detalladamente cómo se ejecuta éste. A partir del modelo definido bajo *Simulink*, *RTW* genera el código expresado en lenguaje *C++*, en el que puede introducir su propio código gracias a las funciones *S*; además, *RTW* también proporciona una interfaz en tiempo de ejecución que permite la ejecución del código. Tanto los algoritmos del modelo como la propia interfaz se compilan dando lugar a una aplicación autónoma. El siguiente diagrama muestra una vista del ejecutable:



En general, el diseño conceptual de un controlador de la ejecución del modelo no cambia entre el estilo del prototipado rápido y el del caso embebido. Las siguientes secciones describen la ejecución del modelo para entornos monotarea y multitarea, tanto en el caso simulado (no-tiempo real) como en el tiempo real. Para la mayoría de los modelos, el entorno multitarea proporcionará una mayor eficiencia.

Los conceptos siguientes serán útiles para la descripción de cómo se ejecuta el modelo:

- *Inicialización*: inicia los códigos del modelo y de la interfaz en tiempo real.
- *SalidasModelo*: llama a todos los bloques del modelo cuyo evento de tiempo ha llegado obteniéndose, así, la respuesta correspondiente.
- *ActualizarModelo*: llama a todos los bloques del modelo que, para el instante dado, requieren la actualización de sus estados discretos.

- *DerivarModelo*: su función es análoga al concepto anterior, salvo que se aplica a estados continuos.

El siguiente pseudo - código muestra la ejecución de un modelo durante la simulación monotarea:

```
main()
{
        Inicialización
        Mientras (tiempo < tiempo final)
        {
                SalidasModelo       -- para pasos de tiempo mayores
                RegistroTXY         -- Se registra el tiempo, los estados y las salidas
                ActualizarModelo    -- para pasos de tiempo mayores
                Integrar            -- integración de los estados continuos para pasos de tiempo menores
                {
                        DerivarModelo     -- para actualizarlos
                                Hacer 0 o más
                                {
                                        SalidasModelo
                                        DerivarModelo
                                }Fin Hacer (el número de iteraciones depende del método de resolución)
                        }Fin Integrar
        }Fin Mientras
        Finalización
}
```

La fase de iniciación comienza primero, la cual asigna valores coherentes de los estados del modelo dándole paso al motor de ejecución; entonces el modelo se ejecuta, un paso cada vez. Primero *SalidasModelo* se lleva a cabo en el instante *'t'*, a partir de lo cual los datos de *E/S*[2] del espacio de trabajo se registran y *ActualizarModelo* logra los valores nuevos de los estados discretos. A continuación, si el modelo posee estados continuos, *DerivarModelo* integra las derivadas de éstos para generar sus valores para el instante $t_{nuevo} = t + h$, donde *'h'* es el paso de integración. El tiempo avanza y el proceso se repite.

El pseudo – código que a continuación se muestra presenta la ejecución de un modelo bajo la simulación multitarea:

```
main()
{
    Inicialización
    Mientras (tiempo < tiempo final)
    {
        SalidasModelo(tid=0)                    -- identificador de tarea (tid) a cero
        RegistroTXY
        ActualizarModel (tid=0)          -- identificador de tarea (tid) a uno
        Integrar                         -- integración para sistemas de estados continuos
        {
            DerivarModelo
            Hacer 0 o más
            {
```

[2] Entrada y Salida.

```
                SalidasModelo(tid=0)
                DerivarModelo
            }
            Fin Hacer(el número de iteraciones depende del método de integración)
            Integrar las derivadas y actualizar los estados continuos.
        }Fin Integrar
        Para i = 1 Hasta NumTids – siendo NumTids el número de tareas
        {
            SalidasModelo(tid=i)
            ActualizarModelo(tid=i)
        }Fin Para
    }Fin Mientras
    Finalizar
}
```

El modo multitarea posee la complejidad añadida del identificador de tarea *–task identifier –*; esto permite utilizar interrupciones solapadas, cada una asociada a una tarea.

El siguiente pseudo – código muestra la ejecución de un modelo en un sistema multitarea en tiempo real gracias al uso de interrupciones:

```
rtOneStep() – Un paso en tiempo real.
{
    Comprobación si se alcanza el instante de interrupción
    Habilitar la interrupción rtOneStep
    SalidasModelo
    RegistroTXY
    ActualizarModelo
    Integrar
    {
        DerivarModelo
        Hacer 0 o más
        {
            SalidasModelo
            DerivarModelo
        }Fin Hacer (el número de iteraciones depende del método de resolución)
        Integración de las derivadas para actualizar los estados continuos
    }Fin Integrar
}

main()
{
    Inicialización -- (incluyendo la instalación de rtOneStep como una rutina de servicio de
                    -- interrupción –ISR- para un reloj en tiempo real).
    Mientras(tiempo < tiempo final)
    {
        Tarea de fondo
    }Fin Mientras
    Enmascarar las interrupciones –inhabilitar la ejecución de rtOneStep
    Completar cualquier tarea de fondo
    Finalizar
}
```

En este caso se ejecuta el modelo en tiempo real –por la *ISR*[3]– sólo cuando se produce una interrupción por el generador de eventos de tiempo, siendo ésta periódica. La rutina de servicio de la interrupción detiene la tarea de fondo para dar paso a la ejecución de la rutina del modelo denominada *rtOneStep*.

[3] *ISR*: *Interruption Routine Service* o rutina de servicio de la interrupción.

Por ejemplo: si el modelo describe a un controlador operando a 100 Hz entonces cada 0.01 s la tarea de fondo será interrumpida. Durante este período de tiempo, como máximo, el controlador debe leer su entrada del convertidor analógico – digital, calcular su salida, escribirla sobre el convertidor digital – analógico y actualizar sus estados. Tras esto, el control se devuelve a la tarea de fondo. Todos estos pasos deben ocurrir antes de que la nueva interrupción se dé.

El siguiente pseudo – código presenta cómo se ejecuta el modelo en un sistema multitarea que gestiona el tiempo real mediante interrupción:

```
rtOneStep()
{
    Comprobar si se alcanza el instante de interrupción
    Habilitar la interrupción por rtOneStep
    SalidasModelo(tid=0)
    RegistroTXY
    ActualizarModelo(tid=0)
    Integrar
    {
        DerivarModelo
        Hacer 0 o más
        {
            SalidasModelo(tid=0)
            DerivarModelo
        }Fin Hacer (el número de iteraciones depende del método de integración)
        Integrar las derivadas para actualizar los estados continuos
    }Fin Integrar
    Para i = 1 Hasta NumTareas –siendo NumTareas el número máximo de tareas
    {
        Si se alcanza el instante de interrupción para la tarea i
        {
            SalidasModelo(tid=i)
            ActualizarModelo(tid=i)
        }Fin Si
    }Fin Para
}

main()
{
    Inicialización –incluyendo la instalación de rtOneStep como ISR para el reloj de tiempo real
    Mientras (tiempo < tiempo final)
    {
        Tarea de Fondo
    }
    Enmascarar las interrupciones –inhabilitar la ejecución de rtOneStep
    Completar cualquier tarea de fondo
    Finalizar
}
```

Ejecutar modelos mediante interrupciones en un entorno multitarea en tiempo real es muy similar al caso previo monotarea, salvo que las interrupciones solapadas se emplean para la ejecución concurrente de las tareas.

Si la aplicación generada por *RTW* se ejecuta bajo el sistema operativo *Windows*, se requiere el objetivo *Real Time Windows*. Éste se instala automáticamente en el sistema a lo largo de la instalación del paquete *Matlab*.

Hay que tener en consideración los tiempos de invocación de las tareas en un sistema operativo de tiempo real, tanto por interrupción como a través de primitivas, para asegurar que el código del modelo se ejecuta en su totalidad antes de que otra tarea sea invocada. Esto incluye el tiempo invertido en el proceso de lectura y escritura de datos del *hardware* externo. El intervalo de muestro debe ser lo suficientemente largo para permitir la ejecución del código del modelo entre dos invocaciones de tarea consecutivas.

Un programa en tiempo real puede no necesitar el 100% del tiempo de *CPU*[4]; esto proporciona la oportunidad de ejecutar tareas de fondo durante el tiempo libre, éstas incluyen operaciones tales como la escritura o lectura de datos respecto a un registro de desplazamiento –*buffer*– o archivo, permitiendo el acceso a los datos del programa por herramientas de seguimiento. Resulta crítico que el sistema asegure que la tarea de fondo podrá ser interrumpida en el instante adecuado para lograr una ejecución en tiempo real.

El código para prototipado rápido define las siguientes funciones que constituyen los puntos de comunicación con la interfaz en tiempo de ejecución:

- *Model()*: La función de registro del modelo que inicia las áreas de trabajo que él necesita. Ésta llama a las funciones *MdlInitializeSizes* y *MdlInitializeSampleTimes*, las cuales son análogas a las usadas por las funciones *S*.

- *MdlStart(void)*: Ejecuta la interfaz en tiempo de ejecución del modelo, presentando varias partes:
  - Código de inicio de los estados de cada bloque.
  - Código generado por la rutina de arranque de cada bloque del modelo.
  - Código asociado a la ejecución de cada bloque del diagrama de *Simulink*.
- *MdlOutputs(int_T tid)*: Actualiza la salida de los bloques en los instantes apropiados.

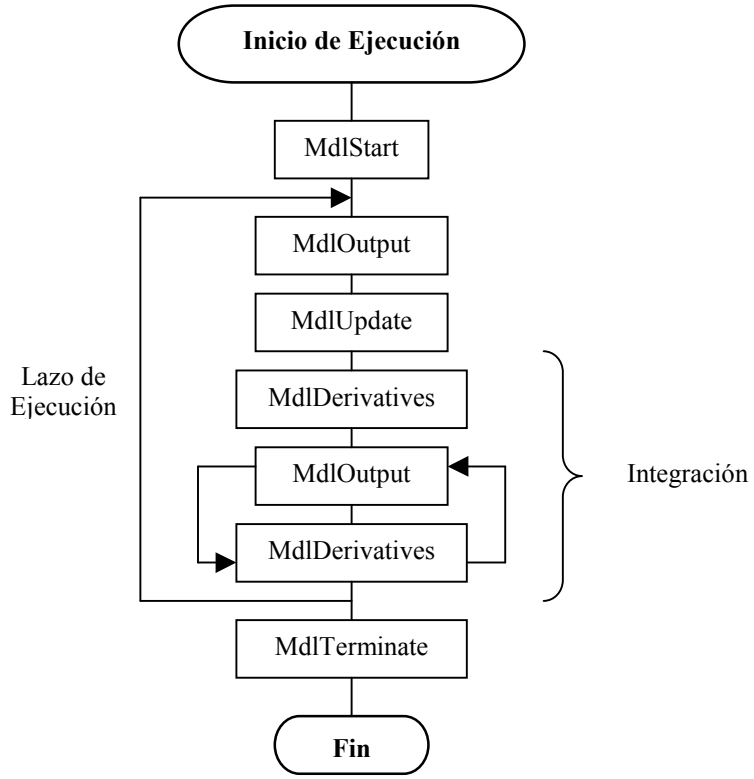
- *MdlUpdate(int_T tid)*: Actualiza los estados discretos.

- *MdlDerivatives(void)*: Devuelve las derivadas de los bloques llamándose para resolver las fases de integración de los estados continuos.

[4] *CPU*: *Control Process Unit*.

- *MdlTerminate(void)*: Contiene el código de finalización para cualquier bloque. Se llama por la interfaz en tiempo real, como parte del proceso de finalización del programa en tiempo real.

Mediante el siguiente diagrama de flujo se describe las relaciones existentes entre estas funciones:



El código generado para sistemas embebidos utiliza sus propias funciones, siendo su interdependencia similar al caso del prototipado rápido.

Las funciones que genera *Real Time Workshop Embedded Coder* son:

- *model_initialize*: Lleva a cabo la iniciación del modelo y se llama al comenzar a su ejecución.

- *model_step(int_T tid)*: Contiene el código que actualiza y obtiene la salida de todos los bloques que describen el modelo. Aparece sólo cuando se opta porque ambos procesos se aúnen en una única función.

- *model_output(int_T tid)* y *model_update(int_T tid)*: La primera da lugar a la obtención de las salidas de todos los bloques y, la segunda actualiza sus estados. Aparece cuando la opción anterior no se habilita.

- *model_terminate*: Sólo se usa cuando se pretende realizar un algoritmo de tiempo finito y contiene la terminación ordenada del modelo.

3.2.2.3 Arquitectura de Prototipado Rápido.

El prototipado tradicional necesita de varios grupos de diseñadores, entre los que la dependencia en el desarrollo de su trabajo es elevada puesto que hasta que una fase del diseño no se termina no se puede iniciar la siguiente, dándose la aparición de numerosos cuellos de botella en el proceso. Con el prototipado rápido [B18] se agiliza enormemente la comunicación de las diferentes fases del diseño de prototipos porque un mismo grupo de trabajo puede observar la interdependencia que entre ellas existe.

El prototipado rápido permite realizar las siguientes actividades:

- Plantear las soluciones gráficamente en un entorno de modelado basado en los diagramas de bloques.
- Evaluar las capacidades de un sistema a priori, sin necesidad de haberlo plasmado en soporte hardware ni de haberlo diseñado mediante programa ni ciñéndonos a un diseño ya fijado.
- Detallar el diseño de nuestro modelo trabajando con los algoritmos y sus prototipos.
- Sintonizar los parámetros de nuestro modelo mientras se está ejecutando en tiempo real.

El proceso de prototipado rápido combina las fases de diseño del *hardware*, del *software* y de los algoritmos; así elimina los cuellos de botella potenciales permitiendo que los ingenieros de diseño tengan acceso directo a los resultados de cada fase, de forma que puedan idear e implementar soluciones antes de construir sistemas de soporte *hardware* excesivamente costosos.

La construcción automática de programa permite realizar cambios de diseño, de forma directa, sobre el diagrama de bloques; de esta forma se facilita el desarrollo de algoritmos –incluyendo la edición, el compilado, el enlazado y la descarga del código al sistema objetivo– en un único proceso. Se parte de la descripción y simulación bajo *Simulink* para proseguir, cuando los resultados de la misma son satisfactorios, con la generación del código *C++* gracias a *RTW*.

Mediante la interfaz externa del prototipado rápido se permite la interacción con el sistema objetivo desde *Simulink*, pudiéndose sintonizar y ajustar los parámetros del modelo en tiempo de ejecución. Tras esto se puede proceder con la fabricación de los modelos en serie, puesto que el prototipo ya ha sido evaluado y validado.

3.2.2.3.1 Entorno de Programa para Prototipado Rápido.

Los archivos generados a partir de un modelo especificado en *Simulink* implementan su sistema de ecuaciones, contiene los parámetros de cada bloque y realiza su puesta en marcha. El entorno de programa de *RTW* proporciona el código fuente adicional que se requiere para construir una aplicación completa y autónoma.

Los módulos de aplicación y el código generado por el modelo de *Simulink* se implementan mediante el uso de *API*; éste define la estructura de datos, llamada modelo en tiempo real, que encapsula todos los datos asociados a la descripción del modelo.

La estructura de un programa en tiempo real está compuesta de tres componentes. Las dos primeras de ellas forman parte de la interfaz en tiempo real de *RTW*, mientras que la tercera constituye las componentes de aplicación.

3.2.2.3.2 Interfaz en Tiempo de Ejecución.

Dividiremos las componentes del código fuente entre las dependientes e independientes del sistema.

- ***Componentes Dependientes del Sistema.***

Éstas contienen la función principal del programa, cuya actividad abarca el control de la periodicidad de los eventos de tiempo, la generación de las tareas, la instalación de las rutinas de gestión de interrupciones, la habilitación del registro de datos y la realización de la comprobación de errores. En los programas de aplicación de *RTW*, la función *main* lleva a cabo las siguientes operaciones:

- La iniciación.
  - Inicia una serie de parámetros numéricos especiales: *rtInf*, *rtMinusInf* y *rtNaN*.
  - Llama a la función que construye el modelo, del mismo nombre que éste, para conseguir un puntero al modelo en tiempo real. Así, se construye y se le asigna un valor a todos los campos del modelo y a cualquier función S que éste contenga.
  - Inicia la información sobre los tiempos de muestreo llamando a la función *MdlInitializeSizes*.
  - Inicia los estados mediante la función *MdlStart*.
  - Pone en marcha el generador de eventos de tiempo o timer.
  - Define la tarea de fondo y permite el registro de datos.

- La ejecución del modelo.
  - Ejecuta la tarea de fondo, es decir, se comunica con el equipo *host*[5] durante las simulaciones o introduce un intervalo de espera hasta el siguiente tiempo de muestreo.
  - Ejecuta el modelo, que se inicia por interrupción.
- La terminación del programa.
  - Si se trata de una aplicación de tiempo finito, se destruye la estructura de datos que representa al modelo en tiempo real, se libera la memoria utilizada y se guarda la registro de datos en un archivo (si fuera posible).

Todas estas funciones se encuentran, tras la generación de código *C++*, en el archivo *rtmain.c* o en *ertmain.c*, para el caso de usar el codificador embebido de *RTW*.

- ***Componentes Independientes del Sistema.***

Aquí se incluye el código que define, genera y destruye la estructura de datos del modelo en tiempo real. Para cada tiempo de muestreo, el programa *main* cede el testigo a la función de ejecución del modelo, la cual ejecuta un paso de su dinámica; en éste se leen las entradas del *hardware* externo, se calculan las salidas del modelo, se escriben las salidas al *hardware* externo y, entonces, se actualizan los estados y el tiempo.

Cuando se utilizan estados continuos en el modelo, su actualización requiere un proceso de integración, en el cual se usa el método de *Euler* de primer orden: $x_{k+1} = x_k + h\ dx/dt$; siendo *'x_k'* el estado para el instante k-ésimo y *'h'* el paso de la iteración variable. Sin embargo, resulta recomendable usarlo fijo cuando se necesita garantizar que todas las tareas se llevan a cabo en un tiempo determinado, es decir, cuando se trata de un modelo en tiempo real.

Los archivos en *C++* asociados a los componentes independientes del sistema son:

- *ode1*.c, *ode2.c*, *ode3.c*, *ode4.c* y *ode5.c*: Realizan la integración de los algoritmos para las aplicaciones en tiempo real, trabajando todos con tiempos de iteración fijos.
- *rt_sim.c* o *ert_sim.c*: Realiza las actividades necesarias para un paso del modelo.
- *simstruc.h*: Contiene las definiciones de la estructura de datos de *Simulink*.
- *simstruct_types.h*: Contiene las definiciones de varios eventos, como la habilitación de subsistemas y los pasos por cero; también constata las variables de registro.

[5] *Host*: anfitrión, del inglés.

3.2.2.3.3 Componentes de Aplicación.

Estas componentes contienen el código generado a partir del modelo de *Simulink*, incluyendo también el de cualquier función *S*; así pues, las funciones relativas a dicho código implementan el modelo en *C++* que se ha descrito en el apartado *Componentes Dependientes del Sistema*.

La estructura de datos del modelo denominada *SimStruct* encapsula sus datos, de forma que existen interfaces normalizados para acceder a ellos, y lo describe completamente. Su contenido incluye: los parámetros del modelo, sus entradas y salidas.

En cada fase de la ejecución del modelo en tiempo real requiere de una serie de funciones, veámoslas:

- Iniciación del Programa *main*
  - Función de registro del modelo: *model*.
  - Inicia los tamaños en el modelo en tiempo real: *MdlInitializeSizes*.
  - Inicia tiempos de muestreo y valores de continua: *MdlInitializeSampleTimers*.
  - Arranca el modelo: *MdlStart*.
- Ejecución del modelo.
  - Calcula las salidas del sistema y de sus bloques: *MdlOutputs*.
  - Actualiza el vector de estados discretos: *MdlUpdate*.
  - Calcula las derivadas de los estados continuos: *MdlDerivatives*.
- Finalizado del Programa *main.*
  - Terminación ordenada y finalización del programa: *MdlTerminate*.

Si el modelo contiene funciones *S*, su código fuente debe estar en la ruta del compilador; esto se especifica en la plantilla que lo invoca. Cuando el modelo necesite la ejecución de una de estas funciones, se invoca a los punteros de las funciones internas de su estructura de datos; permitiéndose varias instancias, de una misma función *S*, en el modelo.

Es posible editar este tipo de funciones directamente sobre el código del modelo mediante la inclusión de código; para ello se ha de definir la función *S* como archivo *C-MEX* en la plantilla, es decir, constituyendo una librería de enlace dinámico. De esta forma, se mejora la eficiencia en el uso de estas capacidades puesto que se hace innecesaria la llamada a las propias funciones.

Cuando *RTW* interviene, se generan los archivos siguientes:

- *model.c*: Contiene el código *C* que describe el diagrama de bloques de *Simulink*, implementando sus ecuaciones y realizando el proceso de inicialización y la actualización de las salidas.

- *model.h*: Se trata de un archivo de cabeceras que contiene los parámetros de la simulación del diagrama de bloques, las estructuras de *E/S*, las de trabajo y otras. Incluye el siguiente.

- *model_private.h*: Contiene las declaraciones de los parámetros y de las señales exportadas.

- *rtmodel.h* o *ertmodel.h*: Incluye los puntos de entrada [B31] y las estructuras de datos específicas del modelo; esto permite que el programa *main*pueda hacer referencia a los archivos generados por *RTW* sin conocer los nombres de los modelos involucrados.

Para un caso concreto, estos archivos sustituyen *model* por el nombre del modelo concreto con el que se esté trabajando desde *Simulink*, salvo para el último archivo, cuyo nombre es invariable.

3.2.2.4 Entorno de Programa Embebido.

Este entorno está dirigido para el desarrollo de aplicaciones imbuidas. Su arquitectura se desprende del siguiente esquema:

- **Interfaz en Tiempo de Ejecución.**
  - Componentes Dependientes del Sistema: programa *main*, generación y gestión de los eventos de tiempo, manejo de interrupciones, controladores de *E/S* y registro de datos.
  - Componentes Independientes del Sistema: integradores (*ode1.c*, *ode2.c*, *ode3.c*, *ode4.c*, *ode5.c*), coordinador de la ejecución del modelo: *rt_sim.c*.
- **Componentes de Aplicación:** Código generado a partir del modelo, funciones del código como *MdlOutputs*, funciones *S* directamente incluidas y parámetros del modelo.

Existe una gran similitud con la arquitectura diseñada para el caso de la generación rápida de prototipos (*rapid prototyping*). La principal diferencia se manifiesta en la falta de la estructura de datos *SimStruct* y que no contempla el uso de ciertos tipos de funciones *S*.

Esta arquitectura se utiliza por el traductor *Real Time Workshop Embedded Coder*, el cual simplifica, frente al caso del prototipado rápido, el proceso de generación de código. El caso embebido contiene el mismo esquema conceptual de capas que el anterior, pero cada capa se ha simplificado y reducido.

Para un mayor grado de detalle véase el apartado *3.3 Real Time Workshop Embedded Coder*

3.2.3 La Generación de Código y el Proceso de Construcción.

Para dar comienzo a este proceso, se ha de iniciar la interfaz *Simulink*; para ello se puede hacer uso de la línea de comandos de *Matlab* tecleando *simulink* o bien pinchando sobre su icono: 

Se abre la ventana *Simulink Library Browser*, la cual nos permite visualizar, jerárquicamente, cada uno de los elementos de biblioteca que posee *Simulink*. Mediante la opción *Model* del campo *New* del menú *File*, abrimos un nuevo modelo en una ventana independiente. Sobre ella podremos editar, con un diagrama de bloques, el modelo dinámico de nuestro sistema; asimismo se nos facilitan tanto las funciones de simulado y depuración bajo *Simulink* como las asociadas a la generación de código fuente e invocación de los compiladores adecuados relativas a *RTW*.

3.2.3.1 Interfaz de Usuario de Real Time Workshop.

Existen muchos parámetros y opciones que afectan en la manera en la que *RTW* genera el código *C++* a partir del modelo *Simulink* y construye un archivo ejecutable. Para fijarlos, se trabaja con los campos del cuadro de diálogo *Simulation Parameters*. Este cuadro se puede abrir de dos maneras diferentes sobre la ventana de edición del modelo:

- Eligiendo la opción *Simulation Parameters* del menú *Simulation*.
- Dentro del menú *Tools*, se escoge *Real-Time Workshop* para seleccionar la opción *Options...*

La lengüeta *Real-Time Workshop* se encuentra dividida en dos secciones; la superior contiene el menú *Category* y el botón *Build* y la inferior, las opciones de configuración. Este menú posee varias opciones, es decir:

- *Target Configuration*: Se trata de un conjunto de opciones de alto nivel relacionado con el control de los procesos de generación y construcción de código.

- *TLC debugging*: Define las opciones de los perfiles de ejecución y depurado del compilador apropiado al leguaje del sistema objetivo, es decir, del *Target Language Compiler*.

- *General code generation options*: Contempla las opciones para la generación de código que son comunes a todas las configuraciones del sistema objetivo.

- *General code appearance options*: Abarca las opciones del formato de los identificadores y del código que son comunes a todas las configuraciones del sistema objetivo.

- *Target-specific code generation options*: Uno o más grupos de opciones que son específicos a la configuración del objetivo seleccionado, estando éstas definidas en sus plantillas.

Pulsando en el botón *Build* inicia el proceso de construcción y de generación de código. Los métodos siguientes son exactamente equivalentes a pulsar este botón:

- Seleccionando *Build Model* de la opción *Real-Time Workshop* del menú *Tools* (*Ctrl-B*).

- Invocando el comando *rtwbuild* en la línea de comandos de *Matlab*. La sintaxis de esta orden es *rtwbuild nombreDelModelo* o *rtwbuild('nombreDelModelo')*, donde *nombreDelModelo* es el nombre de la ruta de directorio del modelo fuente. Esta opción se utiliza cuando ésta no está cargada en *Simulink*.

Cuando la opción *Generate code only* esté seleccionada, el botón *Build* se torna en *Generate Code*, de forma que únicamente llevaría a cabo la generación de código sin iniciar la construcción de una aplicación autónoma.

De todas las categorías posibles, veremos las que más interés despierta respecto a la generación de código y a la relación con el sistema objetivo.

