Capítulo 2

Code Composer Studio

Texas Instruments provee un software llamado *Code Composer Studio (CCS)* que es un entorno de desarrollo diseñado para programar las diferentes familias de dispositivos que fabrica. Este software se puede instalar en diferentes sistemas operativos, tales como Windows, Linux y Mac OS. Sin embargo, los ejemplos realizados en el presente libro se desarrollaron utilizando la versión 7 del CCS[1] instalada una distribución del sistema operativo Linux, Ubuntu 16.04. CCS puede descargarse desde la página oficial de la empresa *Texas Instruments*, por medio del siguiente link *http://www.ti.com/tool/CCSTUDIO*.

Texas Instrumets tiene dos tipos de descargas para ejecutar la instalación del software: instalador en línea e instalador fuera de línea. La instalación en línea todos los archivos necesarios para instalar en el computador, por lo que es necesario tener buena conexión a la red mientras se ejecuta la instalación. El instalador fuera de línea contiene dichos archivos y bibliotecas para instalar, sin embargo, es muy importante seleccionar el o los dispositivos que se van a programar porque el software requiere los controladores de las tarjetas a programar para que sean reconocidas, y de la misma manera realizar una correcta ejecución del programa creado en el dispositivo.

Antes de instalar CCS en Linux, puede ser necesario instalar la biblioteca *libc6-i384*, esto se puede saber al iniciar el asistente de instalación si aparece una ventana emergente indi-

[1] Software de ambiente de desarrollo gratis, de la compañía Texas Instruments

cando un error por falta de la biblioteca mencionada. La instalación de dicha biblioteca, en Ubuntu, se hace al ejecutar la siguiente línea desde la terminal *sudo apt-get install libc6-i384*. Una vez instalada, el instalador del *Code Composer Studio* continuará con la instalación del software.

La comunicación entre los dispositivos embebidos en tarjetas de desarrollo de TI y CCS se mantiene mediante una interfaz serial denominada como XDS, cuyo soporte en los controladores JTAG permite embeber el hardware en la tarjeta de desarrollo, utilzando un puerto UART para conectarse a la PC. Dicha comunicación permite acceder en tiempo real al contenido de la memoria de los dispositivos, visualizar la información de los registros de la arquitectura del dispositivo, contabilizar ciclos de reloj entre dos segmentos de código, graficar segmentos de memoria, cargar programas, hacer sesiones para depurar o probar un código instrucción por instrucción, entre otras cosas.

En el presente capítulo se explican las características del entorno de desarrollo *Code Composer Studio*, así como la creación de nuevos proyectos, contenido de un programa en lenguaje ensamblador, compilación y ejecución en la tarjeta utilizada.

2.1. Creación de proyectos en Code Composer Studio

Cuando se ejecuta el software CCS, aparece una ventana como la que se muestra en la Figura 2.1, en la cual se solicita seleccionar un directorio de trabajo. En dicho directorio se van a alojar todos los proyectos generados por medio de ficheros con el nombre de cada proyecto.

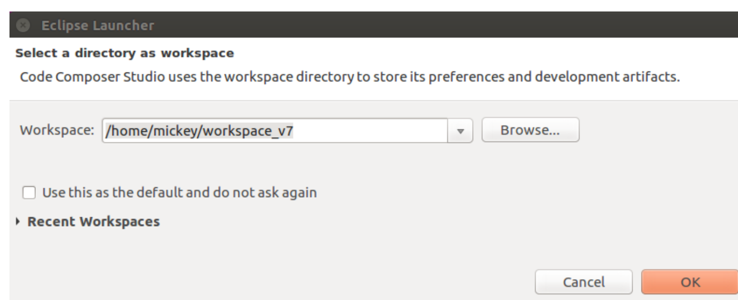


Figura 2.1: Selección del directorio de trabajo.

Posteriormente se muestra la ventana principal de *Code Composer Studio* donde se muestran botones para crear un nuevo proyecto, buscar ejemplos instalados, importar proyectos, etc. Para crear un nuevo proyecto se debe seleccionar el icono *New Project* (o bien

File→New→CSS Project), como se muestra en la Figura 2.2

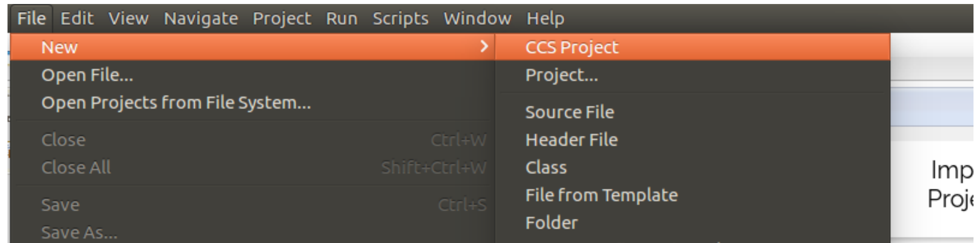


Figura 2.2: Pantalla de inicio

Es necesario configurar el proyecto con las características del sistema embebido a utilizar (microcontrolador o DSP), dichas características se ingresan en la ventana que aparece a continuación (ver Figura 2.3). A lo largo del presente manual, se desarrollaron ejemplos de programas implementados en el DSP TMS320F28377S, de tal manera que la configuración realizada en la Figura 2.3 contiene la información para dicha tarjeta.

La ventana mostrada en la Figura 2.3 se puede dividir en tres secciones, la primera es la mostrada dentro del rectángulo resaltado, en ella se seleccionan los datos relacionados de la tarjeta a utilizar y el nombre del proyecto, en este caso se seleccionó la tarjeta *2837xS Delfino* de la familia *TMS320F28377S* y el tipo de conexión *XDS100v2* por USB. El botón *Verify...* que se encuentra a la derecha del campo *Connection* genera una prueba de comunicación entre CCS y la tarjeta. Al conectar la tarjeta con el cable USB proporcionado en el Kit de evaluación y presionar el botón de *Verify...*, aparecerá una ventana emergente. Si la prueba fue exitosa, la ventana mostrará hasta el final del reporte un mensaje como el que se muestra en la Figura 2.4.

La segunda sección de la ventana de configuración del proyecto (Figura 2.3) es de ajustes avanzados (*Advanced settings*), al dar un clik en esta sección se despliega una ventana como se muestra en la Figura 2.5a. En esta sección se puede agregar un archivo con extension *cmd* en el apartado *Linker command file*, esto se logra por medio del botón *Browse...* y seleccionando la ruta en donde se encuentre dicho archivo.

El documento con extensión *cmd* es un archivo enlazador de comandos que describe el nombre, la localización y el tamaño del mapa de memoria del dispositivo, en la Sección 2.3 se explica con mayor detalle el contenido y declaración del archivo *cmd*. En el Apéndice A se expone el archivo *cmd* diseñado para las aplicaciones desarrolladas en el presente trabajo, mismo que se puede utilizar para desarrollar los ejercicios propuestos.