Siendo las opciones de configuración:

- *System target file*: Especifica el fichero en el que se indican las opciones del objetivo para el *TLC*[6]. Con el botón *Browse...* se muestran todos los ficheros de configuración que están disponibles, tanto los proporcionados con *RTW* como los editados por el usuario, los cuales contienen las características de los tres campos de esta categoría. Podemos seleccionar una

[6] *TLC*: acrónimo de *Target Language Compiler*, es decir, *Compilador del Lenguaje del Objetivo*.

de las posibilidades preestablecidas o teclear, directamente, la ruta de directorio del archivo *tlc* que queremos usar.

- *Template makefile*: Indica el nombre de la plantilla concreta para trabajar con el compilador y depurador adecuados a nuestra configuración. Este campo toma el valor adecuado de forma automática según el archivo *tlc* seleccionado; si queremos especificar su valor debemos introducir el nombre de la plantilla incluyendo su extensión –*tmf* –.

- *Make command*: Se trata de una orden de alto nivel que controla el proceso de construcción de *RTW*. Además del nombre de esta orden, se pueden añadir argumentos como las opciones del compilador específico, como rutas de directorio. Al ser invocada la aplicación de compilado, estos argumentos se pasan a través de la línea de comandos.

Dentro de las opciones, que son muchas, resulta de interés práctico la posibilidad de generar un informe del proceso de construcción del código en formato hipertexto.

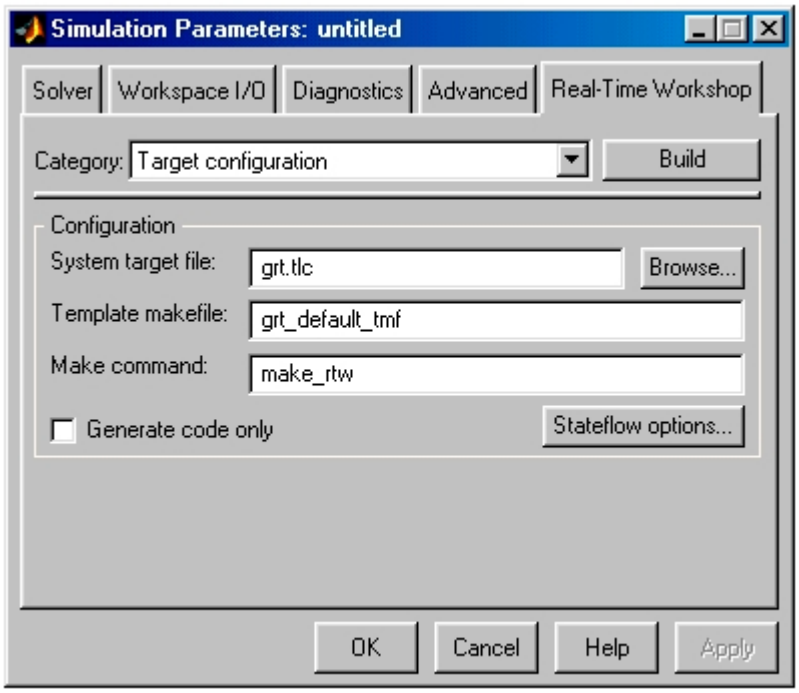


Figura: Configuración del Objetivo.

3.2.3.2 Parámetros de Simulación y la Generación de Código.

Esta sección presenta como los parámetros de simulación de nuestro modelo modifican la generación de código. Este tema se organiza en torno a 4 lengüetas del cuadro de diálogo *Simulation Parameters*, es decir: *Solver*, *Workspace*, *Diagnostics* y *Advanced*. Veremos las más interesantes desde el punto de vista de la generación de código.

- *Solver*:
  - *Solver time*: Permite definir el comienzo y el final del tiempo de simulación; si éste último se define como *inf*, entonces la simulación se ejecuta indefinidamente.
  - *Solver options*: Podemos especificar que el paso de iteración sea fijo o variable (estados continuos); además se puede indicar el método de actualización o de integración de los estados, respectivamente.
- *Workspace I/O*: Esta sección expone diferentes métodos por los cuales *RTW* puede guardar datos del programa generado en un archivo para su análisis posterior; siendo sus campos:
  - *Load from workspace*: Permite que las variables del espacio de trabajo de *Matlab* se toman como entradas por el proceso de la simulación.
  - *Save to workspace*: Los resultados de la simulación se transfieren a variables del espacio de trabajo para que podamos trabajar con ellas fuera de línea.

Los datos se almacenan en el archivo *model.mat*, siendo *model* en nombre del modelo concreto bajo *Simulink*.

- *Diagnostics*: Especifica qué acción debe realizarse cuando se dan varias condiciones en el modelo. Se puede indicar si se ignora una condición, aviso o mensaje dados o si se alcanza cierto error. Bajo cualquier concepto, si se alcanza un error durante la construcción del modelo, este proceso se detiene. En el cuadro de las opciones de configuración se muestran los diferentes eventos y cómo son tratados: como errores, como avisos o, simplemente, ignorados.

- *Advanced*: Incluye múltiples opciones que afectan a la capacidad del código generado, permitiendo especificar cómo los parámetros de los bloques serán representados en el código y cómo se relacionan con él. Por otro lado, es posible optimizar la cantidad de memoria utilizada para albergar el código, tanto en la cantidad como en la eficiencia de su uso. Esto se manifiesta, entre otras cosas, en la posibilidad de definir el número de bytes necesario para codificar los tipos de datos básicos.

3.2.3.3 Seleccionar una Configuración para el Sistema Objetivo.

El proceso de generación de código para un objetivo específico se controla con tres elementos:

- Un archivo del sistema objetivo con extensión *tlc*
- Una plantilla que indica las características concretas de la interfaz con el compilador concreto.
- Un comando para el compilador y el depurador.

Al seleccionar archivos específicos para distintos sistemas objetivos permite especificar esta configuración en un único paso. Para que podamos acceder a él hemos de pulsar el botón *Browse...* de la lengüeta *Real-Time Workshop* del cuadro de diálogo *Simulation Parameters*.

Para seleccionar uno tan sólo hay que hacer *clic* sobre él y pulsar *Ok* o bien, hacer doble clic sobre la opción deseada. Así, de forma automática, *RTW* selecciona el archivo apropiado para el sistema objetivo, la plantilla y el comando para el compilador.

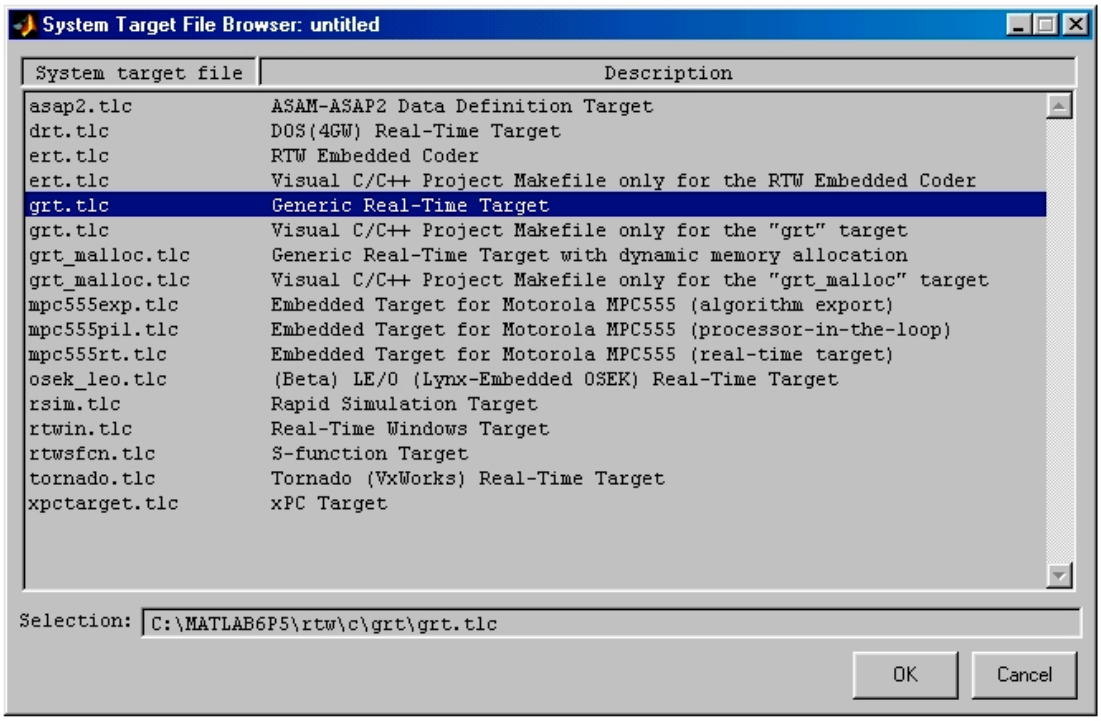


Figura: Selección de los archivos del Sistema Objetivo.

3.2.3.4 Construir un Archivo Ejecutable.

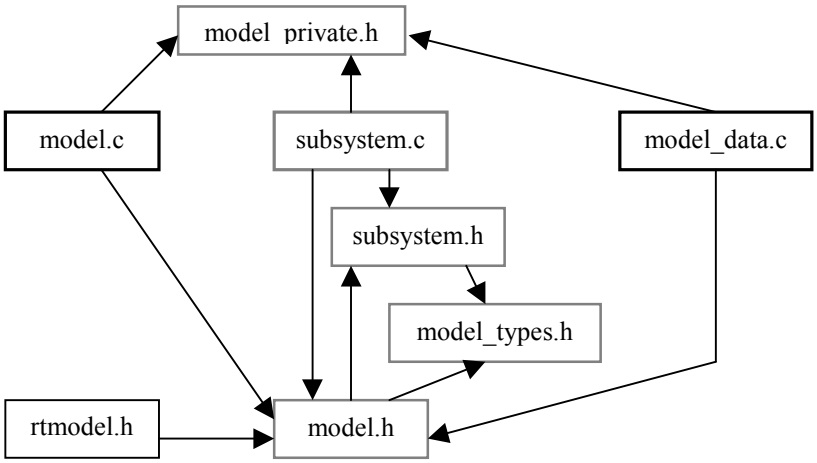
RTW genera [B16] código *C++* en un conjunto de ficheros fuente que varían muy poco para sistemas objetivos diferentes; no todos los posibles archivos se generarán para todos los modelos, puesto que ciertos archivos aparecerán sólo si se incluye en el diagrama de bloques ciertos subsistemas o tipos de datos particulares.

Los archivos que genera *Real Time Workshop Embedded Coder* varían ligeramente, aunque significativamente del conjunto que se describirá a continuación:

- *model.c*: Contiene los puntos de entrada –funciones– para todos los códigos que implementan el algoritmo del modelo.
- *model_private.h*: Contiene las definiciones y los datos locales que se requieren por el modelo y por los subsistemas. Se incluye en el código fuente del modelo de forma automática.
- *model.h*: Define la estructura de datos del modelo y el interfaz público de los puntos de entrada y las estructuras de datos, como por ejemplo, para la estructura del modelo en tiempo real (*model_rtM*) que se accede vía *macros*.
- *model_data.c* (condicional): Contiene las declaraciones de la estructura de datos de los parámetros y de los bloques de entrada / salida. Estas estructuras se declaran externas –*extern*- en *model.h*.
- *model_types.h*: Provee las declaraciones para la estructura del modelo en tiempo real y de los parámetros.
- *rtmodel.h*: Contiene las directivas *#include* requeridas por los módulos estáticos del programa principal como son *grt_main.c* o *ert_main.c* y *grt_malloc_main.c* o *ert_malloc_main.c*. Si se edita un módulo de programa principal propio, se debe tener en cuenta que se ha de incluir *rtmodel.h* explícitamente.
- *model_pt.c* (opcional): Proporciona las estructuras de datos y habilita que un programa ejecutándose acceda a los parámetros del modelo sin usar el modo externo.
- *model_bio.c* (opcional): Proporciona las estructuras de datos que habilitan que el código acceda a las salidas de los bloques.

Todos los módulos de código descritos se escriben en la carpeta de construcción dentro del directorio de trabajo.

Este diagrama muestra que el archivo de cabecera (*model.h*) incluye a todos los archivos de cabecera dependientes (*subsystem.h*); cada flecha indica una relación de inclusión o dependencia.



3.2.3.4.1 Compilado y Enlazado.

Una vez que se haya generado el código *C++* se determina si se continúa o no con el proceso de compilado y enlazado para lograr un programa ejecutable. Esta decisión se gobierna por los parámetros siguientes:

- Opción *Generate code only*: Si se selecciona, se omite la fase de compilado.
- Objetivo *Makefile-only*: Las versiones de los archivos de compilador de *Visual C/C++ Project* del tiempo real genérico (*grt*), del caso dinámico (*grt_malloc*) y del caso embebido (*RTW Embedded Coder*) generan un proyecto para el compilador *Visual C/C++*: *model.mak*. Para construir un archivo ejecutable, se debe abrir este proyecto con *Visual C/C++ IDE* para, posteriormente, compilar y enlazar el código del modelo.
- La Variable *HOST* en la plantilla: Esta variable identifica el tipo de sistema sobre el que se ejecutará la aplicación en tiempo real, pudiendo tomar uno de estos tres valores: *PC*, *UNIX* o *ANY* (cualquiera); esta última opción es útil cuando se usan compiladores cruzados.

3.2.3.5 Elegir y Configurar el Compilador.

El proceso de construcción de *RTW* depende de la correcta instalación de uno o más de los compiladores admitidos. Nótese que un compilador, en este contexto, se refiere al entorno de desarrollo conteniendo un enlazador y un traductor a código máquina. El proceso de construcción también requiere la selección de la plantilla, la cual determina qué compilador se aplicará sobre el código *C++* generado.

Esta sección discute cómo instalar un compilador y cómo elegir su plantilla específica cuando se trabaja con sistemas basados en *Windows*. Primero se debe instalar uno o más de los compiladores admitidos por *Matlab*; después, se define una variable de entorno asociada a cada uno de ellos.

3.2.3.6 Plantillas y Opciones del Compilador.

RTW incluye un conjunto de plantillas ya construidas que han sido diseñadas para construir programas para sistemas objetivo específicos. Existen dos tipos de plantillas:

- Específicas para un sistema de desarrollo o compilador particular: Se usa la convención en la denominación de estas plantillas; por ejemplo: *grt_vc.tmf* se usa para construir programas en tiempo real genéricos desde *Visual C/C++ IDE* o *ert_lcc.tmf* se usa con la intención de construir aplicaciones para *Real Time Workshop Embedded Coder* bajo el compilador *LCC*[7].

- Plantillas por defecto: Éstas hacen que los diseños del modelo se puedan trasladar de un sistema a otro, sin más que elegir el compilador específico de la nueva arquitectura. Estas plantillas se denominan *objetivo_default_tmf*. Así, para el caso del objetivo en tiempo real genérico (*GRT*) la plantilla que se usa es *grt_default_tmf* o, para el caso que sea embebido (*ERT*), *ert_default_tmf*.

3.2.4 Formato de los Códigos Generados.

RTW facilita cuatro formatos de código diferentes, cada uno especifica un entorno de trabajo para la generación de código apropiada para la aplicación específica. Los formatos y sus aplicaciones son:

- *Tiempo Real* con localización estática y dinámica: Creación rápida de prototipos.

[7] *LCC*: acrónimo de *Local C Compiler*, el compilador local para *C* de *Matlab*.

- Funciones *S*: Creación de funciones *S* propietarias en librerías de enlace dinámico –*dll* – u objetos *MEX*, así como la utilización del mismo código en varias ocasiones y la aceleración del proceso de simulación.
- Código *C++* embebido: Sistemas profundamente embebidos.

Este apartado plantea la relación entre los formatos del código y las configuraciones disponibles de sistemas objetivo, además considera los factores a tener en cuenta a la hora de elegir el formato y el objetivo.

3.2.4.1 Elegir un Formato de Código para la Aplicación.

La elección de un formato para el código es la opción de generación de código más importante. Al elegir un objetivo, implícitamente se elige un formato. Típicamente, el archivo del sistema objetivo especificará el formato al darle un valor a la variable *TLC* de nombre *CodeFormat*.

Por ejemplo, con la siguiente línea, dentro del archivo *ert.tlc*, podremos indicar que se escoge el formato embebido:

%assign CodeFormat = Embedded-C

Si no se asigna un código, se toma el valor por defecto, siendo: *RealTime*; tal y como se declara en el archivo *grt.tlc*. Ante el desarrollo de un objetivo a medida, se debe considerar cuál será el formato mejor para la aplicación y definir *CodeFormat* coherentemente.

- Si la aplicación no tiene restricciones significativas en cuanto al tamaño del código, al uso de la memoria o de la pila, entonces resulta aconsejable utilizar el objetivo en tiempo real genérico (*GRT*). Este formato presenta el código más comprensible y admite la gran mayoría de los bloques constructivos. Además, es capaz de ejecutar restricciones en tiempo real estrictas; pero si, por ciertos motivos, no se pueden satisfacer las limitaciones de tiempo, se incurre en un error de sistema catastrófico, deteniendo la ejecución del código.

- Por el contrario, si la aplicación exige límites al tamaño del código o al uso de la memoria o que posee una estructura de llamadas simple entonces se debe elegir el objetivo *Real Time Workshop Embedded Coder*, el cual utiliza el formato *Embedded-C*.

- Finalmente, si se usa el formato para funciones *S*, se contempla el modelo como un módulo integrante en un proyecto más amplio. Normalmente, el código así escrito es más compacto que el que se genera automáticamente por *RTW*, luego se gana en velocidad de la simulación.

Capacidad	GRT	GRT dinámico	RTW Embedded Coder	Real Time Windows
Localización estática de memoria	X		X	X
Localización dinámica de memoria		X		
Tiempo continuo	X	X		X
Funciones *S* para *dll*	X	X		X
Funciones *S* incluidas directamente	X	X	X	X
Minimiza el uso de *RAM / ROM*			X	
Admite modo externo	X	X	X	X
Diseñado para prototipado rápido	X	X		X
Producción de código			X	
Permite tiempo real estricto	X*	X*	X*	X
Incluye ejecutable no en tiempo real	X	X	X	X
Múltiples instancias de un modelo		X		

* Los archivos principales *rt_main.c* de los formatos por defecto *GRT*, *GRT dinámico* y *ERT* emulan la ejecución en tiempo real estricto cuando no se está conectado a un reloj de tiempo real; en cambio, cuando sí se está conectado, se ejecuta tiempo real estricto.

3.2.4.2 Formato de Código en Tiempo Real.

Este formato resulta útil cuando se pretende realizar aplicaciones bajo el esquema de prototipado rápido. Si se desea generar código en tiempo real mientras se itera con los parámetros del modelo rápidamente, se debe comenzar el proceso de diseño con un objetivo *GRT*.

Este formato soporta estados y tiempo continuo, así como funciones *S*; además declara la memoria estáticamente, es decir, durante el proceso de compilación. En cambio, no soporta los bloques asociados a la llamada de funciones, al uso de tablas y a las funciones *S* definidas en *Matlab*, aunque sí permite que se usen si se definen desde *Simulink*.

3.2.4.3 Formato de Código en Tiempo Real de Localización Dinámica.

Presenta una estructura similar al caso anterior, salvo las siguientes diferencias:

- Se declara la memoria dinámicamente. Gracias a los bloques proporcionados por *MathWorks Inc*, las reservas dinámicas de memoria se limitan al código asociado a la construcción del modelo; por diseño, no existen memoria que no acabe siendo liberada al finalizar la ejecución del modelo.
- Permite la declaración de múltiples instancias del mismo modelo con su propia estructura de datos.
- Se pueden combinar varios modelos en un único archivo ejecutable. Por ejemplo, para integrar dos modelos, se mantiene una única instancia de cada uno de ellos, evitándose que se den colisiones en los nombres de los datos y de las funciones; siendo éstas posibles si se usa *RTW*.

Este formato no soporta una serie de bloques, al igual que en el apartado anterior.

3.2.4.4 Formato de Código para las Funciones *S*.

Este formato de código se adapta para la generación de funciones encapsuladas *API*; de esta forma, las funciones *S* pueden usarse como un bloque dentro de diferentes modelos. Para que se dé este formato se ha de forzar a que en el archivo para *TLC* la variable *CodeFormat* tome el valor *S-Function*, lo que se logra mediante la siguiente sentencia:

%assign CodeFormat = S-Function

Además, si de activa el modo de trabajo acelerado de *Simulink*, se define que el objetivo actual es un archivo *MEX*, el cual funciona como una librería de enlace dinámico.

3.2.4.5 Formato de Código para Aplicaciones Imbuidas.

Este formato se corresponde con el objetivo *Real Time Workshop Embedded Coder*; usándolo se incluyen métodos enfocados al ahorro de memoria y a la optimización de las capacidades de proceso. Véase el apartado siguiente para más información.

3.2.5 Sistemas Objetivo en Tiempo Real: Plantillas.

Para configurar o adaptar las plantillas se debe estar familiarizado con la forma en la que los comandos de construcción de código máquina funcionan. Las plantillas que aporta *Matlab* se construyen con enunciados mediante etiquetas o *tokens*. El proceso de construcción que desarrolla *RTW* las expande, sustituyéndolas por sus valores específicos de la configuración de trabajo, generando un fichero intermediario entre el código *C* y el compilador al código de nuestra máquina concreta: *modelo.mk*; este archivo, bajo terminología anglosajona, se denomina *makefile*. Las plantillas se diseñan con la intención de generar *makefiles* para compiladores específicos de plataformas concretas. El archivo *modelo.mk* está específicamente elaborado para compilar y enlazar el código generado a partir del modelo, utilizando órdenes concretas del entorno de desarrollo de trabajo.

Diagrama: Generación de *modelo.mk*.

Los comandos incluidos en el archivo punto *m make_rtw* (o comandos diferentes proporcionados con otros objetivos) dirigen el proceso de generación de *model.mk*. El comando *make_rtw* procesa la plantilla *makefile* especificada en la configuración del sistema objetivo; ésta se lleva a cabo a través del cuadro de diálogo *Simulation Parameters*. *make_rtw* copia la plantilla, línea a línea, expandiendo cada *token* o etiqueta. La lista siguiente muestra a cada uno de ellos con sus expansiones respectivas:

Token	**Expansión**
|>COMPUTER<|	Tipo de Computadora.
|>MAKEFILE_NAME<|	*Model.mk* – el nombre del *makefile* que fue creado a partir de la plantilla *makefile*.
|>MATLAB_ROOT<|	Ruta en la que *Matlab* está instalado.
|>MATLAB_BIN<|	Localización del ejecutable de *Matlab*.
|>MEM_ALLOC<|	O bien *RT_MALLOC* o bien *RT_STATIC*. Indica cómo se coloca la memoria.
|>MEXEXT<|	La extensión de los archivos *MEX*.

|>MODEL_NAME<|	Nombre del diagrama de bloques de *Simulink* en curso.
|>MODEL_MODULES<|	Cualquier módulo adicional de código fuente *c* que se haya generado. Por ejemplo, se puede separar un modelo grande en dos archivos, *model.c* y *model1.c*. En este caso, este *token* se expande como *model1.c*.
|>MODEL_MODULES_OBJ<|	Los nombres de los archivos objeto *obj* correspondientes con cualquier módulo fuente *c* generado.
|>MULTITASKING<|	Vale 1 si está activado el modo multitarea, 0 si no.
|>NUMST<|	Número de tiempos de muestreo en el modelo.
|>RELEASE_VERSION<|	La versión de realización de *Matlab*.
|>S_FUNCTIONS<|	Lista de los códigos fuente *c* de las funciones *S*.
|>S_FUNCTIONS_LIB<|	Lista de las librerías de funciones *S* disponible para enlazar.
|>SOLVER<|	El nombre del archivo que contiene el integrador, ej. *ode3.obj*.
|>BUILDARGS<|	Las opciones pasadas a *make_rtw*. Esta etiqueta se proporciona para que el contenido de *model.mk* cambie cuando se modifiquen los argumentos de construcción, así se fuerza la actualización de todos los módulos al cambiar las opciones de construcción.
|>EXT_MODE<|	Vale 1 para habilitar la generación del código de apoyo al modo externo, 0 en otro caso.

Estos *tokens* se expanden al sustituir los valores de los parámetros conocidos por el proceso de construcción. Además, *make_rtw* expande los que proceden de otras fuentes:

- Los *tokens* específicos al sistema objetivo definidos a través de la configuración del objetivo que se incluye en el cuadro de diálogo *Simulation Parameters*.

- Las estructuras de las opciones de *RTW* del archivo del sistema objetivo. Cualquier estructura definida en la matriz *rtwoptions structure* que contenga un campo *makevariable* será expandida.

El ejemplo siguiente está extraído de la ruta *...Matlab\rtw\c\grt\grt.tlc*. La sección que comienza con *BEGIN_RTW_OPTIONS* contiene el código *m* que inicia *rtwoptions*. Mientras que la directiva *rtwoptions(2).makevariable = 'EXT_MODE'* fuerza que la etiqueta |>*EXT_MODE*<| sea expandida como 1 (*on*) o como 0 (*off*), dependiendo de cómo se fije la opción *External Mode* en las opciones de generación de código.

Tras la generación de *modelo.mk RTW* invoca la orden de construcción del código máquina. En la ventana de comandos de *Matlab*, esta operación se puede llevar a cabo mediante:

makecommand –f modelo.mk

Esta orden se define por la macro *MAKE* de la plantilla apropiada para la aplicación de compilación.

Las plantillas poseen cierta estructura dividida en cuatro secciones:

- La primera contiene los comentarios iniciales que describen cuál es el sistema objetivo.
- La segunda define las macros que utiliza la aplicación de construcción –*make_rtw*– para procesar la plantilla. Las macros son:
  - *MAKE*: Ésta se usa para invocar a la aplicación de construcción. Ejemplo: si *MAKE = miConstructor*; entonces el comando llamado será *miConstructor –f modelo.mk*.
  - *HOST*: Indica cuál es la plataforma objetivo de la plantilla; se puede definir como *PC*, *UNIX*, *ANY* o cualquiera de las configuraciones compatibles. Mediante el comando computer en la ventana de comandos de *Matlab* se muestra las plataformas hardware que son compatibles.
  - *BUILD*: Indica si *make_rtw* debe llamar al compilador desde el proceso de construcción de *Real Time Workshop*.
  - *SYS_TARGET_FILE*: Se trata del nombre del archivo objetivo del sistema. Se usa para comprobar si existe en la sección de la configuración del objetivo de la lengüeta *Real Time Workshop* del cuadro de diálogo *Simulation Parameters*.
  - Un conjunto de macros que definen los mensajes mostrados por la consola de *Matlab*, las cuales son opcionales.
- La tercera sección define las etiquetas que sustituirá *make_rtw*.

- La cuarta contiene las reglas de construcción utilizadas en la elaboración de un ejecutable a partir del código fuente generado. Estas reglas son específicas para la aplicación de compilado usada.

3.2.6 Aplicaciones.

Las aplicaciones que en el mundo de la ingeniería del diseño tiene *Real Time Workshop* son muy numerosas, extendiéndose desde el mundo de la automoción hasta el ámbito aerospacial. Puesto que *RTW* admite la generación de código desde cualquier bloque descrito mediante *Simulink*, el número de aplicaciones y de ejemplos prácticos de éste crece considerablemente; si bien es cierto que *Real Time Workshop Embedded Coder* no resulta compatible con todos los bloques de *Simulink*, pero sí con los que se necesitan en las aplicaciones imbuidas, con lo que su abanico de aplicaciones también es amplio.

A modo de ejemplo se presentarán modelos concretos de posibles aplicaciones. Así pues, dentro del mundo de la automoción podríamos citar desde los sistemas de control de la inyección –con la coordinación temporal de los inyectores– hasta sistemas de climatización automática, pasando por la caja de cambios inteligente o el sistema de frenos *ABS*. De igual forma, dentro de la aviónica cabe señalar sistemas de interés como la anulación automática del desvío del cabeceo en los aviones comerciales, o sistemas de control de vuelo como el piloto automático o, en el caso espacial, sistemas de orientación y posición automática de satélites de comunicaciones.

Todos estos ejemplos vienen descritos, mediante *Simulink*, en las demostraciones de *Matlab*. Hay que considerar que ellos incluyen tanto la dinámica de la planta como la del controlador. En un caso real tan sólo sería necesario el diseño del sistema de control; claro que, como un paso previo a la implantación sobre una planta real del controlador, se requiere de la simulación de todo el sistema completo.

3.3 Real Time Workshop Embedded Coder.

En este apartado se busca presentar este paquete de desarrollo, pero plasmando las diferencias [B26] que ofrece respecto a *RTW*; las coincidencias ya han sido expuestas en secciones precedentes.

Este software, siendo un añadido a *RTW*, nos proporciona un marco de trabajo para el desarrollo de la generación de código que está optimizado para una mejor velocidad de cálculo

y para un menor uso de memoria; en definitiva, para aprovechar mejor los escasos recursos que posee un sistema embebido.

La optimización que ofrece el formato de código *Embedded-C* incluye:

- Una estructura específica de datos con restricciones de tiempo para el modelo que optimiza el uso de memoria. A veces, la estructura se retira completamente.
- Interfaz de encapsulado de objetos reducido, lo que permite incorporar al código generado líneas del lenguaje *C* directamente.
- Las funciones que generan la salida del modelo y la actuación de sus estados se han combinado en una única rutina.
- Las funciones *S* que son incluidas directamente ven reducida la cabecera de llamada y, por lo tanto, el tamaño del código.
- La declaración de memoria se realiza solamente de forma estática, lo que reduce las cabeceras y facilita un comportamiento determinista.
- La incompatibilidad con ciertos bloques constructivos de *Simulink*, como son los que trabajan únicamente con variables continuas –vía integración de las discretas–.

Real Time Workshop Embedded Coder abarca las siguientes capacidades:

- La generación automática de un ejemplo de un programa main, incluyendo comentarios.
- La generación automática de un organizador de tareas determinista, admitiendo varios tiempos de muestreo, dirigido a entornos monotarea y/o multitarea.
- La ejecución de modelos compatibles con el manejo de interrupciones asíncronas.
- Forzar la codificación exclusiva de las variables a números enteros.
- La generación de variables en punto flotante gracias a la compatibilidad con la llamada a las librerías correspondientes de *ANSI C* o *ISO C*.
- La validación del código generado desde *Simulink*.
- La generación de un informe detallado en formato hipertexto sobre el código generado, incluyendo enlaces activos a los segmentos de código del modelo. Este informe describe los módulos del código ayudando a identificar las optimizaciones relevantes llevadas a cabo sobre el programa.
- Las opciones de generación de código permiten optimizar la inicialización de los datos, facilitando la reducción en el uso de memoria *ROM*.
- La generación automática de datos bajo el formato *ASAP2*, el cual está muy extendido en los sistemas de calibración en la automoción comercial.

Real Time Workshop Embedded Coder genera un conjunto de ficheros de código (*c*) y de cabecera (*h*) que se coloca dentro de una carpeta que depende directamente del directorio de

trabajo; por defecto, su nombre es *nombreDelModelo_ert_rtw*. Este conjunto de archivos difiere ligeramente, pero significativamente, del que es generado cuando se trabaja bajo la configuración *GRT*.

Veamos, a modo de tabla, estos documentos:

- *model.c*: Contiene todos los puntos de entrada al código para realizar el algoritmo del modelo (*model_step*, *model_initialize*, *model_terminate*, *model_SetEventsForThisBaseStep*).
- *model_private.h*: Contiene las macros y los datos locales que requiere el modelo y los subsistemas. Este archivo se incluye automáticamente en la generación de código, siendo innecesaria su inclusión cuando se escribe código manualmente.
- *model.h*: Define las estructuras de datos y el interfaz público de los puntos entrada del modelo, así como para su estructura de datos en tiempo real. Si se quiere que el código manual, que se encuentra en otros módulos actúe con el modelo, debemos incluir *model.h* explícitamente.
- *ert_main.c*: Este archivo se genera opcionalmente, es decir, solamente si la opción *Generate an example main program* está activada.

Existen más archivos, pero los hemos omitido porque la información que aportan se ha considerado de menor relevancia.

Para generar el programa *main* se ha de habilitar la opción adecuada, dando lugar a la elaboración del archivo *ert_main.c*; para ello, en el cuadro de diálogo *Simulation Parameters: nombreModelo* se ha de marcar la casilla denominada *Generate an example main program*.

El cuadro de selección del sistema operativo de la máquina objetivo –*Target operating system*– permite elegir entre dos opciones, de las cuales nos interesa la que aparece en la figura anterior, es decir, *BareBoardExample.* Ésta está diseñada para sistemas que ejecutan el programa principal bajo el control de un reloj en tiempo real, pero sin un sistema operativo con primitivas críticas en tiempo.

ert_main.c incluye: La función *main()*, para la generación del programa, y el código del coordinador de tareas, que determina cómo y cuándo se ejecutan los cálculos de los bloques en cada iteración temporal del modelo. Estas partes dependen principalmente de si el modelo posee

un único tiempo de iteración o varios, asociados a los distintos bucles, y, también, del modo del integrador del modelo.

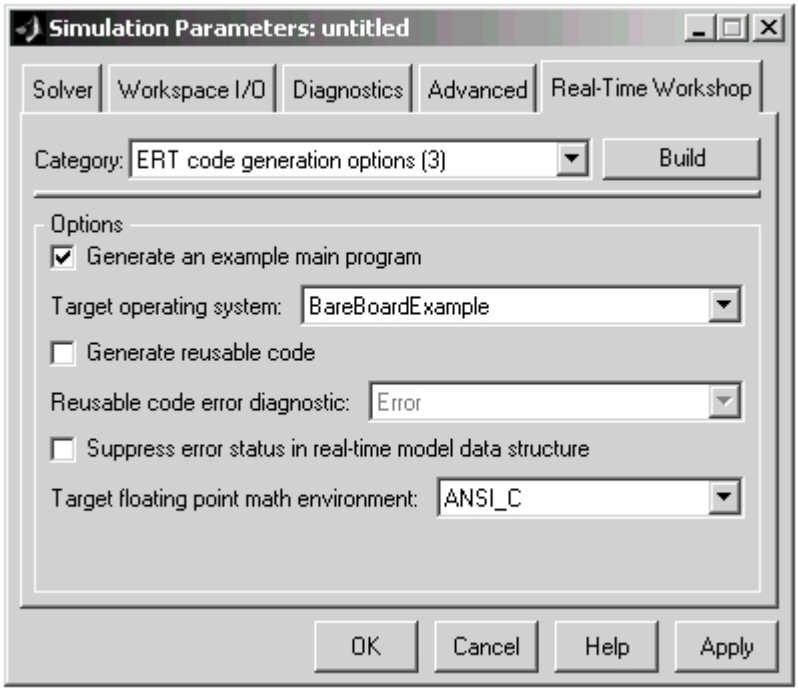


Figura: Opciones del Código Embebido.

Si la opción *Generate an example main program* no se selecciona, *Real Time Workshop Embedded Coder* proporciona el módulo *ert_main.c* como base para las modificaciones de usuario. Puesto que siempre se genera este fichero, pero con diferentes contenidos, hay que ser cuidadoso para evitar que las actualizaciones eliminen información de las versiones anteriores, que sí podrían tener activada dicha opción.

Puesto que se dispone de una placa base que no requiere de un sistema operativo ni de un generador de eventos en tiempo real externo, nuestra configuración se adapta a la opción *BareBoardExample*. En este caso la ejecución del modelo se centra en el lazo que realiza la función *main()*; está ejecuta la tarea de baja prioridad (*background*).

Esta tarea es periódicamente interrumpida por un temporizador, provocando el inicio de su rutina de servicio de interrupción (*ISR*), la cual *RTW* la denomina *rt_OneStep*, la cual

conlleva el peso de realizar una iteración del modelo llamando a la función *model_step*. Ante un caso monotarea, simplemente se ejecuta la única tarea de alta prioridad (*model_step*) o *foreground*; pero cuando se trata de una configuración multitarea, *rt_OneStep* asocia prioridades a los distintos bloques organizando su ejecución de acuerdo con los tiempos de sus respectivos bucles.

La opción *Generate reusable code* afecta a la definición de los puntos de entrada del código del modelo. En el caso de que esté activada, los parámetros se pasan por referencia, lo que reduce el uso de memoria. Si no, la definición de dichos puntos admite múltiples formas. Para facilitar el uso de estas funciones en módulos de usuario, hay que incluir *model.h* en ellos explícitamente; además, conviene examinar la forma del prototipo de estos puntos de entrada a la hora de invocarlos.

Veamos los distintos puntos de entrada:

- *model_step*.

  El prototipo de la función admite dos configuraciones:

  - Con un único tiempo de iteración: *void model_step(void);*
  - Con múltiples tiempos de iteración: *void model_step(int_T tid);* siendo *tid* el identificador de cada tarea dado por *rt_OneStep*.

  model_step combina el cálculo de las salidas del modelo así como la actualización de sus estados en una única rutina. En el caso de que se trabaje bajo una configuración de un único tiempo de iteración, esta función calcula los valores de las variables de todos los bloques, efectúa el registro de datos si está habilitado y, en el caso de se trate de un tiempo finito de operación, genera las señales adecuadas para detener la ejecución del modelo en el momento adecuado. La modificación que añade el uso de múltiples tiempos de iteración consiste en la gestión de las variables *tid*, manteniéndose la misma funcionalidad.

  Si el registro de datos no se ha habilitado o se trata de una aplicación de tiempo infinito o se necesita especificar una función específica de terminación entonces la función *model_step* se ejecuta indefinidamente.

- *model_initialize*.

  El prototipo que esta función presenta es: *void model_initialize(boolean_T firstTime)*.

  Si el argumento *firstTime* toma el valor *'1'* entonces la función inicia la estructura de datos en tiempo real del modelo y otras estructuras privadas; en cambio, si toma el valor *'0'*, inicia el valor de los estados del modelo sin intervenir en otras estructuras de datos. El código generado llama una vez a esta función con el argumento *firstTime* a *'1'*.

- *model_terminate*.

  El prototipo de esta función es: *void model_terminate(void)*.

  Al ser llamada, una sola vez en todo el código, los bloques que posean líneas de terminación las ejecutarán. Esta función se requiere cuando la aplicación trabaja con horizonte finito.

- *model_SetEventsForThisBaseStep*.

  Siendo su prototipo por defecto: *void model_SetEventsForThisBaseStep(boolean_T *eventFlags)*; donde *eventFlags* es un puntero al vector de las banderas de los eventos del modelo. En cambio, cuando la opción *Generate reusable code* está seleccionada, el prototipo de esta función se altera presentando la forma: *void model_SetEventsForThisBaseStep(boolean_T *eventFlags, RT_MODEL_model *model_M)*; donde *model_M* es un puntero a una instancia del objeto que representa el modelo en tiempo real.

  Esta función sólo se invoca cuando se trabaja con configuraciones multitarea y con varios tiempos de iteración, cuya funcionalidad radica en la gestión de las banderas de evento lo que determina qué tareas necesitan ejecutarse en función del tiempo base de iteración (el mínimo de todos ellos).

3.4 Construcción de una Nueva Plantilla.

Con el objetivo de trabajar con el compilador elegido en el capítulo dos, se hace necesaria la elaboración de una plantilla que haga de interfaz entre *Real Time Workshop Embedded Coder* y *MetroWerks CodeWarrior*. Para ello se investigó entre los documentos que incluye el sistema de ayuda de *Matlab*, en donde se hallaron dos artículos de interés. En el primero se explicaban, a través de un ejemplo sencillo, los pasos a seguir para construir la configuración de la nueva

interfaz; en el segundo, simplemente se mostraba la forma de proceder para que éste apareciera en la lista de plantillas disponibles.

3.4.1 Metodología.

Este punto se centra en la descripción de los dos documentos encontrados, así como de las cuestiones prácticas que fueron surgiendo durante el proceso de construcción de la nueva interfaz.

Para definir una configuración nueva para un sistema objetivo de trabajo se deben trazar estos pasos de forma consecutiva [B10]:

- Puesto que los archivos de código fuente que genera *RTW* dado un modelo expresado por *Simulink* se almacenan en una carpeta del directorio de trabajo de *Matlab*, se crea una carpeta de destino, que en nuestro caso concreto, tomará el nombre *d: \matlab\work\arm7tdmi*. En esta carpeta, además del código fuente, se guardan los archivos de las plantillas (*tmf*)[8] y los de configuración de la interfaz (*tlc*)[9]. Los comentarios incluidos al principio de estos archivos se usan como texto informativo en el cuadro de diálogo *System Target File Browser*, en el que se muestran las configuraciones de sistemas objetivo admisibles.

- Cuando a *Matlab* se le introduce una orden, éste busca en todas las rutas de directorio que tiene especificadas en una lista los archivos que pudiera necesitar la ejecución de dicha orden. Así pues, cuando se solicita la actuación de *RTW*, se debe incluir en esa lista la carpeta que será destino de los archivos fuente. Las configuraciones de los sistemas objetivos predefinidos ya aparecen en dicha lista tras la instalación de *Matlab*; teniendo que añadir explícitamente los casos definidos por el usuario. Con este fin se usa, en la pantalla de comandos, la orden *addpath('d: \matlab\work\arm7tdmi')*; existe también un método siguiendo menús y cuadros de diálogo, pero creo que la forma presentada es más rápida.

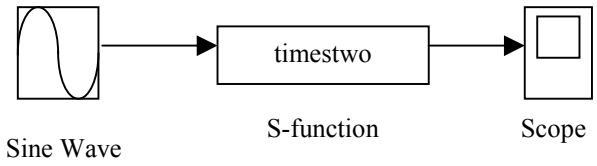
- La ruta de directorio de destino es siempre el directorio de trabajo, por definición dada por *Matlab*. Luego hemos de indicar que *d: \matlab\work\arm7tdmi* es nuestro directorio de trabajo; para lograr esto, en el cuadro de dirección que aparece en la parte superior de la ventana de *Matlab*, introducimos la ruta deseada.

[8] *.tmf*: *template makefile*.
[9] *.tlc*: *target language compiler*.

- En el ejemplo introductorio que ofrece el sistema de ayuda se hace uso de una función *S* dentro del modelo de *Simulink*, la cual está definida a través de un fichero fuente en *C*. Para que dicha función pueda ser utilizada debe incluirse en el directorio de destino el fichero fuente correspondiente. Con esta intención, copiamos *timestwo.c* desde *d: \matlab \toolbox\ rtw\ rtwdemos\ tlctutorial \timestwo* \ en la carpeta *d: \matlab \work \arm7tdmi* \.

- Para cada función *S* que se use se requiere el archivo de interfaz *mex* asociado; para ello se usa la orden *mex timestwo.c*. Entre las opciones elegimos el compilador que *Matlab* proporciona. Tras finalizar el proceso, aparece un archivo de enlace a librería dinámica de nombre *timestwo.dll* dentro del directorio de trabajo.

- Se hace necesaria la creación de un modelo bajo *Simulink* para generar el código correspondiente. El ejemplo nos muestra el diagrama que debemos definir:



  - Sin más que hacer doble clic en el bloque que define la secuencia sinusoidal y rellenando los campos, se define su amplitud en 1.
  - Hay que definir el bloque representativo de la función *S*; para lo cual hacemos doble clic en él e introducimos en el cuadro de diálogo que aparece el nombre del archivo fuente asociado sin su extensión (las funciones *S* sólo admiten código fuente descrito en *C++*).
  - Asimismo, el integrador debe cumplir ciertas condiciones para que sea admisible por *RTW*, esto es: se escoge un intervalo de iteración fijo *–fixed_type–* y el método integración de *Runge-Kutta*.
  - Tras la definición de los bloques y de sus interconexiones se guarda el modelo con el nombre *d:\matlab\work\arm7tdmi\timestwo.mdl*.

- Iniciamos la simulación dentro de la ventana del modelo de *Simulink* que acabamos de generar, pero se nos muestra un mensaje de error que nos informa que el modelo y la función *S* no pueden tener el mismo nombre (*timestwo*). Para solucionar esto nombramos al modelo *senoPor2.mdl*, guardándolo en la misma ruta de directorio.

- Iniciamos la simulación nuevamente comprobándose, gracias al bloque *Scope*, que la secuencia de salida tiene efectivamente el doble de amplitud que la de la secuencia de entrada.

- Se requiere, por cada archivo asociado a una función *S*, un archivo homónimo cuya extensión sea *tlc* para que el código de dicha función quede incluido en los archivos generados por *RTW*; en caso contrario, se generaría una llamada a la función externa definida por *timestwo.c*, con lo que se consumirían más recursos de tiempo y memoria del sistema objetivo. Por lo tanto, copiamos el archivo *timestwo.tlc* en *d: \matlab \work \arm7tdmi* desde la ruta de directorio *d: \matlab6pd\toolbox\rtw\rtwdemos\tlctutorial\timestwo*.

- Se copian los archivos *grt_main.c* y *grt.tlc* desde la ruta de directorio *d: \matlab\rtw\c\grt* a la carpeta *d:\matlab\work\arm7tdmi* cambiando sus nombres por *arm7tdmi_grt_main.c* y *arm7tdmi_ert_main.c*, respectivamente. El primero de estos archivos forma el esqueleto de un programa *main* de ejemplo, siendo el segundo su correspondiente archivo *tlc*. Se actúa de forma análoga con los archivos *ert_main.c* y *ert.tlc*, siendo estos los correspondientes al modo de estilo de código embebido o *Embedded-C*.

- El código fuente generado no se almacena directamente en el directorio de trabajo, sino en una carpeta de éste cuyo nombre está indicado en el interior del archivo *tlc* correspondiente. Para ello, al final del documento, se asocia el valor *'arm7tdmi_rtw'* al sufijo del nombre de dicha carpeta –el prefijo lo coloca automáticamente *Matlab* a partir del nombre del modelo *Simulink*–. Esta asignación queda reflejada en el interior del archivo *grt.tlc* mediante la siguiente instrucción: *rtwgensettings.BuildDirSuffix = 'arm7tdmi_rtw'*; de forma análoga cambiamos este campo del fichero *ert.tlc* por el valor *'arm7tdmi_ert'*. Así, el campo *BuildDirSuffix* del objeto *rtwgensettings* toma el valor deseado.

- En la ruta de directorio *d:\matlab\rtw\c\grt Matlab* contiene varias plantillas específicas para diferentes compiladores, tanto para un objetivo en tiempo real genérico como para el caso de un sistema embebido.
  - Copiamos las plantillas *grt_lcc.tmf* y *ert_lcc.tmf*; que están definidas para el compilador que proporciona *Matlab*, es decir, el compilador *LCC*; en la carpeta *d:\matlab\work\arm7tdmi*.
  - Les cambiamos el nombre por *arm7tdmi_grt.tmf* y *arm7tdmi_ert.tmf*, respectivamente.
- A la hora de adaptar una plantilla a un compilador concreto se debe hacer uso de la documentación que el fabricante del mismo proporciona. Para el caso del compilador *LCC*, los cambios a realizar sobre ellas consisten en actualizar las siguientes etiquetas por estos valores:

- Cambios en *arm7tdmi_grt.tmf*:
  - SYS_TARGET_FILE = arm7tdmi_grt.tlc.
  - REQ_SRCS = $(MODEL).C $(MODULES) arm7tmdi_grt_main.c ...
  - %.obj : $(MATLAB_ROOT)/rtw/c/grt/%.c por
  - %.obj : $(MATLAB_ROOT)/work/arm7tdmi/%.c
- Cambios en *arm7tdmi_ert.tmf*:
  - SYS_TARGET_FILE = arm7tdmi_ert.tlc.
  - REQ_SRCS = $(MODEL).C $(MODULES) arm7tmdi_ert_main.c ...
  - %.obj : $(MATLAB_ROOT)/rtw/c/ert/%.c por
  - %.obj : $(MATLAB_ROOT)/work/arm7tdmi/%.c

Nota: los cambios siguientes son sólo aplicables al compilador *LCC*. Si se usa otro compilador, las reglas para la fuente (*REQ_SRCS*) y los archivos objeto (*%.obj :*) pueden diferir ligeramente.

- Se puede comprobar, tras pulsar el botón *Browse...* del cuadro de diálogo *Simulation Parameters: senoPor2*, que aparecen cuatro configuraciones nuevas de sistemas objetivo:
  - arm7tdmi_ert.tlc: RTW Embedded Coder.
  - arm7tdmi_ert.tlc: Visual C/C++ Project Makefile only for the RTW Embedded Coder.
  - arm7tdmi_grt.tlc: Generic Real-Time Target.
  - arm7tdmi_grt.tlc: Visual C/C++ Project Makefile only for the grt target.
- Como opciones de configuración de *RTW* se ha tomado:
  - System Target File: arm7tdmi_grt.tlc o arm7tdmi_ert.tlc.
  - Template Makefile: arm7tdmi_grt.tmf o arm7tdmi_ert.tmf.
  - Make Command: make_rtw.

Nota: la primera opción se aplica cuando se usa el estilo de código *GRT* o tiempo real genérico; en cambio, la otra opción sólo se utiliza para el caso del estilo *ERT* o tiempo real embebido.

- Iniciamos el proceso de generación de código sin más que pulsar el botón *Build* del cuadro de opciones de *RTW*. En el caso de seleccionar la primera opción, se obtiene el conjunto de ficheros fuente *C++* dentro de la ruta de directorio especificada (*d: \matlab \work \arm7tdmi \senoPor2_grt_rtw*); en cambio, con la segunda opción se incurre en error puesto que la función *S timestwo* trabaja con señales continuas, siendo un formato no admitido por el modo embebido.

- Editamos el archivo fuente que contiene la función *main*, es decir, el que se llama *senoPor2.c*, el cual se encuentra dentro de la ruta de directorio apuntada por *...\arm7tdmi\senoPor2_arm7tdmi_grt_rtw*, con la intención de localizar la función *MdlOutputs*. En ella se observa el código añadido por la función *S* es:

```
/* S-function block: <Root>/S-funcition*/
/* Multiply input by two */
rtB.S_function = rtB.Sine_Wave*2.0;
```

  Es decir: el bloque *Sine_Wave* ve multiplicada su salida por 2.0 y se asigna como salida del bloque *S_function*. El objeto *rtB* contiene todos los bloques que componen el modelo de *Simulink*.

- Para que las plantillas que acabamos de construir [B11] aparezcan en el cuadro de diálogo que muestra las configuraciones del sistema objetivo que están admitidas, tan sólo hay que añadir la ruta de directorio de la carpeta que las contiene al conjunto que comprueba *Matlab* cada vez que necesita un archivo. Para ello debemos, en la ventana de comando, teclear la orden: *addpath('nombre_carpeta')* o bien seguir las indicaciones de campo *Set Path...* del menú *File*.

  - Esta carpeta debe contener una serie de archivos para evitar que se den errores en la búsqueda de las plantillas; estos son:
    - Archivo del sistema objetivo cuya extensión será *tlc*
    - Archivo *make* de plantilla de extensión *tmf*.
    - Archivos de la interfaz en tiempo de ejecución: los ficheros de código fuente que contienen el programa principal y las funciones *S*.

  Nota: La inclusión de la carpeta que los contiene no permanece definida para posteriores sesiones de trabajo bajo *Matlab*. Si pretendemos tener que evitar su inclusión cada vez que vayamos a trabajar con nuestro sistema objetivo, tan sólo hemos de incluir la línea de código *Matlab addpath('d: \matlab6pd\work\arm7tmdi\');* en el fichero de arranque *setup.m*.

  - Existe, dentro del contenido de los archivo de configuración del sistema objetivo (*tlc*), una línea que tiene una función de interés en este punto. Se trata de una línea de comentario que aparece al principio del archivo y que está precedida por la etiqueta de control *SYSTLC*. La cadena de texto que a continuación se escriba será la utilizada por el cuadro de diálogo *System File Target Browser*. Por lo tanto, para las configuraciones

GRT y *ERT* que estamos construyendo para *ARM7TDMI*, llevan las siguientes líneas de texto:

- En *arm7tdmi_ert.tlc*: *%SYSTLC: RTW Embedded Coder for ATMEL ARM7TDMI*.
- En *arm7tdmi_grt.tlc*: *%SYSTLC: Generic Real-Time Target for ATMEL ARM7TDMI*.