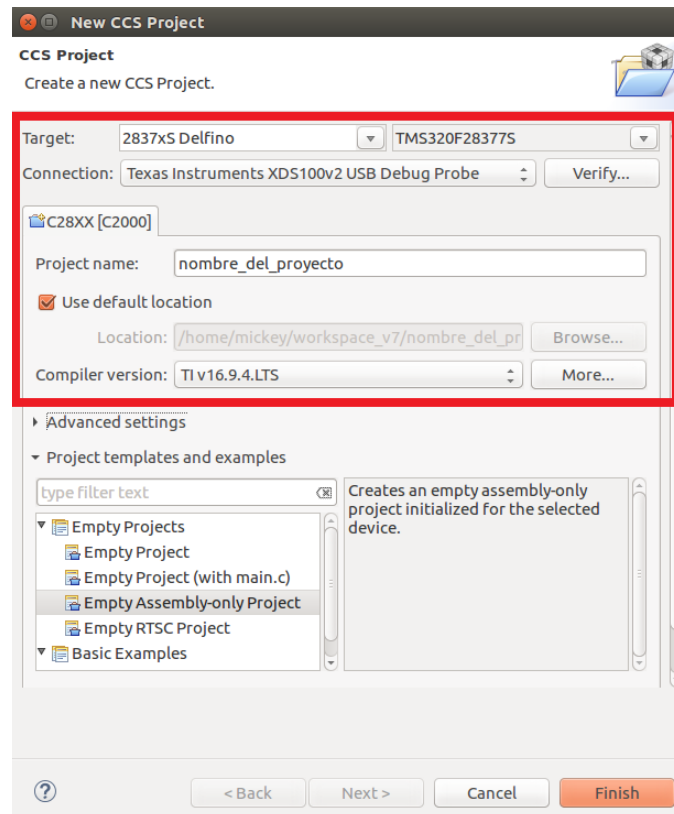


Figura 2.3: Nuevo proyecto con la tarjeta *LaunchPad Delfino*

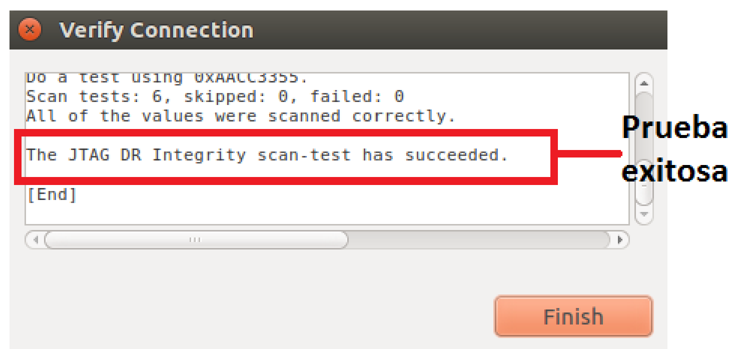
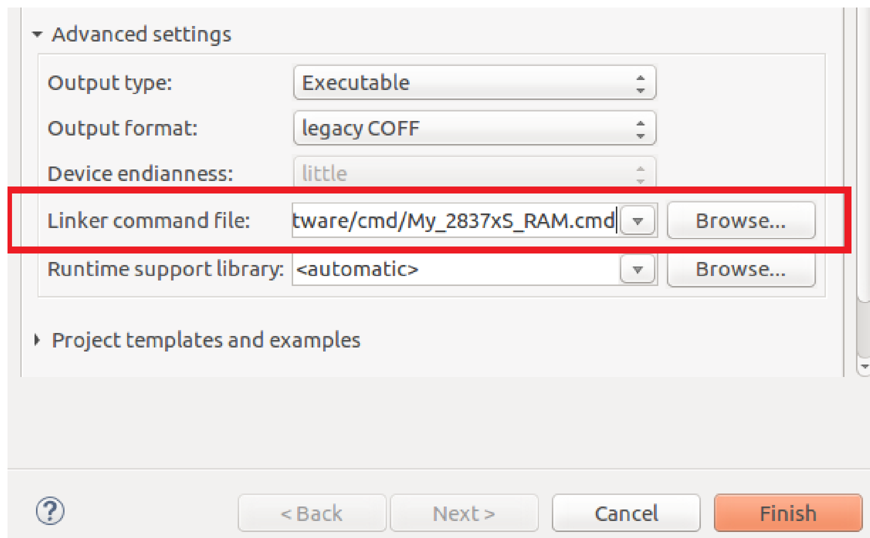
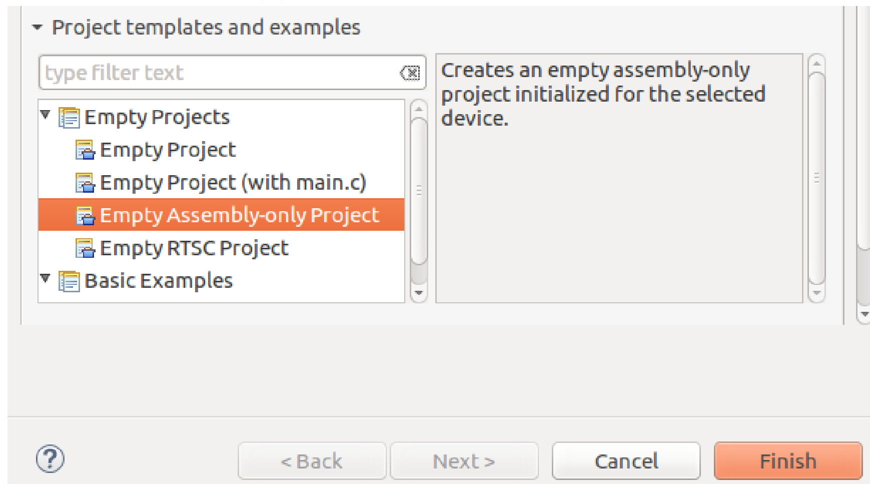


Figura 2.4: Prueba de conexión exitosa

Por último, en la sección de *Project templates and examples* se selecciona el tipo de proyecto, éste puede ser en lenguaje C o ensamblador entre otros. Los ejemplos de este manual se encuentran escritos en lenguaje ensamblador, por lo que se selecciona *Empty Assembly-only*



(a) Selección del archivo cmd.



(b) Selección del tipo de proyecto.

Figura 2.5: Selección del archivo cmd y tipo de proyecto.

Project, como se muestra en la Figura 2.5b.

Si se desea generar un proyecto para programar en lenguaje C, solo basta con seleccionar la opción *Empty Project (with main.c)* en la sección *Project temprates and examples* (ver Figura 2.5b) y automáticamente genera el archivo editor de texto con extensión *C* que contiene la función principal (*main*).

Despúes de crear el proyecto, la interfaz cambiará su apariencia y se verá como se muestra en la Figura 2.6, esta ventana se conoce como *CCS Edit* y es en donde se podrá crear, cargar

y editar un proyecto. Se pueden observar dos secciones dentro de la interfaz: el explorador de proyectos que se encuentra en la parte izquierda y la ventana de notificaciones en la parte inferior.

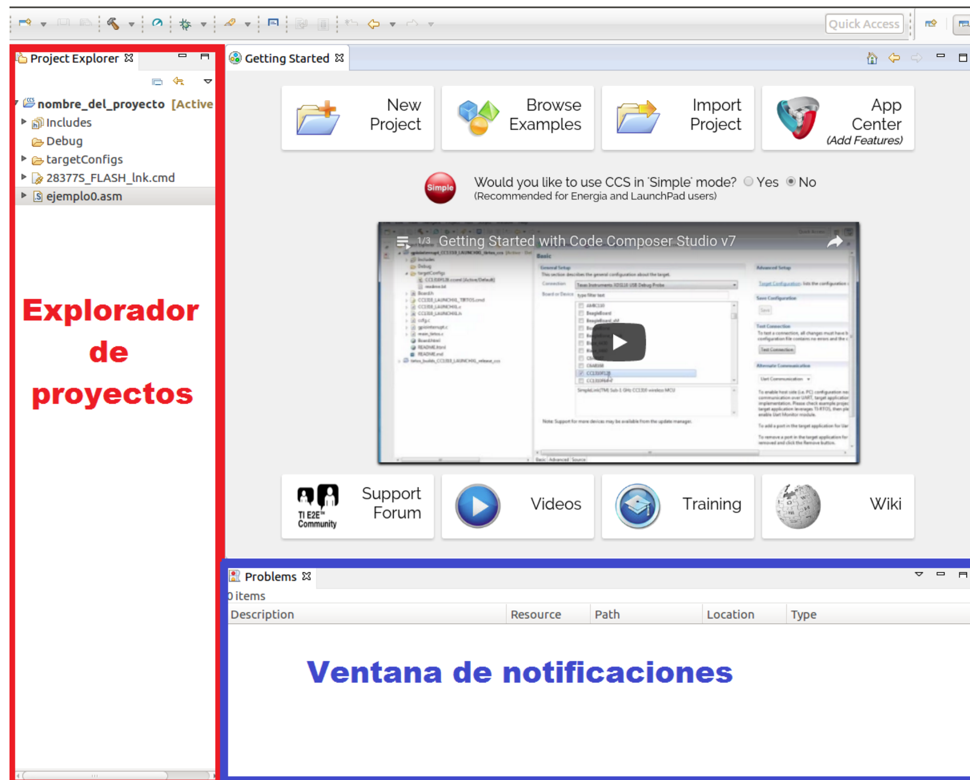


Figura 2.6: Interfaz *CCS Edit*

En el explorador de proyectos (Figura 2.6) se podrán observar todos los proyectos creados y almacenados dentro de la carpeta de trabajo seleccionada (al abrir Code Composer como se mostró en la Figura 2.1). Para activar un proyecto previamente almacenado es necesario dar clic derecho sobre el nombre del proyecto y seleccionar la opción *Open Project*, de esta manera será posible editar, compilar y ejetuar el proyecto, de la misma manera, la edición del proyecto se puede desactivar seleccionando la opción *Close Project*.

La ventana de notificaciones (Figura 2.6) muestra los problemas y avisos generados en la compilación de un programa.