- Ahora, podemos comprobar que en dicho cuadro de diálogo aparecen las configuraciones:
  - *arm7tdmi_ert.tlc*: *RTW Embedded Coder for ATMEL ARM7TDMI*
  - *arm7tdmi_grt.tlc*: *Generic Real-Time Target for ATMEL ARM7TDMI*

Debido a las diferentes pruebas que se realizaron hasta lograr una configuración consistente, el cuadro de en cuestión se mostraba de esta forma:

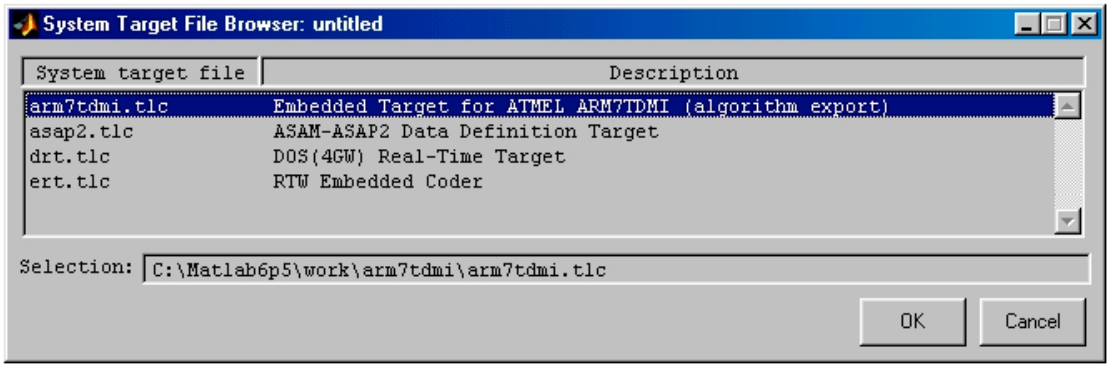


Figura: Sistema Objetivo de *ARM* en el *System Target File Browser*.

Nota: Esta imagen ha sido tratada por computadora, eliminándose de ella las configuraciones innecesarias, para lograr un formato reducido sin perder información relevante. Nótese que en la línea inicial de la plantilla *arm7tdmi.tlc* se incluye: *%% SYSTLC: Embedded Target for ATMEL ARM7TDMI (Algorithm Export)*.

3.4.2 Compiladores que soporta inicialmente Matlab.

El equipo de trabajo de *MathWorks Inc* ha probado, con éxito sobre la plataforma *Windows*, la compatibilidad entre *Real Time Workshop* y diversos compiladores comerciales [B9]; mediante una tabla resumiremos sus resultados:

Compilador	Versiones
Borland	5.2, 5.3, 5.4, 5.5, 5.6
LCC	La versión incluida en *Matlab*
Microsoft Visual C/C++	5.0, 6.0, 7.0
Watcom	10.6, 11.0

Para comprobar qué nuevas versiones se han probado hasta la fecha, resulta conveniente consultar la dirección en Internet de *MathWorks Inc*, es decir, [B37] www.mathworks.com, más concretamente la nota técnica contenida en http://www.mathworks.com/support/tech-notes/1600/1601.shtml.

El compilador que nosotros usaremos –*MetroWerks CodeWarrior*– no aparece en esta lista, lo que implica que no ha sido comprobada su compatibilidad con *RTW*, no que no vaya a ser compatible.

3.5 Estudio de las Plantillas para mpc555dk.

mpc555dk forman las siglas identificativas de un producto comercial de la marca *Motorola*, siendo éste un fabricante de tecnología integrada con amplia cuota de mercado de ámbito internacional. *mpc555dk* consiste en un *kit* de desarrollo para el chip *mpc555*. El equipo de *RTW* ha elaborado un añadido [B36] que proporciona la funcionalidad de construir código fuente embebido para esta máquina desde el entorno *Simulink*, de esta forma se puede aplicar las técnicas de trabajo que ofrece el prototipado rápido.

El interés que muestra este esquema no radica en la posibilidad de utilizar el kit de *Motorola*, puesto que nuestro sistema de desarrollo se basa en la tecnología de *ATMEL*; pero sí se apoya en las similitudes existentes en el planteamiento y en la compatibilidad de *mpc555dk* con el compilador *MetroWerks CodeWarrior* desde *RTW*. Esta posibilidad se plantea, gracias a la documentación que *Matlab* proporciona, en la que menciona como compiladores cruzados

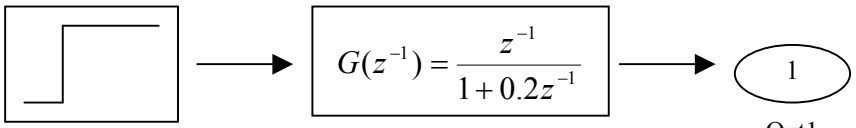
compatibles a *MetroWerks CodeWarrior* y a *Diab Data Power PC 4.3g*, ambos productos comerciales.

Considerando que la compatibilidad entre *RTW* y el compilador *CodeWarrior* no está probada por el equipo de trabajo de *MathWorks Inc*, junto a la funcionalidad del *kit mpc555dk*, nos vemos obligados a estudiar el esquema planteado en el conjunto de plantillas, de ficheros con código fuente de *Matlab* y de librerías de enlace dinámico que logran la comunicación entre *RTW* y *mpc555*. De esta forma podremos o bien elaborar un esquema similar para la placa de *ATMEL*, si no es excesivamente complejo, o bien adaptar el planteamiento existente a nuestro equipo de trabajo.

3.5.1 Compatibilidad del Código generado por RTW con CodeWarrior.

Previamente al estudio de la estructura y dependencia de los archivos que constituyen la interfaz entre *RTW* y *mpc555dk* se ha de comprobar que el código fuente en lenguaje *C++* es compatible con el compilador *CodeWarrior*. Para asegurar esto se ha ideado un modelo de prueba, descrito mediante un diagrama de bloques en *Simulink*, cuyo código fuente asociado se incluye en un proyecto nuevo dentro de ese compilador.

El diagrama de bloques que describe el modelo de prueba es el siguiente:



Siendo:

- El bloque de excitación genera una secuencia *u(n)* que constituye un escalón unitario.
- El bloque intermedio modela el comportamiento de un sistema de ejemplo, siendo estable.
- El bloque de salida presente la funcionalidad de enviar al espacio de trabajo de *Matlab* los resultados de la operación.

La plantilla que escogimos para definir el sistema objetivo correspondía al caso *GRT*, pero el integrador se tomó de tiempo de intervalo fijo debido a las características restrictivas de un sistema embebido. Tras la invocación a *RTW* se generó un grupo de archivos dentro del directorio de destino, los cuales contenían el código fuente en lenguaje *C++* asociado al diagrama.

Estos ficheros eran:

- untitled_data.c
- untitled.c
- untitled_types.h
- untitled_private.h
- rtmodel.h
- untitled.h

Siendo el directorio de destino: *d: \matlab \work \untitled_grt_rtw*; el cual depende inmediatamente del directorio de trabajo. Nótese que el nombre del modelo *Simulink* es, en este caso, *untitled.mdl*[10]; el cual aparece tanto en el nombre de la carpeta destinataria del código fuente como en él de los archivos generados.

Abrimos un nuevo proyecto bajo *CodeWarrior*, el cual se situó en la ruta de directorio*: ...\Mis Documentos \Códigos Fuente \RTW \untitled* con el nombre *untitled.mcp* del *PC* del laboratorio. En él incluimos todos y cada uno de los ficheros que *RTW* había generado para procesarlos, posteriormente, uno a uno. Se produjeron tres errores entre los archivos de código fuente *C++* porque en ellos se hacía referencia a ciertas librerías que no aparecían en la carpeta de destino de estos ficheros. Las librerías solicitadas son:

- d:\matlab\include\bin\tmwtypes.h
- d:\matlab\simulink\c\simstruct_types.h
- d:\matlab\rtw\include\c\rt_logging.h

Este error era de esperar porque estas tres librerías daban lugar al conjunto de dependencia estático que aparecía en el informe –en formato *hipertexto*[11]– de la generación de código fuente, el cual genera *RTW* como opción. Para solventar este problema incluimos, explícitamente, cada una de estas librerías en el proyecto de prueba de *CodeWarrior*.

Volvemos a procesar uno a uno los ficheros de este proyecto sin incurrir en ningún error o aviso, lo que nos impulsa a compilar y enlazar todos los archivos del mismo en su conjunto. De esta acción se generaron, según el informe que presentó *CodeWarrior*, 1244 líneas de código junto con 568 líneas de datos; todas ellas asociadas al fichero *untitled.c*, el cual contiene el modelo del diagrama expresado en lenguaje *C++*. Además, en la ventana de mensajes del compilador, se mostró un único aviso –lo que no bloquea el proceso de construcción del código máquina– y ningún error.

[10] *Untitled* significa *sin título*; el modelo no tenía nombre definido cuando se generó el código.
[11] Cuyas extensiones asociadas son: *htm*, bajo *Windows*, o *html*, bajo *UNIX*.

Ya tenemos código máquina, pero hemos de depurarlo en el sistema real. Para ello conectamos el sistema de desarrollo de *ATMEL* al *PC* e iniciamos el depurador *AXD Debugger*. Éste detecta la placa de trabajo satisfactoriamente; pero surge un problema en cuanto solicitamos la ejecución del depurador sobre el sistema tras la descarga del código en el sistema embebido mostrándosenos el siguiente mensaje: *Processor ARM_1 raised an exception. Cause: the processor was reset*; lo que significa que el procesador ha alcanzado una excepción porque ha sido reiniciado. Esto bloquea cualquier ejecución de código.

Como solución se plantea la elección de otra plantilla de trabajo, así pues, en lugar de usar el estilo de código *GRT* seleccionamos el caso *ERT*, más apropiado para sistemas objetivo embebidos. Para ellos optamos por la plantilla *ert.tlc* localizada en el directorio *d:\Matlab\rtw\c\ert* dentro del cuadro de diálogo *Target File System Browser* de *Matlab*. Para concretar más la generación de código, actuamos sobre las opciones que nos brinda *RTW* forzando las siguientes condiciones:

- Se toma *ANSI_C* como estándar para la codificación numérica flotante en lugar de *ISO_C*, por compatibilidad con el kit de desarrollo facilitado por *ATMEL*.
- Se solicita la generación del informe en *HTML*, así observaremos más fácilmente cuáles son los archivos generados y las dependencias estáticas existentes.
- El sistema objetivo o *target*, en inglés, se define como *BareBoardExample*, siendo una posible traducción libre: ejemplo para placa base.

Estas opciones se encuentran en las categorías *ERT code generation options (1)* y siguientes.

De acuerdo con el informe *html* se han generado los archivos de código fuente, dentro de la ruta *d: \Matlab \work \untitled_ert_rtw*, que a continuación se indican:

- ert_main.c
- untitled.c
- untitled_data.c
- untitled.h
- untitled_private.h
- untitled_types.h

Estableciéndose la dependencia estática con las siguientes librerías de *Matlab*:

- d:\matlab\extern\include\tmwtypes.h
- d:\matlab\simulink\include\simstruc_types.h
- d:\matlab\rtw\c\libsrc\rt_logging.h

Definimos un nuevo proyecto en *CodeWarrior* cuya ruta de directorio es *...\Mis Documentos \Códigos Fuente \rtw\ertPrueba1*. En él incluimos todos los archivos generados por *Real Time Workshop Embedded Coder* y las librerías con las que existe dependencia estática. Al intentar la elaboración del código máquina aparecen 3 errores, todos ellos asociados a símbolos que no se han definido correctamente; estos son:

- *rt_UpdateTXYLogVars*
- *rt_StartDataLogging*
- *rt_StopDataLogging*

Todos estos símbolos hacen referencia al proceso de registro de datos al tiempo que se efectúa la ejecución del código del modelo. El primero de ellos se refiere al registro de los datos de tiempo y de los valores de los estados y de las secuencias para cada instante; el segundo, a los datos iniciales y el último, a los finales.

La aparición de estos errores resulta coherente con las capacidades de nuestro sistema objetivo embebido, ya que éste no dispone de un dispositivo de almacenamiento para efectuar los registros. Para solucionar este problema se recurre a la inhabilitación de la opción consistente en la generación de un historial, con el formato de un archivo de *Matlab*[12], de la evolución de las variables durante la simulación. Nuevamente, ejecutamos *RTW* para obtener el código fuente con las nuevas opciones, almacenándose los archivos correspondientes en la carpeta: *d:\Matlab\work\ertPrueba2_ert_rtw*; nótese que el nombre del diagrama *Simulink* se ha definido como *ertPrueba2*, tal y como se muestra en el prefijo del nombre del directorio.

Se construye un nuevo proyecto con el compilador *CodeWarrior* cuyo nombre es *ertPrueba2.mcp*; en él se incluyen todos los archivos necesarios para que el proceso de compilado y enlace se realice satisfactoriamente –archivos fuente, librerías y dependencias estáticas–. Iniciamos la descarga a través del depurador asegurándonos que, previamente, el modo de operación de la placa sea de usuario; una vez realizado esto, se ejecuta el programa descargado mostrándose el mensaje de que dicho código trabaja con horizonte infinito, es decir, indefinidamente, a través de la consola que proporciona *AXD Debugger*.

Para poder comunicarnos con las variables que aparecen en el desarrollo del código planteamos tres posibles vías. La primera de ellas se basa en el uso del modo externo que facilita *Simulink*; la segunda exige que la estructura de plantillas entre *RTW* y *MetroWerks CodeWarrior* esté correctamente definida y, la tercera se fundamenta en una comunicación

[12] Estos archivos poseen la extensión *mat*.

manual, tal y como se ha llevado a cabo en este ejemplo de prueba. Hasta este momento, la única vía factible es la tercera.

Desde *Matlab* abrimos el panel que controla el modo externo, es decir, el panel cuyo nombre es *External Interface Control Panel*. Pero, de acuerdo con los mensajes que se nos muestran, este modo de funcionamiento no es compatible con el sistema objetivo concreto con el que estamos trabajando; es posible que la causa de este error nazca de que el archivo que realiza las funciones de interfaz (*C-Mex*)[13] no esté correctamente definido.

Por lo tanto, la vía que podemos usar pasa por la actuación manual incluyendo en un proyecto nuevo de *CodeWarrior* los archivos generados por *Real Time Workshop Embedded Coder* Este proceso se pretende automatizar con el estudio expuesto en los puntos siguientes.

3.5.2 Estructura e interdependencia de las Plantillas.

Estas plantillas se encuentran todas ellas dentro de la carpeta cuya ruta de directorio se expresa como: *d:\Matlab\toolbox\rtw\targets\mpc555dk\mpc555dk*; a parte de en las carpetas que de ella dependen. Apuntaremos aquí las parejas plantilla–librería que más interés contengan para nuestro estudio:

- *mpc555_settings.tlc*: Esta librería define la configuración para el objetivo *MPC555*. Contiene multitud de etiquetas relativas a la localización de los archivos ejecutables del compilador y del depurador concretos que se estén utilizando, así como otras asociadas al puerto de comunicaciones, tanto a su identificación como a su funcionamiento. Muchas de estas etiquetas aparecen definidas como una cadena vacía de caracteres; de esta forma se pueden indicar las rutas de directorio –cadenas, al fin y al cabo– para las etiquetas, bien manualmente o bien automáticamente a través de funciones de *Matlab*, tal y como se verá en el punto siguiente.

- *mcp555rt.tlc* y *mpc555rt.tmf*: Es la plantilla utilizada por el modo de funcionamiento en tiempo real, el cual exige la existencia de un sistema operativo en tiempo real mediante primitivas sobre el sistema objetivo de trabajo. Este no es nuestro caso.
  - Incluye las librerías *mpc555_settings.tlc* y *mpc555rt_genfiles.tlc*.
  - Define la etiqueta *TargetCompiler* con la cadena *diab*; de esta forma se indica que el modo de funcionamiento en tiempo real sólo puede llevarse a cabo con el compilador *Diab Data Power PC 4.3g*.

[13] Los archivos de interfaz en *Matlab* se denominan *C-MEX*, siendo su extensión: *dll*.

- Define las opciones del compilador entre las sentencias *Begin_RTW_Options* y *End_RTW_Options*. Éstas aparecen en el panel de opciones de *Real Time Workshop*; estando algunas definidas, otras no.

- *mpc555pil.tlc* y *mpc555pil.tmf*: Esta plantilla se necesita cuando se usa el modo de funcionamiento denominado *PIL*[14]; el cual posibilita el modo externo, según la documentación. Incluye las librerías *mpc555_settings.tlc* y *ert_pil_lib.tlc*, y, para importar la configuración global, *mpc555pil_genfiles.tlc*.

- *mpc555_ext.tlc* y *mpc555_ext.tmf*: Se usa para el modo de funcionamiento que se llama *Algorithm Export*, el cual está diseñado para construir una aplicación autónoma sobre *PC* mediante el generador de código *C++* y el compilador cruzado *Diab Data Power PC 4.3g*.

3.5.3 Definición de las Preferencias del Objetivo.

Con el objetivo de definir las preferencias de la interfaz acorde a las características concretas del sistema objetivo se llevó a cabo una serie de estudios [B34] que desembocaron en el planteamiento expuesto en el punto *3.6 Estudio del Funcionamiento de la Interfaz "MPC555DK"*. En el apartado actual explicaremos los estudios que a aquél nos llevaron, para lo cual se construyó un ejemplo muy sencillo de diagrama de bloques sobre el que aplicar los algoritmos de *RTW*.

Se observa que, dentro de las librerías *tlc* propias a cada uno de los modos de funcionamiento que soporta *mpc555dk*, aparece la definición de la estructura de datos que describe los parámetros de la interfaz. Presentamos, resumidamente, los campos de interés:

- *CompiledModel*: Éste el nombre de la estructura completa.
  - *Settings*: Siendo el campo que fija la configuración de la interfaz, dentro del cual se definen otros parámetros de nivel inferior.
  - *TargetCompiler*: Se trata de la etiqueta que discrimina entre el compilador cruzado *CodeWarrior* o el *Diab*. Ésta se usará por funciones para *Matlab* contenidas en el subdirectorio de *mpc555dk*.
  - *TargetCompilerVersion*: Indica la versión del compilador.
  - *TargetCompilerPath*: Admite una cadena de caracteres que apunte al archivo ejecutable que forma el compilador.
  - *BSP_Target*: Indica la ruta de directorio que contiene los archivos adecuados para cada forma de trabajo que facilita *mpc555dk*.

[14] *PIL*: acronismo en inglés de *processor-in-the-loop* o *procesador en el lazo*.

- *COM_BAUD*: Define el número de baudios al cual trabajará el puerto de comunicaciones, solamente admite un conjunto finito y dado de valores, siendo los mismos que están predefinidos en el funcionamiento del puerto.
- *CommTimeout*: Define el tiempo máximo de espera de respuesta antes de la generación del evento de comunicación interrumpida.
- *COM_PORT*: Mediante una cadena de caracteres indicamos cuál será el puerto de comunicaciones del equipo anfitrión –*host*– que se conectará con el sistema *objetivo.*
- *CommTargetPort*: De forma análoga, pero refiriéndonos al equipo objetivo.
- Otros.

- *PathInfo*: Se trata de un campo del objeto *CompiledModel* que contiene, entre otros, los siguientes campos.
  - *MODEL_FILE*: Indica el nombre del modelo.
  - *MODEL_REL_PATH*: Indica la ruta de directorio relativa del modelo.
  - Otros.
- *PlugIns*: Contiene información sobre las librerías y los añadidos que forman parte del *mpc555dk*.
  - *MPC_555DK_PLUGIN_INCLUDES*
  - *MPC_PLUGIN_LIBS*

Además de la definición de la estructura de los campos, se asigna una serie de valores concretos a cada uno de ellos según la configuración de cada modo dentro de los archivos correspondientes *tlc*Para ello se hace uso de la sentencia:

% assign nombreDeEtiqueta = Valor.

Que, concretándola para cada campo, se transforma en:

```
% assign TargetOS = BareBoardExample
...
%%import global settings:
% include arm7tdmi_settings.tlc
%% must be CodeWarrior for ARM Target at this time
% assign CompiledModel.Settings.TargetCompiler = CodeWarrior.
...
% assign CompiledModel.Settings.TargetCompiler = CodeWarrior
% assign CompiledModel.Settings.TargetCompilerPath = d:\sw_fcs\armm\adsv1_2\bin\ide.exe
% assign CompiledModel.Settings.BSP_Target = DebugRel
% assign CompiledModel.Settings.COM_BAUD = 38400
% assign CompiledModel.Settings.CommTimeout = 4
% assign CompiledModel.Settings.COM_PORT = COM2
% assign CompiledModel.Settings.CommTargetPort = COM2
```

Consideraciones:

- Antes de guardar las modificaciones, el archivo original se salvaguarda con extensión *bak* manteniéndose el mismo nombre.
- El archivo de librería *arm7tdmi_settings.tlc* contiene la misma información que el original *mpc555dk_settings.tlc*, salvo que se ha guardado con otro nombre.
- El campo *BSP_Target* presentaba, inicialmente, el valor *phyCore-555*, que es el compilador interno apropiado para la máquina de *Motorola mpc555*.

Seleccionamos la plantilla *armt7tdmi_ert.tlc* para compilar el ejemplo de *Simulink* mediante *RTW*, lo cual nos muestra un error debido a que cierta etiqueta no está definida en los archivos que forman la interfaz; ésta es:

- En la línea 120 de *arm7tdmi_settings.tlc*: *PIL_TLC_DIR*.

Para solventarlo estudiamos las referencias a de este campo previas a la línea 120, aparece la siguiente:

*% assign CompiledModel.PathInfo.**PIL_TLC_DIR** =*
%<TARGET_ROOT>%<PATH_SEP>pil

La cual hace uso de las etiquetas *TARGET_ROOT* y *PATH_SEP*. La primera de ellas también tiene referencias previas, es decir:

*% assign CompiledModel.PathInfo.**TARGET_ROOT** = FEVAL(filepart,%<TLC_DIR>,path)*

En donde aparece la etiqueta *TLC_DIR*, la cual también se menciona anteriormente en:

*% assign CompiledModel.PathInfo.**TLC_DIR** = FEVAL(filepart,%<TLC_FILE>,path)*

La etiqueta *TLC_FILE* se define con anterioridad en las líneas de código mediante la asignación siguiente:

*% assign CompiledModel.PathInfo.**TLC_FILE** = FEVAL(strtokn,%<TLCFILES>,[,],2)*

Pero la etiqueta *TLCFILES* no posee referencia anterior en la librería, lo que nos impulsa a afirmar que el origen del error por *PIL_TLC_DIR* se encuentra en *TLCFILES*. Para buscar más información sobre esta etiqueta indagamos en el sistema de ayuda que proporciona *Matlab*.

Este sistema da lugar a la entrada *TLC Error Messages*[15] contenida en el documento denominado *TLC Error Handling*[16]; en ella se define el funcionamiento de la función *FEVAL*[17], pero no ofrece una solución efectiva sobre el significado de la etiqueta *TLCFILES*.

[15] Cuya traducción al español quedaría como: *Mensajes de Error TLC*.
[16] Cuya traducción al español quedaría como: *Manejo de Errores TLC*
[17] Cuyo cometido es evaluar la función dada como parámetro junto con sus argumentos.

Como posible solución se plantea definir dicha etiqueta previamente a que sea usada, para ello añadimos esta sentencia justo antes de la última referencia que se ha señalado:

*% assign CompiledModel.PathInfo.**TLCFILES** = arm7tdmi_ert*

Una vez arreglado este error volvemos a solicitar la generación de código desde RTW, pero aparece otro error, también por falta de definición, es decir:

- En la línea 122 las etiquetas *PATH_SEP* y *PIL_TLC_DIR* no están definidas.

De acuerdo con la ayuda de *Matlab*, la primera etiqueta constituye una palabra reservada definida de antemano; cuyo cometido consiste en indicar cuál es el carácter de texto que usa el sistema operativo para separar un nivel de otro en la jerarquía del sistema de directorios. Así pues, cuando se trabaja con *Windows* se usa el carácter '\', pero cuando el sistema se basa en la tecnología *UNIX* se toma '/' como separador.

En nuestro empeño por obtener una solución, se encontró en la ayuda relativa a *MATLAB Commands for Working with Target Preferences*[18] un modo de trabajo que permita la configuración de una interfaz adecuado. Se halló un par de funciones, sólo funcionales sobre *mpc555dk*, que permite definir y capturar las preferencias de configuración de la interfaz que comunica *RTW* con los dos compiladores cruzados que admite el paquete de desarrollo de *Motorola*. Estas funciones se encuentran dentro de la ruta de directorio de este *kit* de desarrollo y presentan el formato de *Matlab p*[19].

Estas funciones son: *gettargetprefs* y *settargetprefs*. La primera de ellas devuelve, bajo el formato de un objeto, la configuración actual de *mpc555dk*; la segunda la define a partir de un objeto dado.

La línea de comando para obtener la configuración del objetivo *mpc555dk* es:

tpObj = gettargetprefs('mpc555dk');

Siendo éste el único argumento que admite. En la variable *tpOjb* se almacena el contenido del objeto.

Por otra parte, la orden para fijar la configuración del objetivo es:

settargetprefs('mpc555dk','PropertyName','PropertyValue');

[18] *Órdenes de Matlab para trabajar con las preferencias del sistema objetivo*.
[19] Formato intermedio entre *m* y un archivo ejecutable, por lo tanto, no modificable.

Siendo:

- *PropertyName*: El nombre del campo que se desea modificar.
- *PropertyValue*: El valor que se le asignará al campo.

Esta función también admite el formato siguiente cuando se pretende, con una única orden, configurar toda la interfaz completa:

settargetprefs('mpc555dk', tpObj);

Exige que, con anterioridad, se haya definido el objeto *tpObj* con los valores deseados; nosotros tomaremos los siguientes:

- TargetCompiler = 'CodeWarrior';
- TargetCompilerPath = 'd:\sw_fcs\arm\adsv1_2\bin';
- CommTimeout = '4';
- TargetDebugger = 'CodeWarrior';
- TargetDebuggerExe = 'd:\sw_fcs\arm\adsv1_2\bin\axd.exe';
- CommHostPort = 'COM2';
- CommTargetPort = 'COM1';
- CommBaudRate = '38400';

Una vez establecidas las preferencias del sistema objetivo, de forma coherente con el compilador cruzado que estamos usando y con la conexión *host-target* mediante los puertos de comunicaciones, planteamos los tres modos de funcionamiento que proporciona *mpc555dk* para averiguar cuál de ellos funciona correctamente bajo nuestra configuración. Para ello elaboramos un diagrama de bloques de *Simulink* muy sencillo, de forma que nos permita poner en marcha todo el proceso.

Para empezar proponemos el modo en tiempo real, por lo tanto hemos de seleccionar como plantilla *mpc555rt.tmf*. Al solicitar que *RTW* genere el código fuente asociado al diagrama se incurre en errores; debido a esto nos vemos obligados a descartar este modo.

Como segunda opción escogemos la plantilla *mpc555pil.tmf*, de forma que el modo de funcionamiento sea el denominado *processor-in-the-loop*, pero, igualmente, se vuelven a producir errores llevándonos a rechazar este modo.

Como último caso probamos el modo *Algorithm Export*, con lo que hemos de seleccionar la plantilla *mpc555exp.tmf* como interfaz. Invocamos la generación de código por parte de *RTW*, la cual se realiza satisfactoriamente según a los mensajes que *Matlab* nos envía a través de su

consola de comandos. Nuevamente, solicitamos la generación de código, pero invocando todo el proceso de construcción. Gracias a la configuración de las preferencias se abrió automáticamente la aplicación *CodeWarrior*, pero con el compilador interno denominado *555Phytec Debug Version*. Éste el propio de la máquina de *Motorola*, en lugar de la de *ARM*.

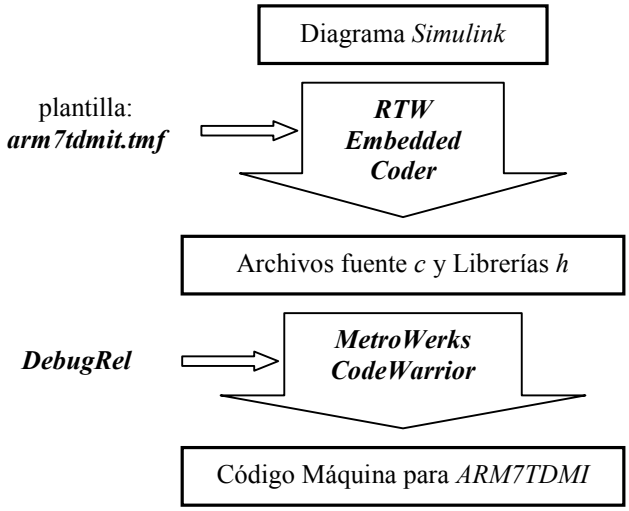
3.6 Estudio del Funcionamiento de la interfaz "mpc555dk".

Tal y como se han ido desarrollando los planteamientos, en los apartados anteriores, sobre el establecimiento de la interfaz de comunicaciones entre *Real Time Workshop* y el compilador *MetroWerks CodeWarrior* tenemos dos opciones. Una de ellas es manual; la otra, automática.

El planteamiento manual divide claramente la generación de código fuente *C++* y su compilación a código máquina. Este modo de proceder posee la ventaja de que ofrece una solución simple y factible, sin más que incluir en un nuevo proyecto del compilador todo el conjunto de archivos generados y sus dependencias estáticas. Pero no aprovecha, al máximo, las bondades de la técnica de prototipado rápido.

Por otro lado, la forma de trabajar automática sí goza de los beneficios que otorgan estas técnicas porque desde la especificación mediante un diagrama de bloques a la fase de pruebas sobre el equipo objetivo existe un único paso, siendo transparentes todos los procesos intermedios de cara al usuario.

Estos pasos intermedios serán detallados mediante el siguiente diagrama:



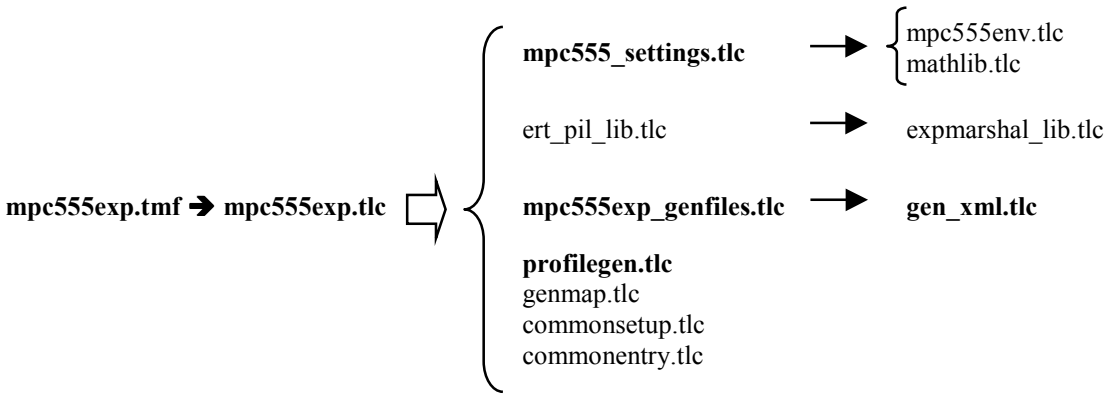
Del diagrama de bloques expresado con el lenguaje *Simulink* se logran los archivos que contienen el código fuente en *C++* junto con sus librerías mediante la intervención de *Real Time Workshop Embedded Coder*. Éste invoca a la aplicación *MetroWerks CodeWarrior* a través de controles *ActiveX* que proporciona el mismo *Matlab*.

Se estudió la opción de invocar la utilidad *make* de *CodeWarrior*, pero debido a que ésta no era accesible desde la línea de comandos [B36.2] de acuerdo con la guía del usuario proporcionada por *ATMEL*, se optó por el uso de los controles *ActiveX*.

3.6.1 Definiciones y Funcionalidad de los archivos TLC.

En este punto se presentará el conjunto principal de los archivos que usa la interfaz de *Motorola* y que están diseñados para ser interpretados por el *TLC*. Para lograr este objetivo se hará hincapié en la estructura de dependencia existente entre los archivos antes mencionados, así como la funcionalidad y el papel que desempeñan en la interfaz que constituyen. Además, en cada uno de estos archivos se define un grupo de variables –o de etiquetas para *TLC*– de carácter global a toda la interfaz.

Primeramente, describamos la estructura de dependencia entre los archivos de interfaz; para, posteriormente, indicar cuál es la funcionalidad y el uso de las variables que en ellos se definen. Esto es:



Al seleccionar como sistema objetivo, dentro del seleccionador que proporciona el panel de opciones de *RTW*, el *kit* de evaluación de *Motorola* denominado *MPC555DK* bajo el modo *Algorithm Export* mediante la plantilla *mpc555exp.tmf*, se opta por la interfaz representada por esta estructura. Hemos usado el tipo de letra *negrita* para remarcar cuáles son los archivos *TLC* más importantes.

- *mpc555exp.tmf*: se trata de la plantilla *makefile* matriz que inicia el proceso de construcción por *RTW* del código fuente y facilita la invocación de *CodeWarrior* para construir un proyecto para el sistema objetivo. Asimismo, en él se especifican las características propias de la utilidad *make* que utiliza la interfaz de *Motorola*, es decir, la denotada por *gmake*[20].
- *mpc555exp.tlc*: es el archivo del sistema objetivo para el modo *Algorithm Export* de *Motorola* para el *kit* de evaluación *MPC555DK*.
- *mpc555_settings.tlc*: define la configuración para el objetivo *MPC555DK*.
  - *mpc555env.tlc*: especifica el entorno *TLC* comprobando y fijando las rutinas.
  - *mathlib.tlc*: los métodos y los datos por defecto que permiten que se dé apoyo a las librerías para que el sistema objetivo trabaje en tiempo real.
- *ert_pil_lib.tlc*: define las librerías que facilitan el uso de la tecnología *PIL* mientras se utiliza un archivo de sistema objetivo embebido con restricciones de tiempo.
  - *pilmarshal.tlc*: se trata de la librería de funciones que facilitan la simulación *PIL* para un sistema embebido en tiempo real.
- *mpc555exp_genfiles.tlc*: gracias a este archivo se generará todo fichero necesario para la construcción del objetivo *mpc555exp*.
  - *gen_xml.tlc*: en él se constituye la función que genera el fichero de intercambio de datos entre *RTW* y *CodeWarrior*.
- *profilegen.tlc*: con este fichero se genera el informe *HTML* resumen del proceso de generación de código realizado por *RTW*.
- El resto de ficheros destacados forman parte de conjunto común a todo proyecto para *RTW*.

3.6.2 Línea de Comandos para CodeWarrior.

Gracias a la guía de usuario que facilita *ARM Limited* a través de su portal en Internet, podemos comprobar que *CodeWarrior* permite la recepción de órdenes a través de la línea de comandos del sistema operativo. Además de esto, se proporciona la lista de comandos y de opciones que poseen.

Según el apunte bibliográfico [B38.3], en su apéndice *A* se especifica la forma de trabajar con el entorno de desarrollo integrado *CodeWarrior IDE*[21] a través de la línea de comandos. Para ello *MetroWerks* ha generado la aplicación denominada *cmdide.exe*. Ésta se constituye como una consola que puede ser iniciada desde la línea de comandos para construir los archivos de un proyecto que haya sido creado y editado con *CodeWarrior IDE*. Por lo tanto, *cmdide.exe*

[20] gmake: se trata de una utilidad *make* proporcionada por *Matlab* y que posee licencia *GNU*.
[21] IDE: acrónimo del inglés, *Integrated Development Environment*.

invoca a *CodeWarrior IDE*, le pasa los parámetros adecuados para que produzca un proceso de construcción del proyecto *MCP* y espera a que *IDE* termine su operación.

Los argumentos de la línea de comandos para *cmdide.exe* son:

Argumentos de la línea de comandos para *cmdide.exe*	
Projectname	Especifica el proyecto *MCP* a utilizar.
/t Targetname	Especifica el nombre del sistema objetivo.
/r	Borra los ficheros objeto del sistema objetivo actual antes de construir.
/b	Construye el objetivo actual.
/q	Cierra el *IDE* después de construir.
/c	Cierra el proyecto después del proceso de construcción.
/v[y|n|a]	Opciones para convertir los proyectos al abrirlos: ▪ *y*: convertir sin preguntar. ▪ *n*: no convertir. ▪ *a*: preguntar si se convierte.
/s	Fuerza a que línea de comandos sea procesada en una nueva instancia del *IDE*, en lugar de usar la instancia actual.

Si más de un proyecto se indica para ser abierto desde la línea de comandos, los argumentos /t y /b se aplican sólo al primer fichero encontrado en la lista de archivos. Si no se especifica ningún proyecto entonces se aplica el comando sobre el proyecto por defecto, es decir, el que ya está abierto por *CodeWarrior IDE*.

Por ejemplo, para construir el proyecto llamado *dhryansi*, se cambia el directorio de trabajo de la consola de comandos a la carpeta que recoge el archivo *dhryansi.mcp* y se teclea:

Cmdide dhryansi.mcp /t DebugRel /c /b

En este caso se construirá el proyecto *dhryansi.mcp* para el sistema objetivo *DebugRel*, cerrándose tras el proceso de construcción.

Considerando el apunte bibliográfico [B38.4][22], se nos muestra cómo realizar el proceso de construcción, llevado a cabo por *CodeWarrior* de forma interna, a través de la línea de comandos. De acuerdo con esta documentación se tiene acceso a las utilidades de compilado, de depurado, de ensamblado y de enlazado.

Otra forma de obtener cuál es el argumento asociado a cada una de las opciones del proceso de construcción de proyectos *MCP* consiste en observar el cuadro de diálogo siguiente. Para ello hemos de tener abierto la aplicación *MetroWerks CodeWarrior*, ésta depende de la ruta de directorio *...\ARM\ADSv1_2\Bin\IDE.exe*. Con un proyecto *MCP* activo seleccionamos la opción *DebugRel Settings...* contenida por el menú *Edit*. Una vez abierto el cuadro de diálogo correspondiente, nos centramos en cualquier opción de *Language Settings*[23]. Como podemos observar, aparece un cuadro denominado *Equivalent Command Line*[24], el cual se adapta ante los cambios en las diferentes opciones de la plantilla.

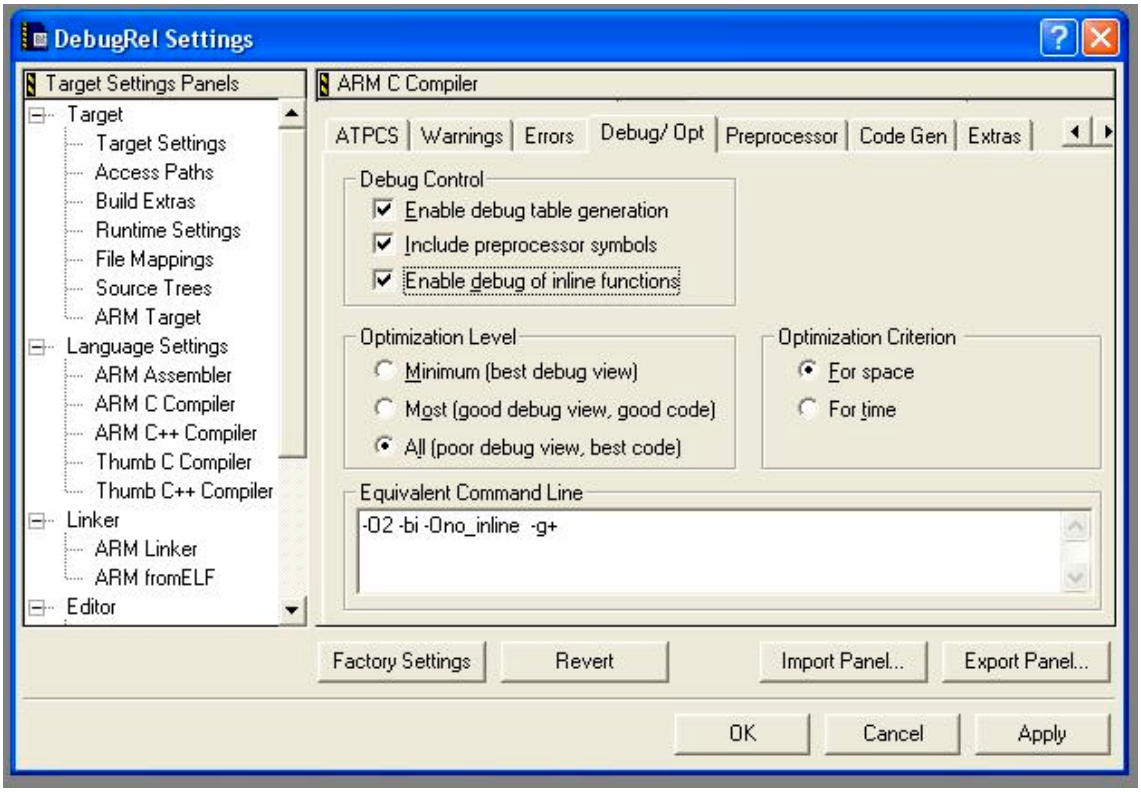


Diagrama: Línea de Comandos Equivalente.

[22] En su apartado *3.3 Building form the command line* (pág. 102 y ss.).
[23] Language Settings: del inglés, *Configuración del Lenguaje*.
[24] Equivalent Command Line: del inglés, *Línea de Comandos Equivalente*.

3.6.2.1 Compilado desde la Línea de Comandos.

Para el compilado existen cuatro variantes:

Variantes de Compilado.			
Nombre	**Lenguaje**	**Código fuente**	**Salida**
armcc.exe	*C*	*C*	Código *ARM* de 32 bits
tcc.exe	*C*	*C*	Código *Thumb* de 16 bits
armcpp.exe	*C++*	*C* o *C++*	Código *ARM* de 32 bits
tcpp.exe	*C++*	*C* o *C++*	Código *Thumb* de 16 bits

- Construyendo un ejemplo desde la línea de comandos:
  - Compilar el archivo *C main.c* con cualquier de las variantes:

    armcc -g -O1 -c main.c (for ARM)
    tcc -g -O1 -c main.c (for Thumb)

    Donde:
    - *'-g'*: le dice al compilador que añada las tablas de depuración.
    - *'-O1'*: le indica que seleccione la mejor optimización posible siempre que mantenga una vista de depuración adecuada.
    - *'-c'*: le dice que compile únicamente (sin enlazado).
  - Enlazado de la imagen generada por el compilado usando el siguiente comando:

    armlink main.o -o embed.axf

    Donde:
    - *'-o'*: especifica que el archivo de salida es el argumento siguiente, es decir, *embed.axf*.
  - Úsese *armsd* o *AXD Debugger* para cargar y probar la imagen.

3.6.2.2 Ensamblado desde la Línea de Comandos.

La sintaxis básica para usar el ensamblador[25] para *ARM* desde la línea de comandos es:

armasm inputfile

Por ejemplo, para ensamblar el código existente en el archivo llamado *miarchivo.s*, se escribe: *armasm -list miarchivo.lst miarchivo.s*

[25] El ensamblador para *ARM* es la aplicación *armasm.exe*, situada en la carpeta *...\ADSv1_2\Bin*.

Esto produce un archivo objeto que se llama *miarchivo.o* y un fichero de listado de nombre *miarchivo.lst*. La descripción completa de las opciones de la línea de comandos y su sintaxis se recoge en la *Guía del Ensamblador para ADS*[26].

- Construir el ejemplo:
  - En cualquier editor de texto se introduce el código ensamblador para *ARM*, se guarda en el directorio de trabajo como *addreg.s*.
  - Se teclea en la línea de comandos: *armasm –list addreg.lst addreg.s*; para ensamblar el código fuente.
  - Se teclea: *armlink addreg.o –o addreg*; para enlazar el archivo.
- Cargar y Ejecutar el ejemplo en el depurador:
  - Se teclea *arssd addreg* para cargar el módulo en el depurador a través de la consola.
  - Se escribe *step*[27] para ejecutar paso a paso el resto del programa. Tras cada instrucción, se puede escribir *reg*[28] para mostrar el valor de los registros. Cuando el programa termine se escribe *quit* para regresar a la línea de comandos.

Para estudiar con detalle el lenguaje de programación en ensamblador para *ARM* o *Thumb*, véase la *Guía del Ensamblador para ADS*.

3.6.3 Archivos XML: Etiquetado y Estructura para CodeWarrior.

Los archivos *XML*[29] se han convertido últimamente en un estándar para la transferencia de documentos entre aplicaciones en Internet. La empresa [B38.2] que más empeño y dedicación ha aportado al desarrollo de este tipo de archivos ha sido *SUN Microsystems*, a la que se ha añadido la gran mayoría de las empresas del sector, incluyendo *Microsoft Corporation*. Esto último ha propiciado una explosión en el uso de este estándar.

Gracias a este hecho, *Matlab* exporta e importa ficheros con el formato *XML* al igual que *MetroWerks CodeWarrior*. Esto posibilita establecer un puente de comunicación automática entre los ficheros generados por *RTW* y el entorno de desarrollo de *ADS*.

El formato de etiquetado del lenguaje *XML* se ha heredado de *HTML*. De hecho, la sintaxis es idéntica, salvo que las etiquetas de *HTML* poseen significado definido a priori y, en un fichero *XML* se requiere definir, a modo de cabecera, cuál será la estructura de etiquetado que presentará.

[26] En la referencia original: *ADS Assambler Guide*.
[27] Step: del inglés, paso.
[28] Reg: abreviatura de *register*, del inglés, registro.
[29] XML: acrónimo de *Extensible Mark-up Language*, del inglés, *Lenguaje de Etiquetado Extensible*.

Una diferencia crucial entre ambos lenguajes estriba en que para un fichero *HTML* la etiqueta representa una actuación sobre el valor que acota, mientras que en *XML* tan sólo identifica al dato que engloba. Veamos un ejemplo:

- *TEXTO*: Un intérprete de *HTML* mostraría "**TEXTO**", considerando el contenido en negrita.
- *<NombreEtiqueta>Valor</NombreEtiqueta>*: Al interpretarse como *XML* el programa asociaría el valor dado a la variable interna de nombre *"NombreEtiqueta"*.

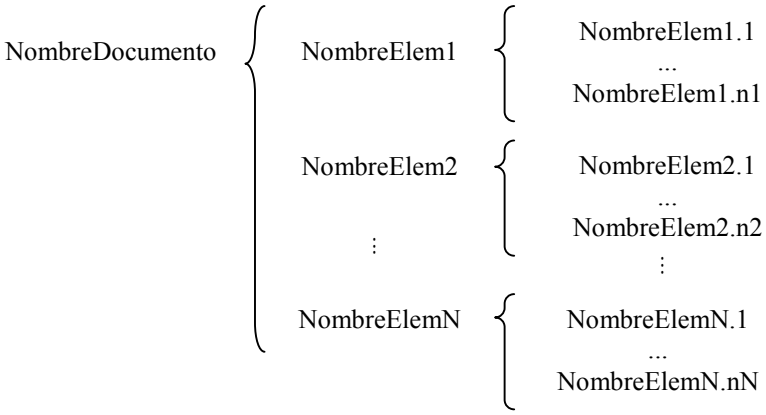
Un fichero *XML* tan sólo puede analizarlo correctamente la aplicación para la que está destinado, puesto que la estructura de etiquetado es propia de cada programa. La ventaja de *XML* radica en que los valores y las etiquetas se muestran con caracteres *ASCII* perfectamente reconocibles con cualquier editor de texto, siendo, por lo tanto, modificables por cualquier editor.

Por otro lado, como la estructura del etiquetado en *XML* no está definida, debe existir una cabecera al inicio del mismo que especifique dicha estructura. La manera en la que se indica esto presenta la siguiente forma:

```
<!DOTYPE NombreDocumento [
<!ELEMENT nombreElem1(nombreElem1.1, nombreElem1.2,...,nombreElem1.n1)>
<!ELEMENT nombreElem2(nombreElem2.1, nombreElem2.2,..., nombreElem2.n2)>
...
<!ELEMENT nombreElemN(nombreElemN.1, nombreElemN.2,..., nombreElemN.nN)>
<!ELEMENT nombreElem1.1(...)>
<!ELEMENT nombreElem1.2(...)>
...
<!ELEMENT nombreElemN.nN(...)>
...
<!ELEMENT nombreElemM.M1.....MN(#PCDATA)>
]>
```

Así pues, el elemento *nombreElem1* se sitúa en el nivel más alto de la jerarquía de etiquetado, formándose por los elementos desde *nombreElem1.1* hasta *nombreElem1.n1*, los que se encuentran en el segundo nivel de etiquetado. Si un elemento no posee elementos constituyentes entonces es que se forma únicamente de un valor para la aplicación; esto queda denotado mediante *#PCDATA*.