En caso de no haber seleccionado el archivo *cmd* en la campo de *Linker command file* de la ventana *Advanced settings* (Figura 2.5a), *Code Composer* genera uno por defecto, mismo que puede ser utilizado para la creación del proyecto, sin embargo, es recomendable editar o

crear un archivo *.cmd* para mejorar la distribución de los bloques de memoria como el código (sección *.text*) y memoria de datos (sección *.data*).

Si se desea cargar un archivo *cmd* creado por el usuario, primero es necesario eliminar el *cmd* que generó por defecto el *Code Composer* para evitar fallas en el compilador.

Para agregar un nuevo *cmd* haga clic derecho en la carpeta del proyecto, ubicada en el explorador de proyectos, y seleccione *Add Files...*, posteriormente aparecerá una ventana en donde deberá buscar y seleccionar el archivo *.cmd* que desee insertar. Se recomienda utilizar el *cmd* que se localiza en el Apéndice A.

Después de realizar las configuraciones del proyecto, es necesario crear el archivo donde se escribirá el código del programa. Cuando el tipo de proyecto generado es en lenguaje C, el CCS crea automáticamente el archivo editor de texto con extensión *.c*, sin embargo, cuando es un proyecto para lenguaje ensamblador es necesario crearlo el archivo fuente, para ello, haga clic derecho sobre la carpeta del proyecto y seleccione *new* → *Source file*. Hecho lo anterior, emergerá una ventana como la que se muestra en la Figura 2.7. En *Source file* se escribe el nombre del archivo con terminación *.asm* (tipo de archivo para códigos en ensamblador).

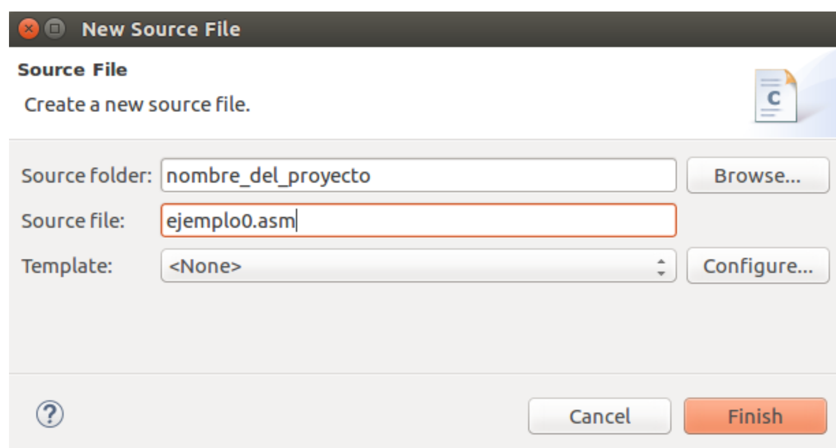


Figura 2.7: Creación de un nuevo archivo en ensamblador

2.2. Configuración del ControlSuite

Para facilitar el uso de los dispositivos programables de Texas Instrument, la empresa ha creó una herramienta llamada *ControlSuite*^{TM} la cual es un conjunto de herramientas,

bibliotecas y documentación útiles para su utilización con tarjetas embebidas de TI, teniendo un apartado dedicado para sus microcontroladores de la familia C2000 [6].

2.2.1. Instalación

ControlSuite^{TM} puede ser descargado de forma gratuita desde la página de Internet de TI [7], http://www.ti.com/tool/controlsuite. A la fecha de redacción de este trabajo, esta herramienta solo se encuentra disponible para sistemas operativos *Windows*, sin embargo, en la *wiki* de TI se pueden encontrar algunas opciones para su instalación en Linux. El desarrollo de este trabajo se realizó con la versión de Code Composer Studio (CCS) v7.4.0.00015, en una computadora con Ubuntu 16.04 LTS.

La descarga de *ControlSuite*^{TM} se puede realizar como un archivo ejecutable (*.exe*), o como una carpeta comprimida (*.zip*). Las implementaciones de ejemplo que se desarrollaron en este libro se realizaron con la versión 3.4.4 de la herramienta. Para trabajar con ControlSuite en el sistema operativo Ubuntu, se realizó la instalación en Windows siguiendo el asistente, teniendo presente la dirección que se ingresa para descargar los archivos de la herramienta. Posteriormente se exportaron todos esos archivos en la carpeta de trabajo en Ubuntu para poderlos utilizar con CCS.

2.2.2. Creación de un proyecto empleando ControlSuite

El primer paso es crear un proyecto nuevo para lenguaje C como se explicó en la Sección 2.1. Posteriormente, se selecciona la carpeta del proyecto en el *Explorador del proyectos* (Figura 2.6), dar clic derecho sobre el icono del directorio para desplegar el menú y seleccionar la opción *propiedades*.

Al desplegarse la ventana de propiedades del proyecto, seleccionar la pestaña *Resource* → *Linked Resources*, como se observa en la Figura 2.8. Aquí se encuentran las *direcciones* o *paths* que emplea CCS para compilar los proyectos, es importante señalar que **NO SE DEBEN MODIFICAR** los valores originales porque puede alterar la configuración del compilador y éste puede dejar de funcionar.

Se puede definir una variable global que será igual a la dirección general o *path* de la carpeta de trabajo de ControlSuite a utilizar. Esto se realiza, haciendo clic en el botón "New...", desglosando la ventana "Edit Variable" donde se deberá escribir el nombre de la variable, por ejemplo *CONTROLSUITE*. Posteriormente se debe seleccionar la dirección deseada, esto se realiza por medio del botón "Folder..". Al tener diferentes versiones la herramienta, se han creado diferentes versiones de bibliotecas, lo cual se puede ver en la carpeta

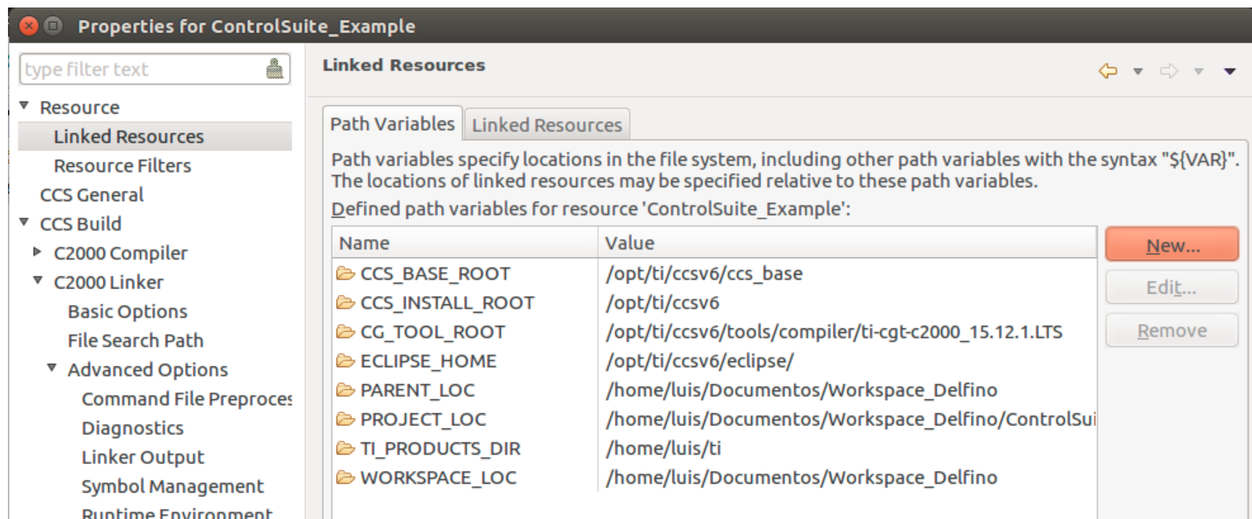


Figura 2.8: Direcciones o *paths* del compilador de CCS.

`device_support/F2837xS/` del directorio de ControlSuite. Entonces, un ejemplo de la dirección que se le asignará a la variable *CONTROLSUITE* es `/home/Documentos/Libs_Delfino/ControlSuite/device_support/F2837xS/v190`, donde **v190** indica la versión de las bibliotecas, como se muestra en la Figura 2.9.

Figura 2.9: Creación de nuevo path.

A continuación, **se comprobarán las opciones del procesador en la pestaña *Build* → *C2000 Compiler* → *Processor Options***. La configuración se muestra en la Figura 2.10, resaltando que esta corresponde al uso del DSP TMS320F28377S.

Posteriormente, **se tienen que incluir las direcciones o *paths*** donde se encuentran las bibliotecas a utilizar, para ello se accede a la pestaña *Build* → *C2000 Compiler* → *Include Options* y en la sección "*Add dir to #include search path*" se deben incluir una a la vez,

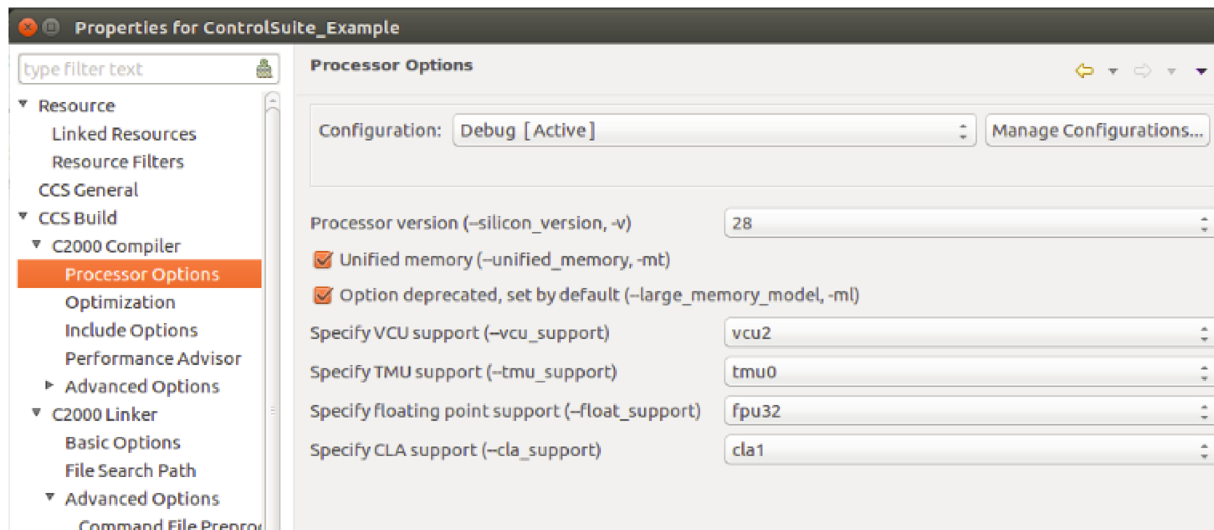


Figura 2.10: Configuración de las opciones del procesador.

las carpetas `F2837xS_headers/include` y `F2837xS_common/include`, utilizando la variable *CONTROLSUITE* para abreviar la dirección, por lo que se deberá dar clic al botón "Add", cuyo icono es una hoja con una cruz verde, señalado en la Figura 2.11 junto con la ventana que despliega con la dirección declarada.

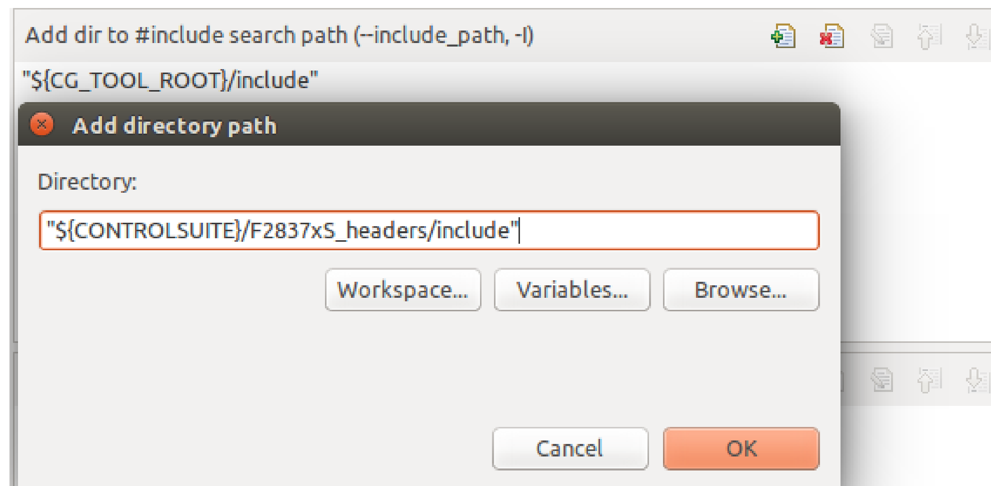


Figura 2.11: Ventana para incluir direcciones bibliotecas de ControlSuite a considerarse al compilar.

En la Figura 2.12 se observan las direcciones que se deben agregar al compilador para que pueda utilizar las bibliotecas de ControlSuite.

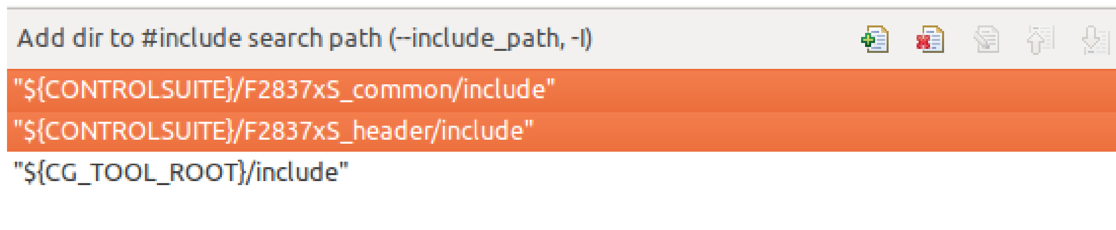


Figura 2.12: Direcciones que deben estar consideradas para incluir bibliotecas de ControlSuite.

Además, **se agregará un símbolo para indicarle al procesador el CPU a emplear (CPU1)**, esto se configura en la pestaña *CCS Build* → *C2000 Compiler* → *Predefined Symbols*. En la sección "*Pre-defined NAME*", se da clic al icono de la hoja con una cruz verde, para desplegar la ventana "Enter Value", donde se deberá escribir CPU1, como se muestra en la Figura 2.13 y dar clic en el botón de *OK*. Este símbolo es importante cuando se emplean procesadores de varios núcleos, para el caso de la Delfino F28377S no es necesario agregar dicho símbolo porque contiene un procesador de un núcleo.

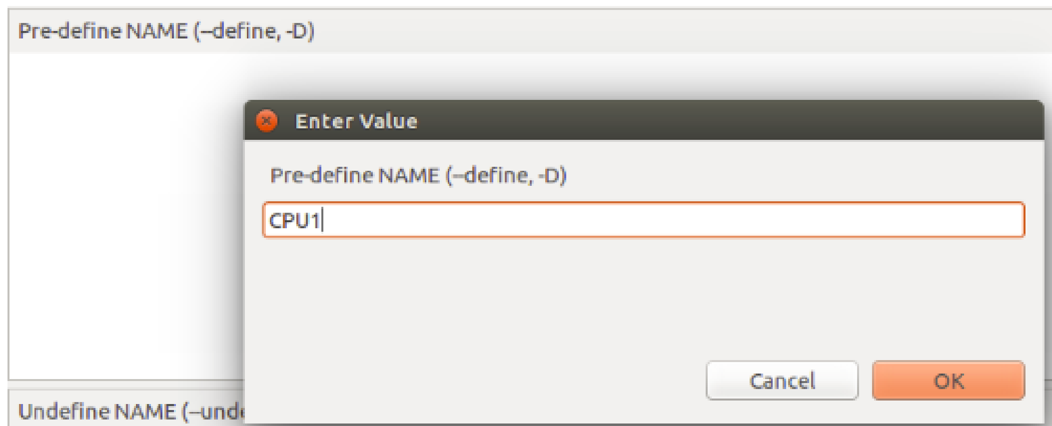


Figura 2.13: Definición del CPU a emplear en la compilación.

El mapa de memoria definido en los archivos *.cmd*, puede ser configurado de diferentes formas acorde a las necesidades de la aplicación como se verá en la Sección 2.3. ControlSuite contiene diferentes archivos *.cmd* de uso general, con diferentes características, las cuales son utilizadas por las diferentes bibliotecas que se pueden utilizar dentro de un proyecto para el uso de los periféricos, por lo que se debe de agregar al compilador la ubicación de dichos archivos *.cmd*.

Para realizar lo mencionado en el párrafo anterior, seleccionamos la pestaña *CCS Build* → *C2000 Linker* → *File Search Path*. En la sección "*Add <dir> to library search path*" se

agregarán las direcciones `F2837xS_headers/cmd` y `2837xS_common/cmd` (nuevamente con el botón cuyo icono es una hoja con una cruz verde).

Después, en la sección "*Include library file or command file as input*" se deben agregar los archivos `F2837xS_Headers_nonBios.cmd`, ubicado en la carpeta `F2837xS_headers/cmd` y la biblioteca `rts2800_fpu32.lib` que se encuentra en la raíz del CCS (`ti/ccsv6/tools/compiler/c2000_15.12.3.LTS/lib`). En la Figura 2.14 se muestra la lista de direcciones y archivos que se deben incluir en esta ventana.