Quedando, gráficamente, la estructura de etiquetado de la siguiente forma:



Antes de la cabecera se indica la tabla de caracteres que utilizará el documento. Para ello se hace uso de las siguientes líneas:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<?codewarrior exportversion="1.0.1" ideversion="4.2" ?>
```

Ciertamente, tan sólo se necesita la primera línea para indicar la tabla a usar. En ella se concreta la versión del documento dentro del *standard* de *XML* y el tipo de codificación que, en este caso, es *UTF-8*. Ésta es la codificación *ISO International*.

Tras la cabecera se incluye el contenido de datos del archivo *XML*el cual presenta una forma similar a la de un fichero *HTML*. Siguiendo con el esquema de etiquetado propuesto, el contenido podría tener la siguiente forma:

```
<NOMBREDOCUMENTO>
        <NOMBREELEM1>
                <NOMBREELEM1.1>valor1.1</NOMBREELEM1.1>
                <NOMBREELEM1.2>valor1.2</NOMBREELEM1.2>
                ...
        </NOMBREELEM1>
        <NOMBREELEM2>...</NOMBREELEM2>
        ...
</NOMBREDOCUMENTO>
```

En este ejemplo se ha supuesto que las etiquetas *nombreElem1.1* y *nombreElem1.2* se encuentran en el nivel más bajo de la jerarquía de etiquetado; tomándose sus campos como un dato de la aplicación.

Puesto que cada aplicación requiere de una estructura de etiquetado propia, fue necesario estudiar cuál era la que utilizaba el entorno de desarrollo de *MetroWerks CodeWarrior*. Para ello se estudió la documentación incluida en la guía del usuario que proporciona *ARM* a través de su portal de Internet http://www.arm.com.

En esa documentación se indica la utilidad que posee *CodeWarrior* para comparar dos archivos con extensión *XML*, la cual se denomina *Compare Files Setup*. Para llegar hasta ella se debe hacer clic en el menú *Search* y seleccionar la opción *Compare Files...* dentro de la aplicación *CodeWarrior*. Tras esto se muestra un cuadro de diálogo en el que podemos indicar las rutas de directorio asociadas a los dos documentos *XML* que pretendemos comparar.

Si comparamos dos proyectos de *CodeWarrior* expresados en *XML*, uno de ellos elaborado por *RTW* cuando trabaja con la interfaz ideada por *Motorola* y, otro realizado desde *CodeWarrior* al exportar un proyecto *MCP* para *ARM7TDMI*; obtenemos la siguiente comparativa:

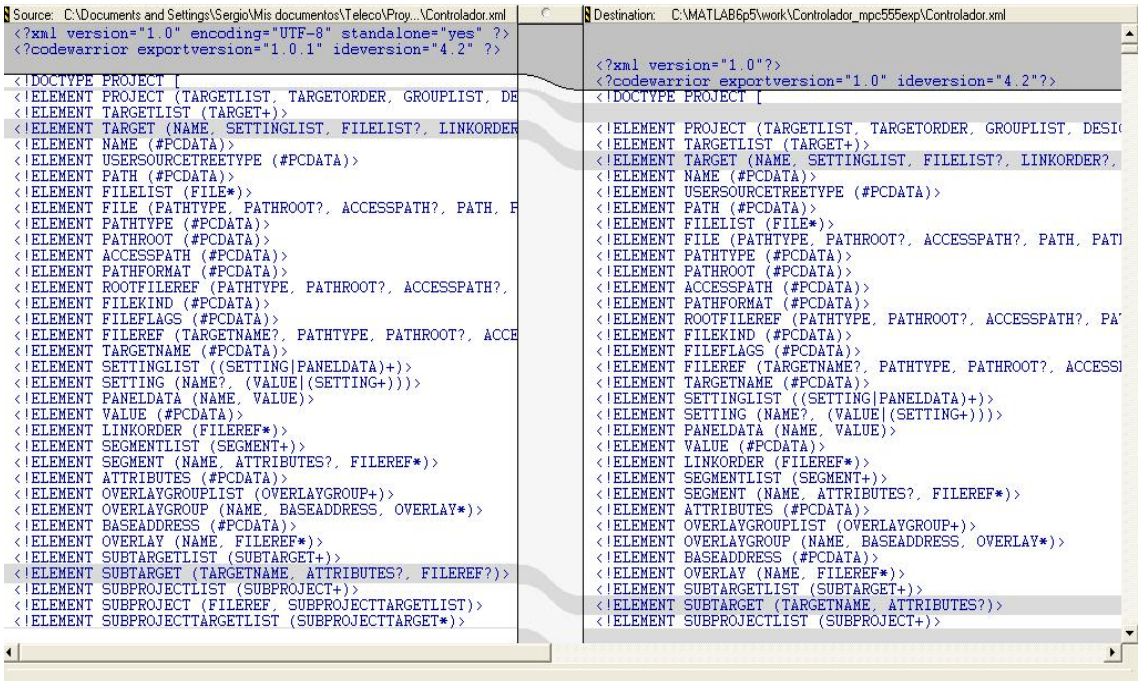


Figura: Comparativa *XML*.

A lo largo de las dos columnas se muestra el contenido de cada uno de los archivos *XML* que han sido seleccionados. Concretando el contenido de esta figura, tan sólo se presenta el principio de la cabecera utilizada para definir la estructura de etiquetado que presenta cada documento. Su contenido completo es mucho más amplio, tal y como lo está indicando la barra vertical.

También se observa que quedan sombreadas las estructuras de etiquetado que son compartidas por ambos archivos; de esta forma se facilita enormemente la comparación. Asimismo, podemos averiguar cuál es la estructura de etiquetado concreta tanto de un proyecto

construido para el sistema objetivo *ARM7TDMI* –documento de la columna izquierda– como de otro diseñado para *MPC555DK* bajo el modo *EXP*.

No sólo es diferente la estructura de etiquetado entre ambos tipos de proyecto, sino también los valores que adopta la parte común de dichas estructuras. A modo de ejemplo introduciremos la siguiente figura que recoge una parte de los documentos:

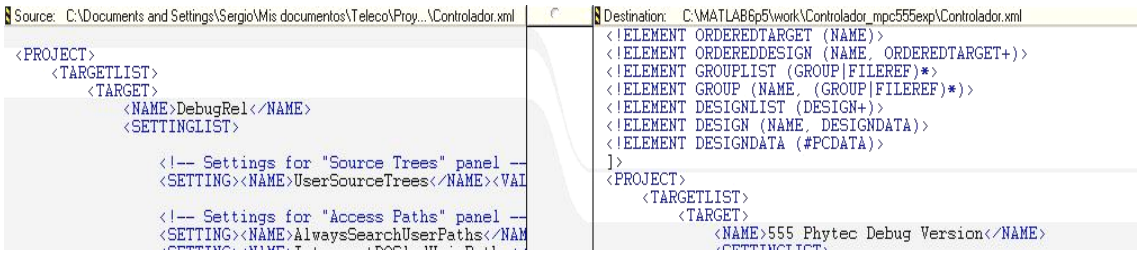


Figura: Indicación del Sistema Objetivo.

Así pues, la etiqueta *<NAME>*, dependiente de *<PROJECT> <TARGETLIST> <TARGET>*, contiene el nombre del sistema objetivo o *target* que utiliza cada proyecto. Siendo *DebugRel* para el documento de la columna izquierda y, *555 Phytec Debug Version* para el de la derecha.

De igual forma es posible averiguar cómo y dónde se incluyen los archivos que forman parte de cada proyecto. Esto es:

Figura: Inclusión de Archivos del Proyecto.

Se constata que la referencia a un archivo fuente, sea código *c* o una librería *h*, se lleva a cabo gracias a la etiqueta <*FILEREF*>; la cual depende de la rama de etiquetado <*PROJECT*> <*GROUPLIST*> o de <*PROJECT*> <*GROUPLIST*> <*GROUP*>, según se esté considerando el proyecto para *ATMEL* o para *Motorola*, respectivamente.

Si bien es cierto que actuando desde la interfaz que proporciona *CodeWarrior* se permite agrupar los archivos que forman un proyecto. De acuerdo con el documento *XML* generado por *RTW*, *Motorola* optó por agruparlos de forma automática. Por ejemplo, dentro del grupo que denominó *ModelFiles* se encuentran los archivos *Controlador.c* y *Controlador_data.c*.

De igual modo, observamos que una referencia a un archivo consta de una serie de términos, como son: <*TARGETNAME*>, <*PATHTYPE*>, <*PATH*> y <*PATHFORMAT*>. Siendo:

- <*TARGETNAME*>: El nombre del sistema objetivo; tomando los valores *DebugRel* o *555 Phytec Debug Version* según los casos.
- <*PATHTYPE*>: Si la ruta *–path–* indicada incluye la dirección completa o únicamente el nombre del archivo. En ambos casos el valor que toma esta etiqueta es *Name*[30].
- <*PATH*>: Indica la ruta de directorio del archivo a incluir dentro del sistema de ficheros del sistema *Host*. Siendo coherente con el tipo de ruta, se incluye sólo el nombre del archivo.
- <*PATHFORMAT*>: Se concreta la plataforma *Hardware* de trabajo, pudiéndose distinguir entre el sistema UNIX o Windows. Se debe indicar porque la cadena que representa la ruta de directorio se expresa de forma diferente según la plataforma.

El archivo *XML* generado por *RTW* para el sistema objetivo *MPC555* se obtiene de forma automática a partir del proceso de generación de código. Éste resulta correctamente interpretado por *CodeWarrior*. En cambio, para los proyectos que se han usado de prueba en puntos anteriores de esta memoria *CodeWarrior* no genera el consiguiente archivo *XML*de forma automática al trabajar sobre un proyecto *MCP*. Se debe solicitar explícitamente a través de la utilidad *Export Project...* . A ella se llega sin más que seleccionándola del menú *File* de la aplicación. Hay que considerar que para que la exportación esté activa resulta necesario tener un proyecto *MCP* abierto en *CodeWarrior*.

[30] *Name*: del inglés, Nombre.

3.6.4 Controles ActiveX y objetos COM de Matlab.

Los controles *ActiveX* son una herramienta desarrollada por *Microsoft Corporation* para tratar las diferentes aplicaciones como si fueran un objeto. Gracias a ello es posible invocar cualquier funcionalidad de una aplicación a través de su interfaz sin más que conocer los nombres de los diferentes métodos. El método de acceso a la misma se lleva a cabo con el *script* correspondiente.

Matlab proporciona un conjunto de funciones que gestionan los controles *ActiveX* permitiendo, de este modo, actuar sobre la interfaz del objeto aplicación. La estructura de su interfaz es propia de cada aplicación, así pues, *CodeWarrior* presenta la suya.

Un objeto *COM* de *Matlab* se constituye como una instancia de la clase objeto componente, cuya denominación original en inglés es *Component Object Model*. Su utilidad consiste en completar el encapsulamiento de las aplicaciones para ser tratadas como objetos. Para acceder a un objeto *COM* se requiere definir previamente una interfaz con la aplicación, para ello *Matlab* facilita una serie de funciones.

Tanto *Matlab* como *Microsoft Windows* pueden actuar como servidores de la automatización sobre objetos *COM*. Una automatización *COM* permite que una aplicación o componente invoque a otra aplicación o componente. La primera se denomina controladora, mientras que la segunda recibe el nombre de servidora. Para el caso que nos ocupa la aplicación *Matlab* controla e invoca al servidor *CodeWarrior*, que responde ante sus peticiones.

Para poder trabajar con un servidor *COM* es imprescindible haber definido un manejador que lo identifique. Con este fin existe la función *actxserver* [B36.1] cuya llamada presenta la siguiente forma:

h = actxserver (progid [, machinename])

Siendo:

- *'h'*: El manejador o *handle* que identifica al servidor.
- *'progid'*: Él es identificador del programa o *program identifier*. Éstos han sido definidos de forma que cada aplicación posee su propio identificador, el cual no se aplica a ningún otro programa.
- *'machinename'*: Únicamente se aplica este campo opcional cuando la aplicación que actúa como servidora se encuentra en una máquina remota con respecto a la que invoca.

Veamos un ejemplo de cómo se invocaría como aplicación servidora a *Microsoft Excel*:

- Lanzamos *Microsoft Excel* y hacemos visible su ventana principal; para ello hay que conocer el identificador único esta aplicación. Introducimos el siguiente comando en la línea de comandos del *Matlab*.

e = actxserver ('Excel.Application')

- A lo que nos responde, siendo *'e'* el nombre del manejador *COM*:

e = COM.excel.application

- Para hacer que *Excel* sea visible se fuerza que el método denominado *Visible* tome el valor 1; para ello hacemos uso de la función *set* de *Matlab* de la siguiente forma:

set(e, 'Visible', 1);

- Llamando al método *get* sobre el objeto *Excel* se obtiene la lista de todas las propiedades de la aplicación: *get(e)*.
- Cuya respuesta es:

```
ans = Application: [1x1 Interface.excel.application.Application]
      Creator: 'xlCreatorCode'
      Parent: [1x1 Interface.Excel.Application.Parent]
      Workbooks: [1x1 Interface.excel.application.Workbooks]
      UsableHeight: 666.7500
...
```

- Podemos crear una interfaz: *eWorkbooks = get(e, 'Workbooks')*; siendo su respuesta el manejador de la interfaz:

eWorkbooks = Interface.excel.application.Workbooks

- Para obtener una lista con todos los métodos admitidos por una interfaz dado se hace uso de la función *invoke*[31] con el argumento el manejador de la interfaz: *invoke(eWorkbooks)*; cuya respuesta es:

```
ans =
   Add: 'handle Add(handle, [Optional]Variant)'
     Close: 'void Close(handle)'
      Item: 'handle Item(handle, Variant)'
      Open: 'handle Open(handle, string, [Optional]Variant)'
      OpenText: 'void OpenText(handle, string, [Optional]Variant)'
```

- Podemos invocar el método *Add*[32] con el fin de obtener un nuevo libro de *Excel*:

w = Add(eWorkbooks)

[31] Invoke: del inglés, invocar.
[32] Add: del inglés, añadir.

- Siendo su respuesta:

$$w = Interface.Excel.Application.Workbooks.Add$$

Además se abre un nuevo libro de nombre *Libro1* dentro de la ventana propia de *Microsoft Excel*.

- Para salir de la aplicación y borrar su manejador del espacio de trabajo de *Matlab*, usamos las siguientes líneas: *Quit(e); delete(e);*

Resulta muy útil para visualizar los métodos que están disponibles para un objeto *COM* dado el visor de objetos que *Matlab* proporciona. Sin más que hacer doble clic sobre el manejador del objeto dentro del espacio de trabajo se nos muestra, en una ventana independiente, la estructura de métodos asociada.

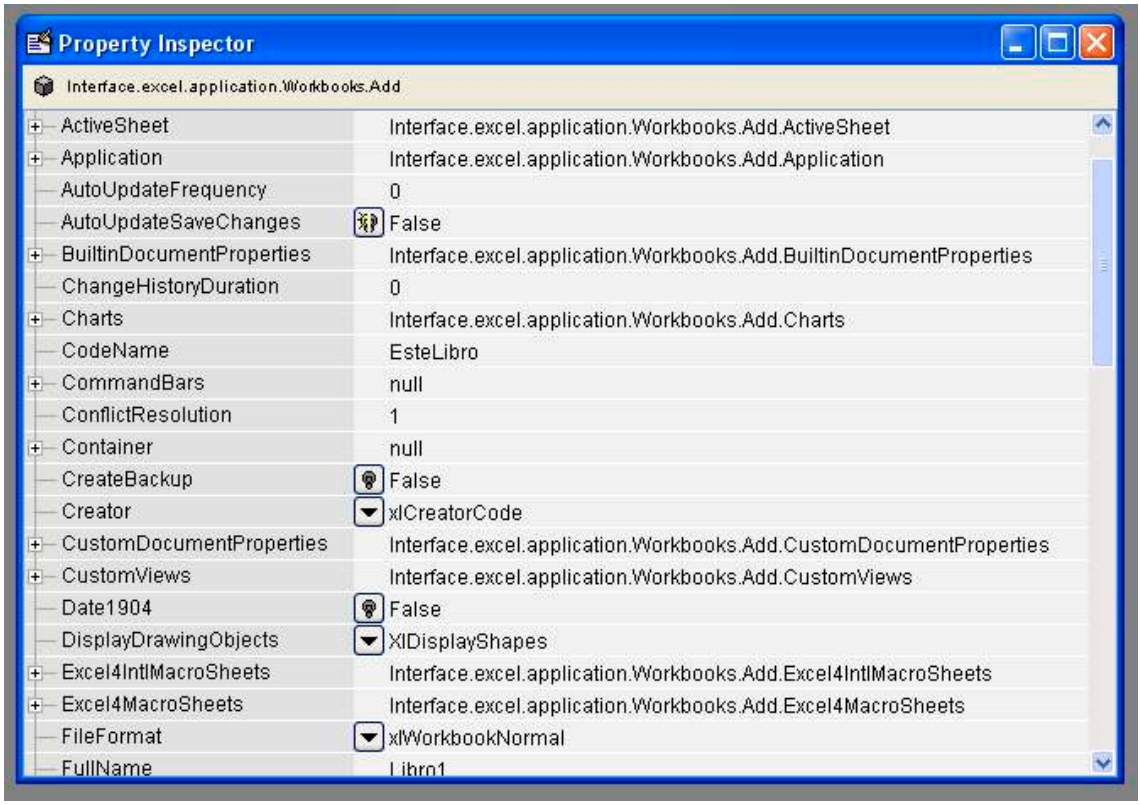


Figura: Métodos del Objeto *COM 'e'*.

3.6.5 Invocación de los métodos de CodeWarrior.

De acuerdo con el método de trabajo que proporciona el uso de los objetos *COM*, *Motorola* diseñó un *script* para *Matlab* que elabora los manejadores adecuados para *CodeWarrior*. Este archivo se denomina *cwautomation.m* y se localiza dentro de la ruta de directorio: ... *\MATLAB \toolbox \rtw \targets \mpc555dk \common \tools \codewarrior*. A continuación describiremos la forma en la que *cwautomation.m* lleva a cabo su cometido:

- Nada más comenzar se establecen los nombres de los diferentes argumentos que permite *cwautomation.m*:

```
switch nargin
  case 2
   in_qualifiedXML  = char(varargin(1));
   out_qualifiedMCP = char(varargin(2));
   % Default action - The default, and backwards compatible, two argument
   % action is to open the project
   action          = 'open';
  case 3
   in_qualifiedXML  = varargin{1};
   out_qualifiedMCP = varargin{2};
   action          = varargin{3};
  otherwise
   error('cwautomation supports two or three arguments');
 end
```

Siendo *in_qualifiedXML* el nombre del primer argumento, *out_qualifiedMCP* el del segundo y *action*[33] el tercero. En el primero se almacena la ruta de directorio del fichero *XML* asociado al futuro proyecto de *CodeWarrior*. El segundo se refiere a la ruta del fichero cuya extensión será *mcp*[34]. Y el tercero, a la orden o acción que llevará a cabo la aplicación *CodeWarrior* –tengamos en consideración que ésta actuará como servidora de *Matlab*–.

Según el valor que tome la variable *action* se llevará a cabo la invocación pertinente del método adecuado. Veamos algunos ejemplos:

- En el caso de que se desee abrir un nuevo proyecto para *CodeWarrior* con los archivos fuente generados por *RTW* se hace uso de las siguientes líneas.

```
case 'open'
    ICodeWarriorApp = CreateCWComObject;
    CloseAll;
    ImportXML(in_qualifiedXML,out_qualifiedMCP);
    return;
```

[33] Action: del inglés, acción.
[34] *mpc*: acrónimo de *MetroWerks CodeWarrior Project*, siendo la extensión usada para los proyectos por CodeWarrior.

Cada una de ellas llama a su función correspondiente. Los diseñadores de *Motorola* no optaron por utilizar prototipos de funciones del estilo *nombreFunción(argumentos)*, sino que prefirieron hacer uso una notación más intuitiva que permite *Matlab*.

La primera de las funciones genera el objeto *COM* asociado a *CodeWarrior*, devolviendo el manejador adecuado. La segunda (*CloseAll*[35]) cierra todos los proyectos que hubiera en curso, en el caso de que la aplicación hubiera sido invocada con anterioridad. Y la tercera invoca al método que envía el archivo *XMI*a *CodeWarrior* para que éste lo transforme en un fichero proyecto *MCP*; quedando las rutas de ambos archivos especificadas en la llamada a la función *cwautomation.m*.

- En el caso de que se pretenda construir todo el proyecto por *CodeWarrior*, *action* tomará el valor *build*[36], siendo el código asociado el siguiente:

```
case 'build'
   ICodeWarriorApp = CreateCWComObject;
   CloseAll;
   ImportXML(in_qualifiedXML,out_qualifiedMCP);
```

La acción *build* lleva a cabo, en sus primeras líneas, la apertura de un proyecto en *CodeWarrior* y transfiere los datos asociados al mismo bajo un fichero *XML*. Posteriormente, hace uso de la función *invoke* proporcionada por *Matlab*, la cual realiza una llamada al método *RemoveObjectCode*[37] que depende del objeto *COM* apuntado por el manejador *IcodeWarriorApp*.

```
try
    invoke(ICodeWarriorApp.DefaultProject,'RemoveObjectCode', 0, 1);
catch
     error(['Error using COM connection to remove objects of current project. ' ...
            'Verify that CodeWarrior is installed correctly. Verify COM access' ...
            'to CodeWarrior outside of MATLAB.']);
end
```

La estructura de control de flujo denominada *try... catch* realiza la primera parte de sus enunciados –los comandos que se encuentran entre *try* y *catch*– mientras se incurra en un error; en cambio, el resto de los enunciados sólo se llevan a cabo tras un error. La función *error* de *Matlab* muestra un mensaje y aborta la función desde la que ha sido llamada, la que en este caso es *cwautomation.m*.

[35] CloseAll: del inglés, Cierra Todo.
[36] Build: del inglés, Construir.
[37] RemoveObjectCode: del inglés, Retirar Código Objeto.

```
try
    invoke(ICodeWarriorApp.DefaultProject,'BuildAndWaitToComplete');
catch
    error(['Error using COM connection to build current project. ' ...
        'Verify that CodeWarrior is installed correctly. Verify' ...
        'COM access to CodeWarrior outside of MATLAB.']);
end
```

Para concluir los procesos relativos a la acción *build* se invoca el método *BuildAndWaitToComplete*[38] gracias a la función *invoke*. Este proceso provoca que *CodeWarrior* compile y enlace el código fuente obtenido a partir del archivo *XML*, siempre que no se den errores durante él mismo.

- En el caso de que la acción indicada sea *download*[39] o *run*[40], se pasa a realizar los siguientes pasos:

```
case { 'download', 'run' }
   % Create a COM connection to CodeWarrior (This may start codewarrior)
   ICodeWarriorApp = CreateCWComObject;
   % Close any and all open projects to prevent errors when
   % importing the current project.
   CloseAll;
   % import an XML project description.
   ImportXML(in_qualifiedXML,out_qualifiedMCP);
   % Remove objects to cause a complete rebuild.
   CleanCW;
   % Make the project build
   BuildCW;
   % Make the project run
   RunCW;
 return;
```

Todas las líneas anteriores constituyen una llamada a una función, pero su prototipo no presenta la forma normalizada que se comentó en párrafos anteriores. Así, por ejemplo, la llamada *RunCW* provoca que se invoquen los métodos necesarios, dentro de la aplicación *CodeWarrior*, para que se ejecute el proyecto en la placa del sistema objetivo.

- En el caso de que *action* tome el valor *kill*[41]:

```
case 'kill'
   % Create a COM connection to CodeWarrior (This may start codewarrior)
   ICodeWarriorApp = CreateCWComObject;
   try
      invoke(ICodeWarriorApp,'Quit', 1);
   catch
      error(['Error using COM connection to close CodeWarrior']);
   end
 return;
```

[38] BuildAndWaitToComplete: del inglés, Construir y Esperar que se Complete
[39] Download: del inglés, descarga.
[40] Run: del inglés y en este contexto, ejecutar.
[41] Kill: del inglés, matar; aunque en este contexto se refiere a cerrar una aplicación.

Mediante una conexión *COM* con *CodeWarrior* se invoca al método *Quit*[42], el cual cierra el programa.

Veamos cómo se han definido cada una de las funciones que *cwautomation.m* ha utilizado:

- Para crear una conexión *COM* con *CodeWarrior*:

```
function ICodeWarriorApp = CreateCWComObject
    vprint([mfilename ': creating CW com object']);
    try
       ICodeWarriorApp = actxserver('CodeWarrior.CodeWarriorApp');
    catch
       error(['Error creating COM connection to ' ComObj ...
                '. Verify that CodeWarrior is installed correctly. ...
                '  Verify COM access to CodeWarrior outside of MATLAB.']);
   end
return;
```

Esta función hace uso de *actxserver*, la cual devuelve un manejador asociado al identificador universal de programa que posee *MetroWerks CodeWarrior*, siendo éste *CodeWarrior.CodeWarriorApp*. La variable que denota a dicho manejador queda denominada como *ICodeWarriorApp*.

- Función para cerrar todos los proyectos que pudieran estar abiertos bajo la aplicación *CodeWarrior*:

```
function CloseAll
   ICodeWarriorApp = CreateCWComObject;
   vprint([mfilename ': Closing DefaultProject']);
   try
      % close projects until there are none and try ends
      while (1 == 1)
         xxx = invoke(ICodeWarriorApp.DefaultProject, 'Close');
      end
   catch
      lasterr('');
   end
 return;
```

En esta función se invoca al procedimiento *Close* que cierra, cada vez que se llama, un proyecto de *CodeWarrior* que se encuentre abierto. Esta operación se repite hasta que no existe ningún proyecto por cerrar; ya que al llamar a ese procedimiento cuando no existe ningún proyecto abierto provoca un error, lo que, consecuentemente, gracias a la estructura *try... catch* da paso al bloque definido en *catch*. Éste contiene la función *lasterr*, la cual devuelve una cadena con el último error mostrado por *Matlab*.

[42] Quit: del inglés; cerrar.

- Función para importar un proyecto definido a través de un archivo *XML*

```
function ImportXML(in_qualifiedXML,out_qualifiedMCP)
    % Add argument checking. This method requires valid import and project arguments.
    if ~(exist(in_qualifiedXML) == 1) & (exist(out_qualifiedMCP) == 1)
        error([mfilename ': Missing or empty import or project argument']);
    end
     if (isempty(in_qualifiedXML) | isempty(out_qualifiedMCP))
         error([mfilename ': Missing or empty import or project argument']);
     end
     ICodeWarriorApp = CreateCWComObject;
     vprint([mfilename ': Importing']);
     try
        ICodeWarriorProject = ...
            invoke(ICodeWarriorApp.Application,...
                        'ImportProject', in_qualifiedXML,...
                         out_qualifiedMCP, 1);
     catch
         error(['Error using COM connection to import project. ' ...
                    ' Verify that CodeWarrior is installed correctly. ...
                      Verify COM access to CodeWarrior outside of MATLAB.']);
     end
```

Nada más comenzar esta función comprueba si el fichero *XML* y el archivo *MCP* existen los dos; en caso contrario se produce un error forzando la salida de la función *ImportXML*. De forma análoga se asegura que ambos documentos no estén vacíos.

Tras esto, se genera un objeto *COM* relacionado con *CodeWarrior* para poder invocar el procedimiento *ImportProject* con los argumentos necesarios. En caso de que se produzca un error durante el proceso de la importación del documento *XML*, se generaría el mensaje indicado en la llamada a la función *error*.

- Función para abrir *CodeWarrior*:

```
function ICodeWarriorApp = StartCW
  vprint([mfilename ': starting']);
  ICodeWarriorApp = CreateCWComObject;
  return;
```

Mediante la línea de comandos *StartCW* se invoca la generación de un objeto *COM* asociado a la aplicación *CodeWarrior*.

- Función para ejecutar el código máquina en la placa del sistema objetivo.

```
function status = RunCW
   vprint([mfilename ': running']);
   ICodeWarriorApp = CreateCWComObject;
   % Use BuildWithOptions to get the project to download and run.
   try
      xxx = invoke(ICodeWarriorApp.DefaultProject,'BuildWithOptions',0,2);
   catch
      error(['Error using COM connection to run current project.' ...
                 ' Verify that CodeWarrior is installed correctly. Verify ...
                   COM access to CodeWarrior outside of MATLAB.']);
   end
```

```
  % Invoke the start method again to be sure codewarrior has
  % completed starting the run before returning.
  StartCW;
return;
```

Esta función, una vez creado el objeto *COM* pertinente, se invoca el procedimiento *BuildWithOptions*[43], el cual elabora todo el proceso indicado, incluyendo la llamada a la aplicación *AXD Debugger* tal y como se expuso en el capítulo 2.

- Función para construir un proyecto de *CodeWarrior*.

```
function status = BuildCW
 vprint([mfilename ': building']);
 ICodeWarriorApp = CreateCWComObject;
 try
  invoke(ICodeWarriorApp.DefaultProject,'BuildAndWaitToComplete');
 catch
  error(['Error using COM connection to build current project. ' ...
          'Verify that CodeWarrior is installed correctly. Verify ...
COM access to CodeWarrior outside of MATLAB.']);
 end
return;
```

Si más que con la línea de comando *BuildCW* se produce la construcción completa del proyecto en curso gracias a la invocación del método *BuildAndWaitToComplete*.

- Función para retirar el código objeto de un proyecto en *CodeWarrior*.