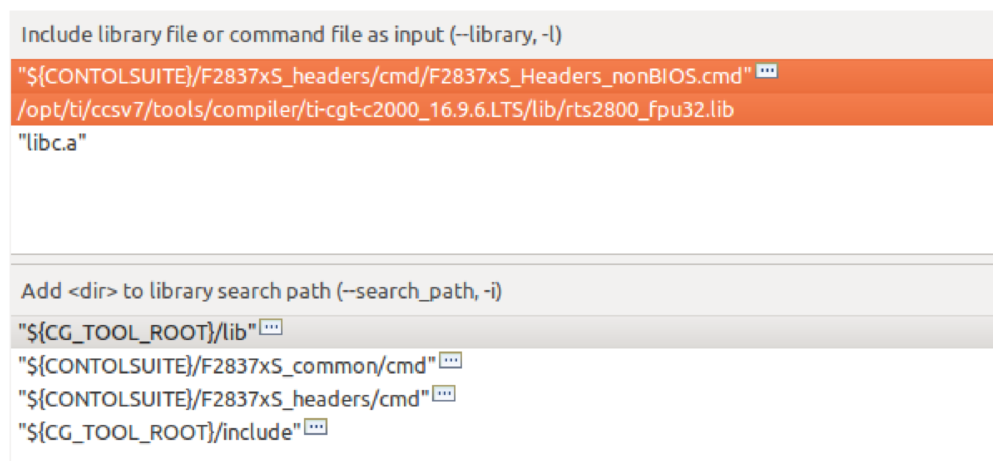


Figura 2.14: Inclusión de direcciones y archivos para el *linker*.

Para aplicar todas las configuraciones realizadas a las propiedades del proyecto, se debe hacer clic en el botón "OK" de la ventana *Properties*.

Por último, **se deben agregar las bibliotecas que configuran los periféricos a utilizar**, lo cual se hace seleccionando la carpeta del proyecto en el *Explorador de proyectos* y dar clic derecho sobre el icono seleccionado, para acceder a la opción *Add files* del menú desplegado. En la ventana "*Add files to ..*", se debe agregar el archivo `F2837xS_GlobalVariableDefs.c` que se encuentra en la dirección `ControlSuite/device_support/F2837xS/v190/F2837xS_headers/source`, como se muestra en la Figura 2.15, recordando que se puede usar la versión de bibliotecas que se desee, en este caso se está utilizando la v190. El archivo citado contiene las definiciones para las secciones de memoria de los diferentes periféricos que tiene el MCU.

Al dar clic en el botón "OK" para agregar el archivo, se desplegará la ventana "File operation" como se observa en la la Figura 2.16. En dicha ventana se muestran las dos formas posibles en las que se puede agregar el archivo. La primera de ellas es haciendo una copia del archivo original en nuestra carpeta del proyectos y la segunda opción genera un acceso

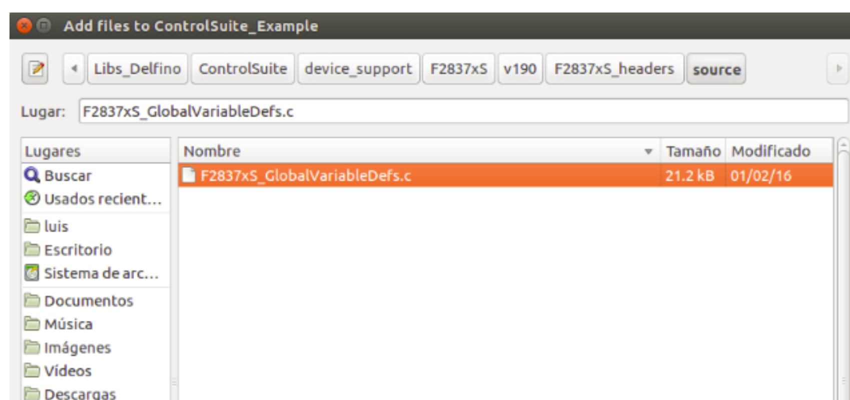


Figura 2.15: Ubicación del archivo *F2837xS_GlobalVariableDefs.c*

directo o *link* hacia el archivo original. Se recomienda utilizar la primer opción para no editar el código fuente de las bibliotecas del ControlSuite ya que no es recomendable modificar dichos archivos, a menos que se tenga el conocimiento necesario.

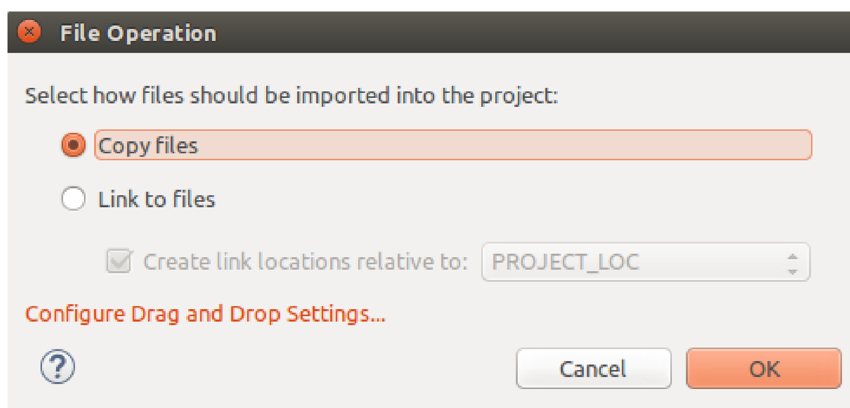


Figura 2.16: Opciones para agregar archivos al proyecto.

Una vez agregado el archivo de **configuración de variables**, se deben agregar los archivos fuentes.c de los periféricos que se van a utilizarán, los cuales se encuentran en `ControlSuite/device_support/F2837xS/v190/F2837xS_common/source`. Sin embargo, independientemente del proyecto que se vaya a desarrollar, es recomendable agregar siempre los siguientes archivos:

- `F2837xS_CodeStartBranch.asm`
- `F2837xS_SysCtrl.c`
- `F2837xS_usDelay.asm`

- `F2837xS_Gpio.c`

Estos archivos contienen funciones de uso común para varios proyectos. Una vez realizada esta configuración, se pueden crear códigos que empleen las bibliotecas previamente agregadas, sin embargo, los ejemplos que utilizan CONTROLSUITE en el presente libro se emplean en el Capítulo 5 para configurar los periféricos de la tarjeta.

2.3. Mapa de memoria

Antes de comenzar a escribir programas de diferentes aplicaciones, es necesario conocer el mapa de memoria del dispositivo de trabajo porque en cada proyecto, se utilizarán diferentes recursos del DPS, implicando manejar distintas partes de los registros de memoria dedicados, aunque los códigos solo manejen datos o periféricos.

En el diagrama de bloques de la Figura 1.1, se observa la arquitectura del TMS320-F28377S, donde en diferentes partes se encuentran algunos de los módulos que forman la memoria del DSP, los cuales se pueden clasificar en dos clases; registros de memoria para periféricos y registros para almacenamiento/transferencia de datos y de códigos. En la Sección seis del manual [4], se encuentran las tablas que contienen el nombre de las diferentes secciones de la memoria, así como su tamaño y direcciones.

El mapa de memoria está formado por tres tipos; RAM, ROM y FLASH organizados en las siguientes partes;

* Memoria general de los dispositivos de la familia C28x; formada por la memoria RAM es utilizada principalmente para correr el código en tiempo real y para definir variables globales inicializadas y no inicializadas, teniendo conexión directa con el CPU.
* Bancos de la memoria FLASH; comúnmente utilizada para guardar los programas, aunque en aplicaciones se deba de transferir las instrucciones del código a la memoria RAM.
* Memoria de interfaz externa EMIF; utilizada para conectar al DSP otras memorias externas a su arquitectura.
* Registros de memoria de periféricos; utilizados para configurar y administrar los datos que adquieren o se escriben como salidas por los diferentes puertos.

La memoria ROM es reservada a excepción de una parte en la cual pueden escribirse datos una sola vez para alguna aplicación, por ejemplo, para grabar coeficientes de algún filtro FIR. Para información más detallada de la memoria del dispositivo se recomienda al

lector consultar los manuales [4, 8].

En cada proyecto se tienen que declarar los módulos de memoria que se utilizarán, además de las correspondientes asignaciones para cada una de las secciones necesarias para implementar algún programa, dicha información se manifiesta por escrito en un archivo con terminación *.cmd*, denominado en los manuales como *linker command file*. La IDE CCS al crear un nuevo proyecto (proceso que se describirá en la Sección 2.1), recomienda al usuario un archivo *.cmd* configurado por TI para el desarrollo de proyectos básicos y generales, sin embargo, el usuario puede definir su propio archivo y para ello se describirá de forma básica a continuación la forma de hacerlo.

2.3.1. Linker command file o archivo *.cmd*

Referente a su nombre, este archivo permite declarar opciones para enlazar el archivo principal de un proyecto con otros secundarios (por ejemplo, al utilizar Control Suite que se verá en la Sección 2.2.2) y también declarar y asignar el mapa de memoria a utilizar. Estos archivos pueden contener

* Opciones de enlace para la crear el archivo ejecutable.
* Nombres de archivos objeto, que contienen asignaciones de bloques de memoria de entrada (datos de entrada).
* Declarar la inclusión de alguna biblioteca.
* Declarar variables globales.

El archivo se divide principalmente en dos partes; *MEMORY* es el apartado donde se definen los bloques de memoria a utilizar del dispositivo y en *SECTIONS* se asignan bloques de memoria a las secciones necesarias para que un programa funcione. El esquema general de este archivo se muestra a continuación

```
MEMORY{
// Declaración del mapa de memoria a utilizar
// (bloques de la memoria RAM/FLASH)
...

}

SECTIONS{
// Asignación de los bloques de memoria a las
// secciones necesarias para un programa
...

}
```

A continuación se presentarán las instrucciones y formato de cada parte del archivo *.cmd*.

Declaración de bloques de memoria

El objetivo y función del bloque **MEMORY** es asignar los nombres e intervalos de la memoria conectada directamente al CPU (RAM o FLASH) que se van a utilizar, declarando esta información con el siguiente formato

```
*
**** Formato para definir el uso de memoria ****
*
file1.obj    file2.obj          // Archivos de entrada
--output_file = prog.out        // Opciones del enlazador
#define zeros 0                 // Var. global

MEMORY
{
   PAGE 0: nombre 1 (atrib.)  :origin= dir. ini., length= tamaño, fill= const.
   PAGE 1: nombre 2 (atrib.)  :origin= dir. ini., length= tamaño, fill= zeros
    .
    .
   PAGE n: nombre n (atrib.)  :origin= dir. ini., length= tamaño, fill= const.
}
```

donde:

PAGE; es una referencia para identificar espacios de la memoria, es decir que define un conjunto de bloques de memoria con un objetivo común. Se pueden declarar hasta 32 767 páginas de memoria, siempre y cuando siempre la página 0 defina la memoria del programa y la página 1 la memoria para datos.

Nombre n (atrib.); es una denominación de referencia al bloque de memoria que se esta declarando para ser utilizado, una práctica común es ocupar los nombres asignados por el fabricante, especificados en la Sección 6.3 del manual [4]. Separado por un espacio, entre paréntesis se puede definir para dicho bloque, uno de los siguientes atributos

* **R** si el bloque es solo de lectura
* **W** si el bloque es solo de escritura
* **X** si el bloque contiene código del ejecutable
* **I** si el bloque puede ser inicializado con datos

Especificar un atributo no es necesario, al no declararlo se deja libre de restricciones el uso del bloque de memoria.

origin; este campo es para especificar la dirección de inicio del bloque de memoria que se esta declarando. La dirección debe estar en el intervalo de la memoria del dispositivo de libre

uso, especificada en bytes por un valor entero de 32 bits que puede ser escrito en decimal, octal o hexadecimal siendo este último el formato el más usual. En lugar de la palabra *origin* se puede utilizar la palabra *org* o la letra *o*.

length; especifica la longitud del bloque de memoria expresado en bytes, considerado por el compilador como un valor entero de 22 bits que se puede escribir en decimal, octal o hexadecimal. El tamaño declarado del bloque puede asignarse acorde a los valores definidos en los mapas de memoria especificados en la Sección 6.3 del manual [4], o pueden ser definidos libremente por el usuario respetando la longitud de palabra de esta variable, acorde a la aplicación. Este campo se puede abreviar con la palabra *len* o la letra *l*.

fill; declara un valor que se desea contenga todas las localidades de memoria del bloque que se esta definiendo. Dicho valor puede ser definido explícitamente o utilizando una variable global. Este campo no es obligatorio declararlo y también puede abreviarse como *fil* o con la letra *f*.

Un ejemplo breve sobre la declaración de la función *MEMORY* en el archivo *.cmd* se muestra a continuación, en donde se puede observar la definición de las dos páginas necesarias para el funcionamiento de un programa que no utiliza periféricos del MCU y la extensión del bloque *RAMGS0* que cubre los bloques del mapa predeterminado *RAMGS1* a *RAMGS12*, con la finalidad de contar con más espacio para cargar o guardar datos para una aplicación de cálculo.

```
MEMORY{
   PAGE 0:
   /* Memoria para el programa */
   /* Bloques de memoria (RAM/FLASH) pueden ser declarados en la */
   /* página 1 para usarlos como datos */
   /* BEGIN es utilizado para iniciar desde FLASH o RAM el modo */
   /* bootloader, en este caso se inicia desde FLASH */

   BEGIN          : origin = 0x080000, length = 0x000002 //obligatorio
   RAMM0          : origin = 0x000122, length = 0x0002DE
   RAMD0          : origin = 0x00B000, length = 0x000800
   RAMLS0         : origin = 0x008000, length = 0x000800
   RAMLS1         : origin = 0x008800, length = 0x000800
   RESET          : origin = 0x3FFFC0, length = 0x000002 //obligatorio

   /* Sectores de FLASH */

   FLASHA         : origin = 0x080002, length = 0x001FFE
   FLASHB         : origin = 0x082000, length = 0x002000
   FLASHC         : origin = 0x084000, length = 0x002000
   FLASHD         : origin = 0x086000, length = 0x002000
```

```
    FLASHE          : origin = 0x088000, length = 0x008000
    FLASHF          : origin = 0x090000, length = 0x008000

    PAGE 1:
    /* Memoria de datos */
    /* Bloques de memoria (RAM/FLASH) pueden ser declarados en la */
    /* página 0 para usarlos como datos */

    BOOT_RSVD       : origin = 0x000002, length = 0x000120 //obligatorio
    RAMM1           : origin = 0x000400, length = 0x000400
    RAMD1           : origin = 0x00B800, length = 0x000800

    RAMLS5          : origin = 0x00A800, length = 0x000800

    RAMGS0          : origin = 0x00C000, length = 0x00D000
    RAMGS13         : origin = 0x019000, length = 0x001000
}
```

Como nota adicional, no es necesario tener que utilizar los dos tipos de memoria FLASH y RAM, es posible utilizar solo un tipo de memoria acorde a la implementación que se vaya a desarrollar. Al ocupar FLASH el tiempo de escritura para entrar al modo *Debug* o almacenar el código en el dispositivo, es mayor que al utilizar solo memoria RAM, razón por la que como parte de este trabajo, se propone al usuario un archivo *.cmd* que utiliza solo RAM en diferentes aplicaciones, dicho mapa de memoria se encuentra en el Apéndice A.

Asignación de bloques de memoria

El compilador maneja la memoria del dispositivo como dos bloques lineales; uno para el programa y otro para datos, el contenido específico de cada uno de estos bloques es el siguiente

* La memoria de programa contiene el código ejecutable, registros de inicialización y tablas de casos al utilizar la instrucción *switch*.

* La memoria de datos contiene variables externas, estáticas y la pila del sistema.

Ambos bloques son organizados en *secciones* que cumplen un propósito particular en la etapa de compilación del proyecto y son declaradas en la segunda parte del archivo *.cmd* denotada como **SECTIONS**. Una *sección* se puede definir como la unidad mínima de un archivo objeto, que ocupa bloques de memoria consecutivos y se clasifican en dos tipos básicos

* Secciones inicializadas; contienen código o datos definidos al iniciar la ejecución del programa.

* Secciones no inicializadas; son localidades del mapa de memoria reservadas para datos que no han sido definidos al inicio de la ejecución del programa.

Para generar el código ejecutable, el compilador necesita definir las secciones base y asignarles bloques de memoria a cada una de ellas, en caso de que no se definan por el usuario CCS implementa un algoritmo para declarar una distribución, dicho procedimiento se puede consultar en la Sección 8.7 de [9]. En la Tabla 2.1 se listan las secciones base y algunas de uso especial

Tabla 2.1: Lista se las secciones básicas para generar el código ejecutable de un proyecto en CCS para el TMS320F28377S.