```
function status = CleanCW
   vprint([mfilename ': cleaning']);
   ICodeWarriorApp = CreateCWComObject;
   try
      invoke(ICodeWarriorApp.DefaultProject,'RemoveObjectCode', 0, 1);
   catch
      error(['Error using COM connection to remove objects of current project. ' ...
          'Verify that CodeWarrior is installed correctly. Verify COM ...
          access to CodeWarrior outside of MATLAB.']);
   end
 return;
```

En esta función se llama al procedimiento denominado *RemoveObjectCode*.

El archivo *cwautomation.m* es únicamente invocado desde *tgtaction.m*, en el que se genera la decisión de la acción que se va a solicitar al *target*. Esto se lleva a cabo a través de este código para *Matlab*:

```
% pass the method on to cwautomation
%
if isempty(errmsg)
 try
      lasterr('');
      cwautomation( ...
          in_qualifiedXML, ...
```

[43] BuildWithOptions: del inglés, Construir con Opciones.

```
            out_qualifiedMCP, ...
            action);
  catch
        errmsg = sprintf('%s:\n%s', ThisFile, lasterr);
  end
end
```

En donde si no existe un mensaje de error *–errmsg–* entonces se inicia una estructura de control de flujo del tipo *try ... catch*, en la que si no se incurre en ningún error durante el proceso se invoca a *cwautomation*. En esta llamada se usan los siguientes argumentos:

- *in_qualifiedXML*.
- *out_qualifiedMCP*.
- *action*.

Todos ellos se definen en líneas previas dentro del archivo *tgtaction.m* de la siguiente manera:

```
if exist('in_qualifiedXML') ==0 | isempty(in_qualifiedXML)
  in_qualifiedXML  = [ ];
  try
    in_qualifiedXML  = evalin('base','importXML');
  end
end

if exist('out_qualifiedMCP') ==0 | isempty(out_qualifiedMCP)
  out_qualifiedMCP = [ ];
  try
    out_qualifiedMCP = evalin('base','outputProject');
  end
end

%
% Get the tgtaction method
%
if isempty(errmsg)
  try
    action = varargin{1};
  catch
    errmsg = 'a tgtaction method is required';
  end
end
```

El método de asignación de valores a las dos primeras variables es análogo, tal y como se desprende de la similitud de las líneas de código. Esto es, si la variable homónima no existe dentro del espacio de variables de *TLC* entonces se le asocia el valor devuelto por la función *evalin*. Esta función evalúa la expresión dada del espacio de trabajo indicado; siendo éste el primero de sus argumentos[44] y aquélla, el segundo.

Con respecto a la variable *action*, ésta toma el valor del primer argumento que recibió la función *tgtaction* cuando fue invocada durante el proceso de trabajo de la interfaz de *Motorola*, siempre y cuando no se produzca ningún error durante el mismo.

Luego, las variables para *TLC* asociadas a *in_qualifiedXML* y a *out_qualifiedMCP* son *importXML* y *outputProject*, respectivamente. Éstas últimas se definen tanto en el archivo *mpc555exp_genfiles.tlc* como en el *mpc555pil_genfiles.tlc*[45] mediante el siguiente código:

```
%matlab assignin("base","importXML", "%<MODEL_ABS_PATH>\\%<ModelName>.xml")
%matlab assignin("base","outputProject", "%<MODEL_ABS_PATH>\\%<ModelName>.mcp")
```

De acuerdo con la ayuda de *Matlab*, la función *assignin* añade una nueva variable al espacio de trabajo seleccionado; siendo éste, en este caso, el denotado por *base*. Además, debido a que la sentencia para *TLC* comienza por la palabra reservada *matlab*, se produce una invocación de la línea de comandos de *Matlab* con la declaración que le sigue a dicha palabra.

Finalmente, las variables *importXML* y *outputProject* toman los valores de los archivos origen proyecto *XML* y proyecto *MCP*, respectivamente. Nótese que *ModelName* toma el valor de *CompiledModel.Name*, es decir, el nombre del modelo de *Simulink*.

3.6.6 Generación Automática del proyecto XML.

En este punto se distinguen claramente dos partes. En la primera de ellas se presenta el análisis de la estrategia seguida por el equipo de trabajo de *Motorola* para la generación automática del archivo *XML*, estando éste asociado al sistema objetivo *MPC555*. En la segunda parte se muestra el proceso de ingeniería gracias al cual se elabora automáticamente el archivo *XML* correspondiente al sistema objetivo de *ATMEL*.

3.6.6.1 Análisis de la estrategia seguida por MPC555.

Aplicando procesos de ingeniería inversa sobre los numerosos archivos que conforman la interfaz de *Motorola*, se descubren dos de ellos en los que residen las funciones que llevan a cabo la construcción del archivo *XML*. Éstos son:

- *gen_xml.tlc*.
- *gen_xml_lib.tlc.*

Ambos se encuentran dentro de la ruta de directorio siguiente:

- *...\MATLAB\toolbox\rtw\targets\mpc555dk\pil\BSPs\phyCORE-555\tlc*

En el primero de estos dos archivos se especifica el formato de la estructura de etiquetas y los valores que éstas tomarán para el archivo *XML* que describe un proyecto *MCP* del sistema

[44] En ambos casos el espacio de trabajo se denomina *base*, que es el espacio *standard* de *Matlab*.
[45] Ambos se encuentran en el directorio: ...\MATLAB\toolbox\rtw\targets\mpc555dk\pil

objetivo *MPC555*. Además, este archivo incluye el código *TLC* cuya función es incluir los archivos generados por *RTW*.

El segundo de estos dos archivos constituye la librería, expresada mediante código *TLC*, que contiene todas las funciones usadas por *gen_xml.tlc*.

Estudiemos el código de ambos archivos, pero empecemos por *gen_xml.tlc*:

- El encabezado:

```
%% File: gen_xml.tlc
%%
%% $Revision: 1.19 $
%% $Date: 2002/04/18 20:07:26 $
%%
%% Copyright 2002 The MathWorks, Inc.
%%
%% Abstract: Embedded real-time system target file for Processor in
%% the Loop Simulation generation of a Metrowerks CodeWarrior project to
%% build the generated code.
%%

%% Global references to:
%%   BSP_DIR; MODEL_ABS_PATH; TARGET_TLC_DIR; MATLABROOT;
%%   model_defined_file; DK_TLC_TYPE; ModelName; PWD; link_cmd_file;
%%   bspFiles; CommAPIFiles; CommFiles; CommTimeout;

%% Function: CreateXMLProject ================================================
%% Abstract: Generate and XML project for Metrowerks and the MPC555.
%%
```

En él se indica el objetivo del archivo; siendo la generación de un proyecto *XML* para *MetroWerks* y el *MPC555*. Asimismo, se muestra el conjunto de las referencias globales que utiliza.

- Definición de la función que genera el código *XML*

```
%function CreateXMLProject() Output

  %assign ModelName = ::CompiledModel.Name

  %% NAME AND OPEN OUTPUT XML FILE
<?xml version="1.0"?>
<?codewarrior exportversion="1.0" ideversion="4.2"?>
<!DOCTYPE PROJECT [

<!ELEMENT PROJECT (TARGETLIST, TARGETORDER, GROUPLIST, DESIGNLIST?)>
<!ELEMENT TARGETLIST (TARGET+)>
```

En esta parte del código *TLC* aparece el prototipo de la función que generará el proyecto *XML*, siendo su nombre *CreateXMLProject*[46]. Además se da comienzo al encabezado de un fichero *XML* y se asocia a la variable *ModelName* el valor de *CompiledModel.Name*. De

[46] CreateXMLProject: del inglés, crear proyecto *XML*

acuerdo con la estructura *CompiledModel* recogida en *model.rtw*, la variable *Name* contiene el nombre del modelo *Simulink* a ser tratado por *RTW*.

- Definición de los subdirectorios de búsqueda:

```
<SETTING><NAME>SearchPath</NAME>
    <SETTING>
        <NAME>Path</NAME><VALUE>%<MODEL_ABS_PATH></VALUE>
    </SETTING>
    <SETTING>
        <NAME>PathFormat</NAME><VALUE>Windows</VALUE>
    </SETTING>
    <SETTING>
        <NAME>PathRoot</NAME><VALUE>Absolute</VALUE>
    </SETTING>
</SETTING>
```

De acuerdo con la estructura de etiquetado establecida en la cabecera de todo documento *XML*, la etiqueta <*SETTING*> se localiza dentro de <*PROJECT*> <*TARGETLIST*> <*TARGET*> <*SETTINGLIST*>. Además, permite que dentro de ella se definan nuevas etiquetas <SETTING>.

Según este extracto de *gen_xml.tlc*, se deduce que se está configurando como ruta de directorio o *SearchPath*, bajo la plataforma *Windows* y con carácter absoluto, el subdirectorio apuntado por la variable para el *TLC*[47] de nombre *MODEL_ABS_PATH*. Ésta hace referencia al directorio de trabajo de *Matlab* en el que se vuelcan todos los archivos fuente y librería generados por *RTW*.

De forma análoga se incluyen numerosas rutas de directorio que contienen archivos que necesita la interfaz de *Motorola* para su correcto funcionamiento. Por ejemplo, mostraremos las líneas de código *XML* relativas a la configuración de las rutas asociadas a una de las referencias estáticas de los archivos fuente generados por *RTW*. Sea:

```
<SETTING>
    <SETTING><NAME>SearchPath</NAME>
        <SETTING>
            <NAME>Path</NAME><VALUE>%<MATLABROOT>\extern\include</VALUE>
        </SETTING>
        <SETTING>
            <NAME>PathFormat</NAME><VALUE>Windows</VALUE>
        </SETTING>
        <SETTING>
            <NAME>PathRoot</NAME><VALUE>Absolute</VALUE>
        </SETTING>
    </SETTING>
    <SETTING><NAME>Recursive</NAME><VALUE>true</VALUE></SETTING>
    <SETTING><NAME>HostFlags</NAME><VALUE>All</VALUE></SETTING>
</SETTING>
```

[47] TLC: acrónimo de *Target Language Compiler*; del inglés, *Compilador del Lenguaje Objetivo*.

La ruta que se incluye en estas líneas depende el directorio raíz en el que *Matlab* haya sido instalado; el cual se hace referencia a través de la etiqueta del *TLC* denominada *MATLABROOT*[48]. Nótese que la carpeta *...extern\include*, dependiente de dicho directorio, contiene la referencia estática *tmwtypes.h*, la cual existe en todo proyecto realizado por *RTW*.

- Definición automática de directorios de búsqueda:

```
%assign path1 = "%<TARGET_TLC_DIR>\\..\\pil\\api\\commapi\\include"
%<insertSearchPath(path1,"absolute")>
```

La segunda de estas dos líneas invoca al compilador del lenguaje objetivo, de forma que éste realice una llamada a la función correspondiente. Gracias a esta función *insertSearchPath*[49], perteneciente al archivo *gen_xml_lib.tlc*, se incluyen las líneas *XML* que definen como directorio de búsqueda el valor de la variable *TARGET_TCL_DIR* para *TLC*, aportando la misma estructura que la selección inmediatamente anterior. Esta variable se define en *mpc555_settings.tlc* tomando el valor de la carpeta que contiene todo la interfaz ideada por *Motorola*.

Veamos cómo se define la función *insertSearchPath*:

```
%function insertSearchPath (path,type) Output
 <SETTING>
    <SETTING><NAME>SearchPath</NAME>
       <SETTING><NAME>Path</NAME><VALUE>%<path></VALUE></SETTING>
   <SETTING><NAME>PathFormat</NAME><VALUE>Windows</VALUE></SETTING>
   <SETTING><NAME>PathRoot</NAME><VALUE>%<type></VALUE></SETTING>
    </SETTING>
    <SETTING><NAME>Recursive</NAME><VALUE>true</VALUE></SETTING>
    <SETTING><NAME>HostFlags</NAME><VALUE>All</VALUE></SETTING>
 </SETTING>
%endfunction
```

Como podemos observar las líneas que son usadas por el *TLC* aparecen al principio y al final de este extracto, sirviendo como definición del prototipo de la función y como finalización de la misma. El uso de los argumentos de la función a lo largo del contenido de la misma se lleva a cabo como toda variable del *TLC*: se debe denotar entre '<' y '>', estando precedido por el símbolo '%'.

Asimismo, el contenido de esta función presenta la misma estructura de etiquetado que la inclusión de una ruta de directorio fija y definida.

[48] *MATLABROOT*: del inglés, Raíz *Matlab*.
[49] *InsertSearchPath*: del inglés, insertar ruta de búsqueda.

- Inclusión de los archivos fuente *c* y de las librerías *h*:

```
<FILELIST>
        <FILE>
          <PATHTYPE>Name</PATHTYPE>
          <PATH>__ppc_eabi_init.c</PATH>
          <PATHFORMAT>Windows</PATHFORMAT>
          <FILEKIND>Text</FILEKIND>
          <FILEFLAGS>Debug</FILEFLAGS>
        </FILE>
...
          %<insertCWFiles(ModelSourceFiles)>
          %if DK_TLC_TYPE != "EXP"
          %<insertCWFiles(bspFiles)>
          %<insertCWFiles(CommAPIFiles)>
          %<insertCWFiles(CommFiles)>
          %endif
...
</FILELIST>
```

Tal y como se desprende de la estructura de etiquetado definida en la cabecera del archivo *XML*, la etiqueta <*FILELIST*> depende de <*PROJECT*> <*TARGETLIST*> <*TARGET*>. En este extracto hemos querido poner de manifiesto que la inclusión de los archivos que formarán parte de un proyecto *MCP* se puede llevar a cabo de dos formas. La primera de ellas es estática, realizándose siempre que se invoque a *RTW* con la plantilla *MPC555DK*. La segunda forma tiene en consideración la dependencia con los ficheros fuente asociados al proyecto *Simulink* con el que se estuviera trabajando; para ello se ha definido la función *insertCWFiles*[50], la cual se construye en *gen_xml_lib.tlc*.

El código que presenta dicha función es el siguiente:

```
%function insertCWFile (file) Output
  <FILE>
    <PATHTYPE>Name</PATHTYPE>
    <PATH>%<file></PATH>
    <PATHFORMAT>Windows</PATHFORMAT>
    <FILEKIND>%<FILEKIND></FILEKIND>
    <FILEFLAGS>%<FILEFLAGS></FILEFLAGS>
  </FILE> \
%endfunction
```

Esta función introduce la misma estructura de etiquetado que <*FILE*> sin más que ser dependiente del argumento *file*. Las variables globales *FILEKIND* y *FILEFLAGS* toman los valores *Text* y *Debug*, respectivamente; siendo definidas al final del archivo *gen_xml.tlc*.

[50] *InsertCWFiles*: del inglés, insertar archivos de *CodeWarrior* (*CW*).

- Definición del orden del proceso de enlazado o *link order*.

```
<LINKORDER>
        <FILEREF>
                <PATHTYPE>Name</PATHTYPE>
                <PATH>__ppc_eabi_init.c</PATH>
                <PATHFORMAT>Windows</PATHFORMAT>
        </FILEREF>
...
          %<insertCWFileRefs(ModelSourceFiles)>
          %if DK_TLC_TYPE != "EXP"
          %<insertCWFileRefs(bspFiles)>
          %<insertCWFileRefs(CommAPIFiles)>
          %<insertCWFileRefs(CommFiles)>
          %endif
...
         <FILEREF>
                <PATHTYPE>Name</PATHTYPE>
                %if DK_TLC_TYPE == "EXP"
                    <PATH>%<link_cmd_file></PATH>
                %else
                    <PATH>555_phytec_SRAM.lcf</PATH>
                %endif
               <PATHFORMAT>Windows</PATHFORMAT>
        </FILEREF>
</LINKORDER>
```

La etiqueta <*LINKORDER*> pertenece a <*PROJECT*> <*TARGETLIST*> <*TARGET*>. Dentro de ella se encuentra <*FILEREF*>, la cual se ha definido de tres formas diferentes por *Motorola*. La primera de ellas es estática, forzando a que el archivo *__ppc_eabi_init.c* siempre sea el primero en enlazarse. La segunda de ellas invoca a la función *insertCWFileRefs*, que se define en el archivo *gen_xml_lib.tlc*. Y, la última cambia según el modo de trabajo de la interfaz ideada por *Motorola*. Es decir, si se está utilizando el modo *EXP*[51] entonces el nombre del archivo a enlazar viene dado a través de la variable para *TLC* denominada *link_cmd_file*, en caso contrario –modos de trabajo *PIL*[52] o *RT*[53]– el fichero indicado se llama *555_phytec_SRAM.lcf*.

La segunda forma resulta interesante. Esto es, además de invocar a la función antes mencionada, hace uso de la variable para *TLC* de nombre *DK_TLC_TYPE*. En ella se recoge el modo de trabajo de la interfaz de *Motorola*. En cualquier caso, siempre esta forma inserta los códigos fuente asociados a los ficheros generados por *RTW –ModelSourceFiles* se define en *mpc555exp_genfiles.tlc* o en *mpc555pil_genfiles.tlc*–. En cambio, si el modo de trabajo no es *EXP* entonces se insertan los archivos referidos por las variables *bspFiles*, *CommAPIFiles* y *CommFiles*, tomando valores en el archivo *mpc555pil_genfiles.tlc*.

[51] *EXP*: *Algorithm Export*.
[52] *PIL*: *processor-in-the-loop*.
[53] *RT*: *real-time*.

Estas tres variables reciben los valores adecuados a través de las siguientes líneas:

```
%assign ModelSourceFiles   = FEVAL("rtwprivate","rtwattic","AtticData","SourceFiles")
...
%assign CommAPIFiles  = ["%<ModelName>_comm.c", "%<ModelName>_comm_c_api.c"]
%assign bspFiles    = ["mw_bsp.a", "uart_diab.a"]
%assign CommFiles    = ["CommAPI.h", "commapi.a"]
```

La variable *ModelSourceFiles* toma el valor devuelto por la función *FEVAL*, la cual evalúa la función cuyo nombre es su primer argumento, en este caso: *rtwprivate*. Esta función, según la ayuda proporcionada por *Matlab*, realiza una llamada a la estructura de datos relativa al proceso de construcción de los ficheros fuente *c* y librerías *h* que realiza *RTW* –recuérdese que dicha estructura recibe el nombre de *model.rtw*–, siendo *SourceFiles* la estructura que alberga los nombres de los archivos fuente generados.

Las otras variables llevan asociados los nombres de varios archivos que forman parte de la interfaz ideada por *Motorola* cuando se hace uso de *PIL* o de *RT*. Nótese que ciertos de estos archivos se crean durante el proceso de actuación de la interfaz, ya que su nombre depende de la variable *TLC* denotada por *ModelName* –siendo su valor una instancia de la clase *CompiledModel.Name*, es decir, el nombre del modelo *Simulink* base para *RTW*–.

La función *insertCWFileRefs* presenta el siguiente código:

```
%function insertCWFileRefs (fileList) Output
    %% Need to check for string type fileList - a single file.
    %if TYPE(fileList) == "String"
        %<insertCWFileRef(fileList)>
    %else
        %foreach idx = SIZE(fileList,1)
            %<insertCWFileRef(fileList[idx])>
        %endforeach
    %endif
%endfunction
```

En esta función se comienza por comprobar si su argumento se refiere a un único archivo o, por el contrario, hace referencia a un conjunto de ellos. Con este fin se hace uso de la función *TYPE* que nos proporciona el tipo de datos que es su argumento. Así pues, si se trata de una cadena o *string* entonces se debe a un único archivo; si no, se inserta cada uno de los archivos usando su índice mediante *foreach*[54] al invocar la función *insertCWFileRef*.

[54] *%foreach*: se trata de una directiva de inclusión múltiple que pertenece a *TLC*.

Veamos cómo funciona esta última función:

```
%function insertCWFileRef (file) Output
  <FILEREF>
    <PATHTYPE>Name</PATHTYPE>
    <PATH>%<file></PATH>
    <PATHFORMAT>Windows</PATHFORMAT>
  </FILEREF> \
%endfunction
```

En ella se incluye el etiquetado necesario para incluir una referencia de archivo dentro de *<LINKORDER>* dado el argumento *file*.

- Indicación del orden el que se trabaja con los diferentes sistemas objetivo o *targets* que soporta *CodeWarrior*.

```
<TARGETORDER>
      <ORDEREDTARGET><NAME>555 Phytec Debug Version</NAME></ORDEREDTARGET>
 </TARGETORDER>
```

Este entorno de desarrollo puede tener asociados a un mismo proyecto *MCP* varios sistemas objetivos. Éstos se presentan bajo un orden a la hora de aplicar el proceso de compilado y enlazado de los archivos que forman dicho proyecto. Pues bien, para el proyecto *XML*generado por la interfaz de *Motorola* se indica como único *target* el denominado por *CodeWarrior* como *555 Phytec Debug Version*.

- Agrupación de los nombres de los archivos fuente y librerías.

Puesto que en la presentación del listado de estos archivos por una ventana de proyecto *MCP* dentro de *CodeWarrior* se puede configurar por parte del usuario, esta especificación debe quedar definida dentro de dicho proyecto. Debido a esto se incluye, dentro de un proyecto *XML*, la agrupación de los diferentes archivos que lo forman. Esto es:

```
<GROUPLIST>
      <GROUP><NAME>Linker Command File</NAME>
        <FILEREF>
          <TARGETNAME>555 Phytec Debug Version</TARGETNAME>
          <PATHTYPE>Name</PATHTYPE>
          %if DK_TLC_TYPE == "EXP"
          <PATH>%<link_cmd_file></PATH>
          %else
          <PATH>555_phytec_SRAM.lcf</PATH>
          %endif
          <PATHFORMAT>Windows</PATHFORMAT>
        </FILEREF>
      </GROUP>
...
      <GROUP><NAME>RTWlibraries</NAME>
        <FILEREF>
          <TARGETNAME>555 Phytec Debug Version</TARGETNAME>
          <PATHTYPE>Name</PATHTYPE>
          <PATH>rtwlib.PPCEABI.H.a</PATH>
          <PATHFORMAT>Windows</PATHFORMAT>
```

```
        </FILEREF>
      </GROUP>

  %if DK_TLC_TYPE != "EXP"
      <GROUP><NAME>BSP Files</NAME>
            %<insertCWFileRef1s(bspFiles)>
      </GROUP>
      <GROUP><NAME>CommFiles</NAME>
            %<insertCWFileRef1s(CommFiles)>
      </GROUP>
      <GROUP><NAME>CommAPIFiles</NAME>
            %<insertCWFileRef1s(CommAPIFiles)>
      </GROUP>
  %endif

      <GROUP><NAME>ModelFiles</NAME>
            %<insertCWFileRef1s(ModelSourceFiles)>
      </GROUP>
</GROUPLIST>
```

Tal y como se observa, la etiqueta *<GROUPLIST>*[55] se constituye con *<GROUP>* y depende directamente de *<PROJECT>*. *Motorola* ha decidido, por cuestiones constructivas, que se definan tres formas de insertar las líneas *XML*. La primera de ellas consiste en *<FILEREF>* estáticos, que incluyen las librerías propias de la interfaz *MPC555DK*. La segunda forma depende del modo de trabajo de dicha interfaz, insertando código únicamente si no se trabaja con el modo *EXP* a través de la función *insertCWFileRef1s*. La tercera y última forma inserta el código asociado a las referencias a los archivos fuente; para ello se invoca a esa misma función con el argumento *ModelSourceFiles*.

Veamos cómo se constituye la función *insertCWFileRef1s*:

```
%function insertCWFileRef1s (fileList) Output
%% Need to check for string type fileList - a single file.
  %if TYPE(fileList) == "String"
    %<insertCWFileRef1(fileList)>
  %else
    %foreach idx = SIZE(fileList,1)
      %<insertCWFileRef1(fileList[idx])>
    %endforeach
  %endif
%endfunction
```

Esta función presenta una forma prácticamente idéntica a la que recibe el nombre *insertCWFileRefs*, salvo que invoca a *insertCWFileRef1* en vez de a *insertCWFileRef*. La función llamada posee el siguiente código:

```
%function insertCWFileRef1 (file) Output
      <FILEREF>
        <TARGETNAME>555 Phytec Debug Version</TARGETNAME>
        <PATHTYPE>Name</PATHTYPE>
        <PATH>%<file></PATH>
        <PATHFORMAT>Windows</PATHFORMAT>
      </FILEREF> \
%endfunction
```

[55] *Grouplist*: del inglés, lista de grupos.

Su objetivo es idéntico al de la función *insertCWFileRef*, es decir, también introduce las líneas *XML* asociadas a una etiqueta <*FILEREF*>. Pero, debido a la estructura interna de una de estas etiquetas definida en la cabecera del documento *XML* ésta puede presentar más de una forma. La sentencia que indica los diferentes formatos que admite esta etiqueta es:

```
<!ELEMENT FILEREF (TARGETNAME?, PATHTYPE, PATHROOT?, ACCESSPATH?, PATH, ...
        PATHFORMAT?)>
```

Como puede apreciarse, algunas de las etiquetas que forman un elemento <*FILEREF*> son opcionales, ya que llevan el añadido *'?'*. Esto se desprende de la documentación sobre *XML* que podemos encontrar en el portal en Internet de *Sun Microsystems*. Así pues, cuando se incluyen archivos en la etiqueta <*LINKORDER*> se toman unas opciones diferentes para <*FILEREF*> que cuando se incluyen en <*GROUP*>.

- Para finalizar:

```
</PROJECT>
%%  %closefile XMLProjectFile
%%  %selectfile NULL_FILE

%endfunction %% CreateXMLProject

%%assign FILEFLAGS = "Debug" | ""
%%assign FILEKIND  = "Text" | "Unknown"
%assign FILEFLAGS = "Debug"
%assign FILEKIND  = "Text"