Sección	**Inicialización**	**Descripción**
.text	Si	Contiene el código ejecutable del programa
.ebss	No	Utilizada para variables globales y estáticas
.data	Si	Sección dedicada para datos COFF ABI ->Contiene datos inicializados
	Si/No	EABI ->Inicializado saliendo del ensamblador; cambiado a no inicializado por el linker
.econst	Si	Contiene constantes de cadena, literales de cadena, tablas de conmutación, declaración e inicialización de variables globales y estáticas, y datos definidos como *const* en C/C++. Esta sección normalmente es de solo lectura
.cinit	Si	Se usa para inicializar variables globales en un programa de C/C++
.stack	No	Utilizado para la pila de llamadas de función
.esysmem	No	Se usa para el grupo de asignación de memoria dinámica.
.switch	Si	Contiene los saltos que se pueden hacer al utilizar la instrucción switch
.cio	No	Buffer para funciones de la biblioteca stdio en programas de C/C++
.pinit	Si	Contiene la lista de constructores globales para programas de C/C++
codestart	Si	Sección para iniciar el modo bootloader desde el bloque asignado
ramfunctions	Si	Memoria para ejecutar funciones en la RAM

La sección *.data* puede no ser declarada, ya que otras secciones como *.ebss* ó *.esysmem* hacen parte de su función, en algunos programas del Capítulo 4 se utilizaa dicha sección con el objetivo mostrar su uso. Una descripción más detallada de las secciones listadas así como de otras que se pueden utilizar, se puede consultar en [9] y [10]. La declaración de las

secciones se realiza con la siguiente sintaxis

```
SECTIONS
{
    nombre 1: [propiedad [,propiedad][,propiedad]...]
    nombre 2: [propiedad [,propiedad][,propiedad]...]
    .
    .
    nombre n: [propiedad [,propiedad][,propiedad]...]
}
```

En el campo de *nombre* se escribe alguna de la secciones mostradas en la Tabla 2.1 seguida una o más propiedades, separadas por una coma teniendo las siguientes opciones:

Asignación de carga; define en qué lugar(es) de la memoria se cargará la sección declarada, la sintaxis para especificar esta propiedad es

```
Nombre de sección: load = nombre del bloque o
Nombre de sección: > nombre del bloque
```

También es posible declarar opciones alternas de asignación de memoria a una sección, para el caso donde el tamaño del bloque no sea el suficiente, el compilador podrá utilizar la segunda, tercera, cuarta, etcétera opción, por lo que la sintaxis anterior se reescribe como

```
Nombre de la sección: > bloque opción 1 | ... | bloque opción n
```

Otro caso que se puede presentar en la asignación, es que el tamaño de un bloque de memoria no sea suficiente para la sección, por lo que es posible hacer un tipo de expansión que consiste en utilizar otros bloques de memoria consecutivos, para que el compilador divida la información de forma uniforme en aquellos que se hayan declarado. La sintaxis de asignación de sección con opción de expansión es la siguiente

```
Nombre de la sección: >> bloque opción 1 | ... | bloque opción n
```

Algunas de las opciones extras que se pueden incluir en esta propiedad son;

* Dirección especifica; en lugar de declarar el nombre de un bloque de memoria para una sección, también se pude indicar la dirección inicial del bloque que se asignará utilizando el signo de igualdad, como se muestra en el siguiente ejemplo

  ```
  .text: load = 0x1000
  ```

* Alineación (**.align**); el ensamblador trabaja con un contador de programa dedicado a las secciones (SPC), el cual va apuntando a las direcciones de memoria asignadas a

alguna sección. La opción *.align* desfasa dicho contador acorde a la cantidad de palabras de 16 bits que se le indiquen y en caso de solo escribir la opción, el ensamblador desfasa la dirección hasta la siguiente sección. La sintaxis para utilizar esta opción se muestra a continuación

```
.align = 4    o    .align(4)    o    .align 4
```

En el archivo *.cmd*, al incluirlo en la asignación de memoria a una sección, desfasa la dirección inicial del bloque de memoria que se utilizará para la tarea de la sección, con el objetivo de tener localidades de separación para no sobreescribir datos.

* Página (**PAGE = n**); su función es especificar la página de memoria en la que se encuentran los o el bloque de memoria asignado a una sección.

Asignación para ejecución; esta propiedad define en qué parte de la memoria se ejecutará la sección, y es declarada con la siguiente sintaxis

```
run = nombre del bloque de memoria     o
run > nombre del bloque de memoria
```

Secciones de entrada; define secciones de entrada, declaradas en archivos objeto que constituyen una sección de salida. Se especifican al escribirse entre paréntesis tipo llave {*archivo_objeto.obj*}.

Tipo de sección; son tres clases que modifican la forma en la que el compilador trata a las secciones señaladas por alguna de estas denominaciones. Para especificar un tipo de sección se sigue el formato

```
type = clase o tipo
```

Las posibles clases o tipos que se pueden utilizar son

* DSECT; crea una sección ficticia que no tiene asignada memoria física al generar el mapa de memoria del archivo ejecutable. Tiene como funciones; declarar símbolos globales que si son asignados a localidades de memoria en el intervalo vinculado a la sección y para buscar símbolos externos no definidos (variables globales en el proceso de ensamble) en bibliotecas.

* COPY; las funciones de esta clase son similares al tipo DSECT, a excepción que este tipo contiene y asocia información que si se escribe en el archivo ejecutable.

* NOLOAD; este tipo hace que la sección de salida solo aparezca en el mapa de memoria del archivo ejecutable, pero no escribe el contenido de la sección, la información de reubicación y la información del número de línea del bloque de memoria.

Valor de relleno; define algún valor para escribirlo en alguna sección no inicializada, mediante la instrucción

```
fill = valor
```

A continuación se muestra un ejemplo de declaración de secciones

```
SECTIONS
{
   // Asignación de memoria a secciones del programa:
   .cinit          : > FLASHB        PAGE = 0, ALIGN(4)
   .pinit          : > FLASHB,       PAGE = 0, ALIGN(4)
   .text           : >> FLASHB | FLASHC | FLASHD | FLASHE  PAGE = 0, ALIGN(4)
   codestart       : > BEGIN                              PAGE = 0, ALIGN(4)
   ramfuncs        : LOAD = FLASHD,
                        RUN = RAMLS0 | RAMLS1 | RAMLS2 |RAMLS3,
                        LOAD_START(_RamfuncsLoadStart),
                        LOAD_SIZE(_RamfuncsLoadSize),
                        LOAD_END(_RamfuncsLoadEnd),
                        RUN_START(_RamfuncsRunStart),
                        RUN_SIZE(_RamfuncsRunSize),
                        RUN_END(_RamfuncsRunEnd),         PAGE = 0, ALIGN(4)

#ifdef __TI_COMPILER_VERSION__
   #if __TI_COMPILER_VERSION__ >= 15009000
    .TI.ramfunc : {} > FLASHD, PAGE = 0, ALIGN(4)
   #endif
#endif

   // Asignación de memoria a secciones no inicializadas para datos
   .stack           : > RAMGS14                           PAGE = 0
   .data            : > RAMGS0                            PAGE = 1
   .ebss            : >> RAMGS13                          PAGE = 1
   .esysmem         : > RAMLS5                            PAGE = 1

   // Asignación de memoria a secciones inicializdas
   // que van en memoria Flash
   .econst          : >> FLASHF | FLASHG | FLASHH         PAGE = 0, ALIGN(4)
   .switch          : > FLASHB,                           PAGE = 0, ALIGN(4)

   .reset           : > RESET,                        PAGE = 0, TYPE = DSECT

}
```

Esto fue un breve resumen sobre el *Linker command file* o archivo *.cmd*, la información detallada para escribir este archivo se puede consultar en los manuales [9] y [10].

2.3.2. Linker command file para periféricos

En los proyectos donde se utilicen algunos de los periféricos del TMS320F28377S, utilizando el conjunto de bibliotecas de *Control Suite* en lenguaje C/C++ (el cual se explicará en la Sección 2.2.2) es necesario incluir para compilar, el mapa de memoria donde se encuentran declarados los registros que utiliza cada puerto para su funcionamiento. El archivo *F2837xS_Headers_nonBIOS.cmd* ubicado dentro de la carpeta de *Control Suite*, en la siguiente dirección

`device_support/F2837xS/vxxx/F2837xS_headers/cmd/`

donde en */vxx* debe elegirse alguna de las versiones que se encuentran disponibles, por ejemplo la v200. A diferencia del archivo *.cmd* descrito en la subsección 2.3.1, las direcciones de los registros en la memoria son estáticos, por lo que no debe de modificarse el archivo *F2837xS_Headers_nonBIOS.cmd*, o en caso de que el usuario desee hacerlo es recomendable apoyarse del manual spruxh5d [8] para direccionar adecuadamente los datos a los registros correspondientes de cada periférico.