%%<CreateXMLProject("blank.txt")>

%% end file
```

  - En donde:
    - *</PROJECT>* cierra el documento *XML*.
    - *%endfunction* indica que se ha terminado la definición del cuerpo de la función en curso, es decir, de *CreateXMLProject*.
    - Se definen las variables *FILEFLAGS* y *FILEKIND* a través de la primitiva *assign*.
    - Y se invoca a la función que se acaba por definir dando el nombre del fichero destinatario del proyecto *XML*, sea *black.txt*.

3.6.6.2 Aplicación a la interfaz para ARM7TDMI.

Para construir la interfaz adecuada, en lo concerniente a la generación del fichero *XML* al sistema objetivo de *ATMEL* se debe trasladar el formato de la estructura de etiquetado que presenta un proyecto *XML* para *DebugRel*. Éste es el nombre con el que *CodeWarrior* denota al sistema objetivo en cuestión.

Para ello se hace uso de la herramienta que ofrece el propio entorno de desarrollo de *MetroWerks* denominada *Compare Files...* Ésta posibilita la visualización comparada de dos proyectos para *CodeWarrior*, estando ambos expresados bajo el formato *XML*. Asimismo, se debe considerar la manera en la que el *TLC* es invocada desde la plantilla existente que genera el proyecto *XML* adecuada para el sistema objetivo de *Motorola*, es decir, para el fichero *gen_xml.tlc*. Estos dos puntos ya han sido expuestos en puntos anteriores.

Con la intención de trabajar sin posibles pérdidas de documentación, el fichero original *gen_xml.tlc* se pasó a denominar como *gen_xml_mpc555.tlc*, almacenándose en el mismo subdirectorio. Para trabajar con un nombrado de ficheros coherente con la interfaz, el fichero *gen_xml_lib.tlc* se copió como *gen_xml_lib_debug.tlc*. Posteriormente, se modificó el documento original *gen_xml.tlc*; de esta forma se asegura que la interfaz ideada por *Motorola* genera el proyecto *XML* adecuado para el sistema objetivo de *ATMEL*.

Para comenzar se eliminó del archivo *gen_xml.tlc* toda la información relativa a la estructura de etiquetado y valores de las etiquetas propias de un proyecto *XML* para *555 Phytec Debug Version*. Gracias a la herramienta *Note Pad*[56] de los accesorios de *Windows XP* se insertó el código fuente, asociado a un proyecto *XML* generado por *CodeWarrior*, al fichero cuyo nombre es *gen_xml.tlc*. Paralelamente, se trabajó con el archivo *gen_xml_mpc555.tlc*, en donde se observaban las líneas de código *TLC* insertadas a lo largo del código *XML*, tal y como se ha mostrado en el punto anterior.

Destaquemos las modificaciones de más interés que se aplicaron a los ficheros *gen_xml.tlc* y a *gen_xml_lib_debug.tlc*.

- Cambio de librería de funciones:

```
%include "gen_xml_lib_debug.tlc"
```

[56] Note Pad: del inglés, Bloc de Notas.

De esta forma, en lugar de incluir el archivo *tlc*de librerías *gen_xml_lib.tlc*, se tendrá como referencia la definición de las funciones que en *gen_xml_lib_debug.tlc*. Hemos de usar otro fichero de librería porque la estructura de etiquetado que presenta un proyecto *XMI*para *DebugRel* difiere lo suficiente respecto al caso de trabajar con *555 Phytec Debug Version*.

- Definición de las rutas de acceso:

```
<SETTING><NAME>UserSearchPaths</NAME>
...
  <SETTING>
    <SETTING><NAME>SearchPath</NAME>
      <SETTING>
        <NAME>Path</NAME>
        <VALUE>%<MODEL_ABS_PATH></VALUE>
      </SETTING>
      <SETTING><NAME>PathFormat</NAME><VALUE>Windows</VALUE></SETTING>
      <SETTING><NAME>PathRoot</NAME><VALUE>Project</VALUE></SETTING>
    </SETTING>
...
  </SETTING>
  <SETTING>
    <SETTING><NAME>SearchPath</NAME>
      <SETTING>
        <NAME>Path</NAME>
        <VALUE>%<MATLABROOT>\extern\include</VALUE>
      </SETTING>
      <SETTING><NAME>PathFormat</NAME><VALUE>Windows</VALUE></SETTING>
      <SETTING><NAME>PathRoot</NAME><VALUE>Project</VALUE></SETTING>
    </SETTING>
...
  </SETTING>
...
</SETTING>
```

Se retiraron todas las inclusiones de rutas relativas a las librerías propias del sistema objetivo de *Motorola* y se mantuvieron las que hacían referencia al directorio que almacena el código fuente, apuntado por la etiqueta para *TLC MODEL_ABS_PATH*, y a los que albergan las librerías de referencia estática.

- Inclusión de los archivos que contienen el código generado:

```
<FILELIST>
        %%Insert RTW generated source and librarian model files
        %<insertCWFiles(SourceFiles)>
        %<insertCWFiles(LibrarianFiles)>
    <FILE>
      <PATHTYPE>Name</PATHTYPE>
      <PATH>tmwtypes.h</PATH>
      <PATHFORMAT>Windows</PATHFORMAT>
      <FILEKIND>Text</FILEKIND>
      <FILEFLAGS></FILEFLAGS>
    </FILE>
    <FILE>
      <PATHTYPE>Name</PATHTYPE>
      <PATH>simstruc_types.h</PATH>
      <PATHFORMAT>Windows</PATHFORMAT>
      <FILEKIND>Text</FILEKIND>
```

```
            <FILEFLAGS></FILEFLAGS>
          </FILE>
          <FILE>
            <PATHTYPE>Name</PATHTYPE>
            <PATH>rtlibsrc.h</PATH>
            <PATHFORMAT>Windows</PATHFORMAT>
            <FILEKIND>Text</FILEKIND>
            <FILEFLAGS></FILEFLAGS>
          </FILE>
</FILELIST>
```

Respecto a la estructura de presentaba ante la interfaz de *Motorola*, las librerías de referencia estática se incluían mediante el uso de funciones específicas. Además, estas dependencias no se referían a los mismos archivos debido a las diferencias en funcionalidad del sistema objetivo *MPC555* y el de *ATMEL*.

Por otro lado, la inclusión de todos los archivos fuente *c* y las librerías locales *h* que han sido generados por *RTW* se lleva a cabo a través de la función *insertCWFiles*, la cual ha recibido como argumento las variables de *TLC* denominadas *SourceFiles* y *LibrarianFiles*. Ambas se definen en el archivo *mpc555exp_genfiles.tlc*[57], que ha sido modificado para ello con la inclusión de estas líneas de código *TLC*:

```
%% Get SourceFiles list from the rtwattic.
%assign ModelSourceFiles   = FEVAL("rtwprivate","rtwattic","AtticData","SourceFiles")

%% Get source files from the model's directory
%assign SourceFiles = FEVAL("dir","%<MODEL_ABS_PATH>%<PATH_SEP>*.c")
%% Get librarian files from the model's directory
%assign LibrarianFiles = FEVAL("dir","%<MODEL_ABS_PATH>%<PATH_SEP>*.h")

%matlab assignin("base","importXML", "%<MODEL_ABS_PATH>\\%<ModelName>.xml")
%matlab assignin("base","outputProject", "%<MODEL_ABS_PATH>\\%<ModelName>.mcp")
```

Con las líneas en negrita se asocia a las variables previamente mencionadas los nombres de archivo que contienen el código fuente –en la variable *SourceFiles*– y los que contienen a las librerías generadas –en la variable *LibrarianFiles*–.

- Definición del orden de enlazado:

```
<LINKORDER>
   <FILEREF>
      <PATHTYPE>Name</PATHTYPE>
       <PATH>tmwtypes.h</PATH>
       <PATHFORMAT>Windows</PATHFORMAT>
   </FILEREF>
   <FILEREF>
       <PATHTYPE>Name</PATHTYPE>
       <PATH>simstruc_types.h</PATH>
       <PATHFORMAT>Windows</PATHFORMAT>
   </FILEREF>
   <FILEREF>
       <PATHTYPE>Name</PATHTYPE>
       <PATH>rtlibsrc.h</PATH>
```

[57] Se encuentra en la ruta de directorio: ...\MATLAB\toolbox\rtw\targets\mpc555dk\pil

```
        <PATHFORMAT>Windows</PATHFORMAT>
    </FILEREF>
            %%Insert source and librarian files references
            %<insertCWFileRefs(LibrarianFiles)>
            %<insertCWFileRefs(SourceFiles)>
</LINKORDER>
```

Así pues; primero se indica que se consideren en el proceso de enlazado las librerías cuya referencia es estática; posteriormente, las que se han generado para el modelo concreto de *Simulink*, y, finalmente, los archivos que albergan el código fuente obtenido por *RTW*.

- Definición del sistema objetivo:

```
<PROJECT>
    <TARGETLIST>
        <TARGET>
            <NAME>DebugRel</NAME>
...
        </TARGET>
    </TARGETLIST>
    <TARGETORDER>
        <ORDEREDTARGET><NAME>DebugRel</NAME></ORDEREDTARGET>
    </TARGETORDER>
...
</PROJECT>
```

Dentro de la etiqueta <*TARGETLIST*> se especifica una lista de sistemas objetivos para los que se construye el proyecto *XML*. De forma similar, dentro de <*TARGETORDER*> se indica el orden de trabajo de cada uno de estos sistemas dentro del proyecto completo. Tal y como se desprende de las líneas de código se define, como único sistema objetivo, a *DebugRel*.

- Definición de los grupos de los archivos de un proyecto para *CodeWarrior*:

```
<GROUPLIST>
    <GROUP><NAME>Model Libraries</NAME>
            %insertCWFileRef1s(LibrarianFiles)>
    </GROUP>
    <GROUP><NAME>Model Source Files</NAME>
            %insertCWFileRef1s(SourceFiles)>
    </GROUP>
    <GROUP><NAME>Static-Dependent Libraries</NAME>
        <FILEREF>
            <TARGETNAME>DebugRel</TARGETNAME>
            <PATHTYPE>Name</PATHTYPE>
            <PATH>tmwtypes.h</PATH>
            <PATHFORMAT>Windows</PATHFORMAT>
        </FILEREF>
        <FILEREF>
            <TARGETNAME>DebugRel</TARGETNAME>
            <PATHTYPE>Name</PATHTYPE>
            <PATH>simstruc_types.h</PATH>
            <PATHFORMAT>Windows</PATHFORMAT>
        </FILEREF>
        <FILEREF>
            <TARGETNAME>DebugRel</TARGETNAME>
            <PATHTYPE>Name</PATHTYPE>
            <PATH>rtlibsrc.h</PATH>
            <PATHFORMAT>Windows</PATHFORMAT>
```

```
        </FILEREF>
    </GROUP>
</GROUPLIST>
```

Así pues, para incluir las referencias de los archivos fuente y librerías generados por *RTW* se utiliza la función *insertCWFileRef1s*, la cual se ha modificado respecto a su contenido original para que la etiqueta <*TARGETNAME*> contenida en <*FILEREF*> tome el valor *DebugRel*. La función *insertCWFileRef1s* se mantiene idéntica a su versión original, luego invoca a la función *insertCWFileRef1*. En cambio, ésta sí ha sufrido modificaciones, estando contenida la versión adecuada para el sistema objetivo de *ATMEL* en el archivo *gen_xml_lib_debug.tlc*; esto es:

```
%function insertCWFileRef1 (file) Output
    <FILEREF>
      <TARGETNAME>DebugRel</TARGETNAME>
      <PATHTYPE>Name</PATHTYPE>
      <PATH>%<file></PATH>
      <PATHFORMAT>Windows</PATHFORMAT>
    </FILEREF> \
%endfunction
```

Como podemos observar lo que cambia es el valor que toma el campo de la etiqueta <*TARGETNAME*>.

3.7 Diagrama de Ejemplo: Ganancia2.

El objetivo fundamental de este apartado es la ejemplificación de los contenidos de los puntos anteriores. Para ello se plantea un modelo *Simulink* muy sencillo sobre el que aplicar los algoritmos de *Real Time Workshop Embedded Coder* para obtener el código fuente correspondiente; éste será analizado comparándolo con el esquema general planteado con anterioridad, buscando las funciones que realizan las iteraciones del modelo; posteriormente se obtendrá el código asociado un proyecto de *MetroWerks CodeWarrior* con el fin de ejecutarlo en la placa de *ATMEL*.

3.7.1 Diagrama de *Simulink*.

El ejemplo que vamos a utilizar es muy sencillo de forma que su análisis ayude en la clarificación de los contenidos de los apartados expuestos previamente. Así pues, el modelo parte de una secuencia sinusoidal, la cual será multiplicada por dos en amplitud por un bloque intermedio y, finalmente, será expuesto por pantalla el resultado. El diagrama de *Simulink* asociado a dicho modelo presenta la siguiente forma:

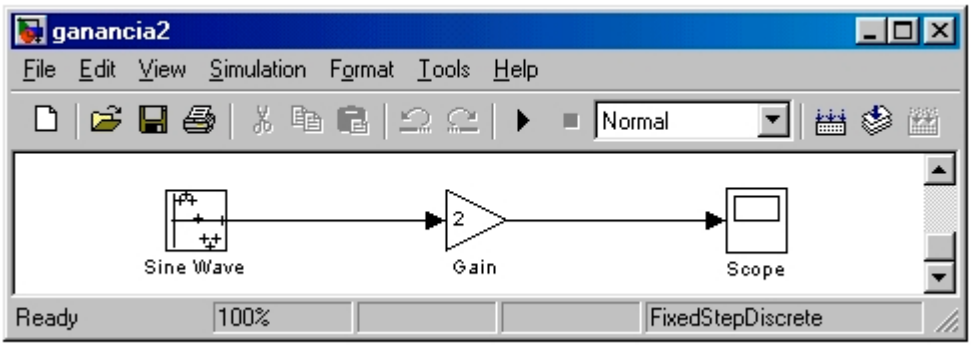


Figura: Diagrama *Simulink* del modelo ejemplo.

3.7.2 Programación de Opciones.

La programación de las opciones se lleva a cabo mediante el cuadro de diálogo *Simulation Parameters*, el cual se desprende del menú *Simulation* o del menú *Tools*, a través del campo *Options...* de la opción *Real Time Workshop...*, de la figura anterior.

Agrupemos los parámetros especificados en una lista:

- La configuración del sistema escogido se define a través de la plantilla *ert_default_tmf* y de la librería *ert.tlc*, de esta forma *RTW* generará código fuente bajo el formato *Embedded-C*, lo que da lugar a la actuación del codificador embebido[58].

- Puesto que únicamente pretendemos generar el código fuente, se optará por habilitar la opción *Generate code only*. Nótese que la compilación a código máquina se llevará a cabo por la aplicación *MetroWerks CodeWarrior*, la cual no será invocada por *RTW*.

- Respecto al integrador se seleccionará *ode4.c*, que implementa un algoritmo basado en el método de *Runge-Kutta*, dentro del tipo de paso de iteración fijo o *FixedStep*.

- Dentro de las opciones propias al estilo de codificación *ERT* se marcará la generación de un ejemplo de función *main* característico para sistemas formados por una placa simple o, usando la terminología técnica de *Matlab*, *BareBoardExample*.

- Para evitar problemas con dispositivos externos de almacenamiento se ha anulado la posibilidad de generar un registro, dentro de un fichero *mat*, de la evolución de las variables y estados de los diferentes bloques del diagrama. Implícitamente, esta decisión fuerza a que se trabaje con horizonte infinito.

- No se fuerza a que se trabaje siempre con formato de codificación numérica *integer*[59], puesto que la máquina de *ARM* es compatible con el tipo *float*[60] de la normativa *ANSI C*. Debido a esto se especifica, en las opciones, la compatibilidad del código generado con la codificación matemática flotante de este *standard*, en lugar de la otra opción posible: *ISO C*.

- Dentro de la lengüeta de opciones avanzadas se especifica el tamaño, en bits, que tendrá cada formato de codificación en la máquina de destino. De acuerdo con la documentación aportada por el fabricante, se define que:

    - Para tipo carácter (*char*): se usarán 8 bits.
    - Para tipo corto (*short*): se usarán 16 bits.
    - Para tipo entero (*int*): se usarán 32 bits.
    - Para tipo largo (*long*): se usarán 32 bits.

[58] El codificador embebido es *Real Time Workshop Embedded Coder*.
[59] *Integer*: del inglés, entero; mediante codificación en complemento *A2*.
[60] *Float*: del inglés, flotante; muy similar a la codificación científica, pero sin parte entera.

- Con la intención de facilitar la compresión del código fuente generado y sus interdependencias, se indica que se redacte un informe *HTML* del proceso, el cual se muestra automáticamente tras éste si ha completado satisfactoriamente.

3.7.3 Análisis del Código generado *C++*.

Tras la generación del código fuente *C++*, se puede estudiar el informe facilitado por *RTW*. Entre la información relevante que se nos muestra podremos destacar la localización de cada uno de los ficheros obtenidos y de sus dependencias estáticas; por otra parte, también cabe reseñar la indicación de cuáles han sido las optimizaciones que se han aplicado al código. Para estudiar el informe, consulte el apartado *6.1.1 Informe sobre la generación de código para el modelo ganancia2.mdl*, localizado dentro del anexo.

Además, formado parte de este informe, se incluye un conjunto de referencias a los códigos fuente generados; incluso, dentro de ellos, se añaden enlaces a las definiciones de librería de las variables y funciones que en el código se utilizan. Esto facilita enormemente la legibilidad y comprensión de las líneas de código y su funcionalidad.

Cabe destacar aquí dos ficheros. El primero de ellos –*ert_main.c*– constituye el ejemplo de programa principal, el cual se marcó como opción. El segundo –*ganancia2.c*– contiene las funciones básicas que, para el estilo de generación de código fuente *Embedded-C*, ejecutan el modelo. Todos los archivos que contienen código fuente pueden consultarse en el anexo *6.1.2 Archivos de códigos fuente generados para ganancia2.mdl*, salvo las librerías por carecer del interés suficiente.

- Respecto a *ert_main.c* hemos señalado las siguientes líneas de código:

Al inicio de la función de gestión de la interrupción –*rt_OneStep()*– se indica donde se ha de inhabilitar las interrupciones y comprobar que se ha realizado el evento adecuado, es decir, el generado por el reloj, para esto último se valida la bandera *OverrunFlag*.

```
39  void rt_OneStep(void)
40  {
41    /* Disable interrupts here */
42
43    /* Check for overrun */
44    if (OverrunFlag++) {
45      rtmSetErrorStatus(ganancia2_M, Overrun);
46      return;
47    }
```

Se invoca a la función que lleva a cabo un paso en la iteración del modelo, siendo *ganancia2_step()*, para bloquear la bandera tras ella y, finalmente, se solicita la habilitación de las interrupciones antes de terminar la *ISR*.

```
53    ganancia2_step();
54
55    /* Get model outputs here */
56
57    OverrunFlag--;
58
59    /* Disable interrupts here */
60    /* Restore FPU context here (if necessary) */
61    /* Enable interrupts here */
62    }
```

Respecto a la función principal, en ella se señalan las líneas de código en las que debe incluirse el generador de eventos en tiempo real tras la llamada a la función de inicialización del modelo –*ganancia2_initialize(1)*–. Una vez finalizado el proceso de simulación, se invoca a la función que destruye el modelo, sea *ganancia2_terminate()*.

```
69    int_T main(int_T argc, const char_T *argv[])
70    {
...
76    /* Initialize model */
77    ganancia2_initialize(1);
78
79    /* Attach rt_OneStep to a timer or interrupt service routine with
80     * period 0.02 seconds (the model's base sample time) here.  The
81     * call syntax for rt_OneStep is
82     *
83     * rt_OneStep();
84     */
85
86    while (rtmGetErrorStatus(ganancia2_M) == NULL) {
87      /* Perform other application tasks here */
88    }
89
90    /* Disable rt_OneStep() here */
91
92    /* Terminate model */
93    ganancia2_terminate();
94    return 0;
95    }
```

- Respecto a *ganancia2.c* hemos destacado el siguiente código:

La función que lleva a cabo la realización de cada iteración del modelo se denomina *ganancia2_step()*.

```
22    void ganancia2_step(void)
23    {
```

Primero se definen las variables locales donde se apuntarán los valores de *E/S* de cada bloque del diagrama. Así pues, *rtb_temp0* será una variable temporal asociada a la salida del bloque cero, es decir el generador de la secuencia sinusoidal.

```
24   /* local block i/o variables */
25   real_T rtb_temp0;
```

En dicha variable se almacena la muestra generada en cada iteración que, de acuerdo con la notación utilizada, sería coherente con la siguiente expresión formal:

$$u(n) = A \cdot sin\left(2.0 \times \pi \times \frac{n + n_0}{N}\right) + A_0;$$

En donde se puede identificar:

- *'u(n)'* con la variable *rtb_te*mp0, es decir, en ella se almacena la muestra generada en cada paso, tal y como ya se ha indicado.
- *'A'* con la expresión *ganancia2_P.Sine_Wave_Amp* que, más claramente, expresa la amplitud del bloque *Sine_Wave* dentro del modelo ganancia2.
- *'p'* con la constante *RT_PI*, siendo el acrónimo del número π para tiempo real.
- *'n'* con la expresión *ganancia2_Dwork.Sine_Wave_IWORK.Counter*, la cual contiene el contador de iteraciones de la simulación.
- *'n_0'* coincide con el valor *ganancia2_P.Sine_Wave_Offset*.
- *'N'*, con la expresión *ganancia2_P.Sine_Wave_NumSamp*, la cual contiene el número de muestras[61] relativo a la periodicidad de la secuencia sinusoidal.
- *'A_0'*, con la constante *ganancia2_P.Sine_Wave_Bias*, la cual depende del parámetro *Bias*[62] del bloque *Sine_Wave*.

```
27   /* Sin: '<Root>/Sine Wave' */
28   /* Sample Based Sine Wave Output Function */
29
30   rtb_temp0 = ganancia2_P.Sine_Wave_Amp
31     * sin( 2.0 * RT_PI*( ganancia2_DWork.Sine_Wave_IWORK.Counter +
32     ganancia2_P.Sine_Wave_Offset)/ ganancia2_P.Sine_Wave_NumSamp )
33     + ganancia2_P.Sine_Wave_Bias;
```

Con referencia al segundo bloque, el cual da lugar a la ganancia del diagrama, únicamente haremos mención a que se almacena el valor de ganancia de dicho bloque, el cual está definido en la constante *ganancia2_P.Gain_Gain*. Nótese que el nombre de este bloque, en inglés, es *Gain –Ganancia*, en español–.

```
35   /* Gain: '<Root>/Gain'
36    *
37    * Regarding '<Root>/Gain':
38    *   Gain value: ganancia2_P.Gain_Gain
39    */
40   rtb_temp0 *= ganancia2_P.Gain_Gain;
```

[61] *NumSamp* forma el acronismo *Number of Samples*.
[62] *Bias*: del inglés, predisposición; desde un punto de vista técnico: valor de cero o de reposo.

Se actualiza el valor del contador que se usa en la generación de la secuencia sinusoidal.

```
42   /* Sin Block: <Root>/Sine Wave */
43   ganancia2_DWork.Sine_Wave_IWORK.Counter =
44   ganancia2_DWork.Sine_Wave_IWORK.Counter + 1;
```

También se comprueba si éste alcanza su valor máximo, dado por el número de muestras que constituye un ciclo de dicha secuencia. En tal caso, se devuelve a su valor inicial, es decir, cero.

```
45   if ((ganancia2_DWork.Sine_Wave_IWORK.Counter) ==
46    (ganancia2_P.Sine_Wave_NumSamp)) {
47     ganancia2_DWork.Sine_Wave_IWORK.Counter = 0;
48   }
```

Las constantes y las expresiones referidas a parámetros de los distintos bloques se definen en las librería formando estructuras –*struct*– en el lenguaje *C++* jerarquizadas; para más información véase el anexo correspondiente.

3.1.4 Proyecto de MetroWerks CodeWarrior.

Ante todo proyecto deben estar ordenados los archivos fuente y las librerías que se usan al mismo, pero de forma que no existan constantes, tipos de datos o funciones por definir durante el proceso de compilador y enlazado.

El orden correcto es el que se muestra en la siguiente figura:

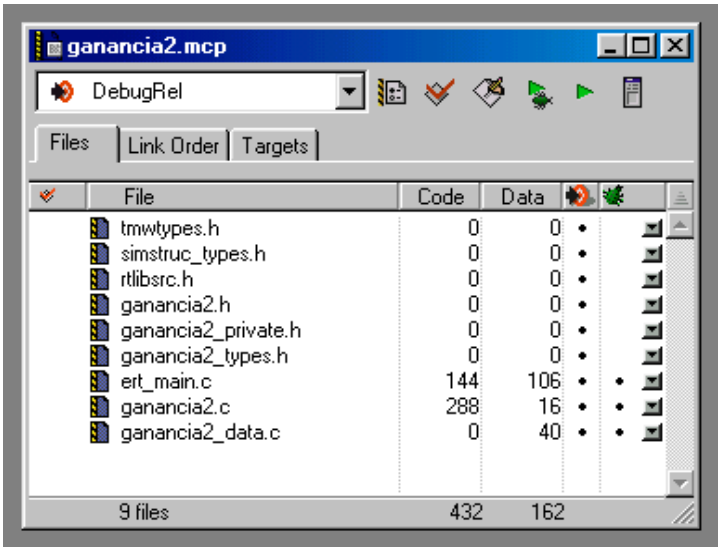


Figura: Orden de los archivos generados.

El proceso de compilado y enlazado se realiza según el orden descendente, es decir, comenzando por los archivos situados en la parte superior de la lista y prosiguiendo con el resto, avanzando por ella hasta el fichero inferior. Tras este proceso se obtienen líneas asociadas a código y líneas relativas a datos, siempre dentro de los archivos que contienen código fuente, o sea, especificaciones de procedimientos en lenguaje *C++*. En las librerías se definen las constantes, las macros y los tipos de datos, pero no albergan código fuente.

De la lista, los tres primeros ficheros constituyen la dependencia estática del código generado por *RTW*; los tres siguientes, las librerías específicas del modelo *Simulink* y, por último, los tres últimos contienen el código fuente. Este orden es coherente con las definiciones, puesto que dado un fichero se usan las establecidas en alguno de los archivos anteriores.

Desde la ventana anterior se puede iniciar el proceso de descarga y ejecución sin más que pulsar con el ratón sobre el botón triangular verde; claro que gracias a la invocación desde *RTW* se puede llevar a cabo de forma automatizada. Previamente a este proceso se debe conectar y configurar adecuadamente el sistema objetivo, tal y como se explicó en el capítulo correspondiente.

De acuerdo con el informe generado tras la compilación por *CodeWarrior*, se genera un total de 26756 bytes de código y 321 bytes de datos.