La estructura del archivo *Lniker command* para periféricos es la misma que se ha descrito en esta sección y sigue las mismas reglas descritas para declarar las asignaciones de memoria, como se puede observar en el breve contenido del archivo *F2837xS_Headers_nonBIOS.cmd* que se muestra a continuación:

```
MEMORY
{
 PAGE 0:    /* Memoria para el Programa */

 PAGE 1:    /* Memoria para Datos */

   // Registros para contener el resultado de la
   // conversión de alguno de las terminales de
   // los cuatro conjuntos que forman el ADC
   ADCA_RESULT   : origin = 0x000B00, length = 0x000020
   ADCB_RESULT   : origin = 0x000B20, length = 0x000020
   ADCC_RESULT   : origin = 0x000B40, length = 0x000020
   ADCD_RESULT   : origin = 0x000B60, length = 0x000020

   // Registros de configuración para el funcionamiento
   // de cada uno de los cuatro bloques que forman
   // el periférico ADC
   ADCA          : origin = 0x007400, length = 0x000080
   ADCB          : origin = 0x007480, length = 0x000080
   ADCC          : origin = 0x007500, length = 0x000080
   ADCD          : origin = 0x007580, length = 0x000080
.
.
```

```
.

SECTIONS
{
/*** PIE Vect Table and Boot ROM Variables Structures ***/
  UNION run = PIE_VECT, PAGE = 1
   {
      PieVectTableFile
      GROUP
      {
         EmuKeyVar
         EmuBModeVar
         FlashCallbackVar
         FlashScalingVar
      }
   }

   // Asignación de memoria a los registros para
   // guardar el resultado de la conversión del ADC
   // por cada uno de sus bloques A, B, C y D
   AdcaResultFile         : > ADCA_RESULT,   PAGE = 1
   AdcbResultFile         : > ADCB_RESULT,   PAGE = 1
   AdccResultFile         : > ADCC_RESULT,   PAGE = 1
   AdcdResultFile         : > ADCD_RESULT,   PAGE = 1

   AdcaRegsFile           : > ADCA,          PAGE = 1
   AdcbRegsFile           : > ADCB,          PAGE = 1
   AdccRegsFile           : > ADCC,          PAGE = 1
   AdcdRegsFile           : > ADCD,          PAGE = 1
.
.
.
}
```

2.4. Componentes de un programa en ensamblador

Un programa fuente en ensamblador debe contener una estructura básica para que se pueda compilador correctamente. Este consiste en expresiones que pueden contener directivas, instrucciones de ensamblador o macroinstrucciones y subrutinas con etiquetas, además de comentarios (opcionales). En la Figura 2.17 se muestra un código que muestra la estructura básica de un programa en lenguaje ensamblador.

```
1          .global _c_int00
2
3  WDCR     .set    07029h
4  CTE_WD   .set    0068h
5  N        .set    1000     ; Tamaño
6  x        .space  N*16     ; Espacio reservado para señal de entrada x
7  M        .set    201      ; Tamaño del
8  coef     .space  M*16     ; Espacio reservado para filtro FIR
9  x_buf    .space  M*16     ; Espacio res
10 x_ext    .word   0        ; Localidad extra
11 y        .space  N*16     ; Espacio reservado para señal de salida y

   .text
   _c_int00
   * Desabilita el Watc
            EALLOW
                MOVL    XAR1,#WDCR
                MOV     *XAR1,#CTE_WD
            EDIS
   *
            SETC    SXM
            SPM     #0          ; Corrimientos nulos para el registro P
            MOVW    DP,#x_buf   ; DP apunta a la página de cont
            MOVL    XAR1,#x     ; XAR1 apunta a la dirección de x
            MOVL    XAR2,#y     ; XAR2 apunta a la dirección de y
            MOV     AR4,#N-1    ; Número de interaciones del filtro

   CICLO    MOV     AL,*XAR1++  ; Dato de x al acumulador para insertar en el
                                   ; bufer de retardo y aumenta localidad del
                                   ; apunta

            MOV     @x_buf,AL   ; Dato en la localidad 1 del bufer de retardos
            MOVL    XAR7,#x_buf ; XAR7 apunta al bufer de retardos
            MOVL    XAR3,#coef  ; XAR3 apunta al bufer de coeficientes
            ZAPA                ; Limpia acumulador y registro P
            RPT #-1
            ||  P,*XAR3++,*AR7++
            ADDL    ACC,P       ; Suma la última operación al registro acumulador
            LSL     ACC,2       ; Corrimiento a la izquierda
            MOV     *XAR2++,AH  ; Resultado en y y aumeta la dirección del
                                ; apuntador XAR2

            MOVL    XAR5,#x_ext ; XAR5 apunta a una localidad posterior a la
                                ; final de x_buf
            RPT #-1             ; Ciclo para recorrer datos del bufer
            ||  D   *--XAR5
            BANZ    CICLO,AR4--

   FIN      NOP
49          LB  FIN
50          .end
```

Declaración de variables y constantes

Etiquetas

Instrucciones

Operandos

Comentarios

Fin del programa

Figura 2.17: Estructura de un programa en lenguaje ensamblador.

A partir de la Figura 2.17 se puede observar que la sintaxis de una instrucción en ensamblador es la siguiente:

ETIQUETA ***MNEMONICO*** *OPERANDOS* ; *COMENTARIOS*

Cabe resaltar que no todos los programas contienen la sección de comentarios, eso dependerá del programador, sin embargo, es recomendable realizar comentarios sobre el código para una guía fácil del desarrollo del mismo. A continuación se explica cada uno de los elementos de la sintaxis el programa:

1. **Etiqueta.** Es un símbolo que indica el inicio de una sección de código de interés, es decir, se utiliza para transferir el control del código en función de ciertas decisiones como saltos a subrutinas de funciones, interrupciones o saltos condicionales, como es el caso de un ciclo o una sentencia *if*. Una etiqueta en un código es opcional y cuando se utiliza se debe colocar en la primera columna del código, desplazando a la derecha los mnemónicos y operandos.

2. **Mnemónico.** Está conformado por Instrucciones, macroinstrucciones y directivas en ensamblador. Los mnemónicos ejecutan la orden dentro de un programa, es decir, operaciones, direccionamientos, saltos condicionales e indirectos, transferencia de datos, etc.

3. **Operandos.** Los operandos varían dependiendo del tipo de instrucción que se utilice y son separados por comas. Existen diferentes tipos de operandos en un programa de ensamblador, estos pueden ser registros, símbolos, acceso a constantes, etiquetas o apuntadores.

4. **Comentarios.** Son optativos dentro del código y no generan recursos en la ejecución del programa. El programador los utiliza para agregar información extra como guía en el flujo del programa o recordatorios. Se recomienda utilizarlos porque se pueden explicar detalles de la implementación del programa. Se anteceden por un ";" cuando los comentarios se realizan después de la instrucción. Los comentarios también se pueden realizar al principio de la fila, para ello se utiliza el símbolo "*" previo al comentario.

2.4.1. Directivas

Dentro de la estructura de un código fuente en lenguaje ensamblador existen diferentes tipos de datos para ser procesados, además de símbolos y secciones. Estos elementos se conocen como directivas y existen diferentes tipos, entre los más utilizados se encuentran los siguientes:

1. Directivas que controlan el uso de la sección

2. Directivas que inicializan valores en la memoria

3. Directivas que reservan espacio en la memoria

4. Declaración de símbolo como constante

5. Directivas de símbolos y final del programa

A continuación se describen la directivas descritas anteriormente:

Directivas que controlan el uso de la sección

En este tipo de directivas se define el tipo de sección dentro de un código en ensamblador, por ejemplo, la declaración de variables o la sección del código. En la Tabla 2.2 se muestran las directivas de control de uso de sección [11].

Tabla 2.2: Directivas que controlan el uso de la sección

Mnemónico	Descripción
.data	Se ensambla en la sección *.data* (datos inicializados)
.sect	Se ensambla en una sección nombrada (inicializada)
.text	Se ensambla en la sección *.text* (código ejecutable)

Directivas para inicializar datos en la memoria

En cualquier programa ya sea de lenguaje ensamblador, C o cualquier otro tipo, es importante clasificar los datos que el programador y el usuario proporcionen para poder ejecutar el programa, es decir, la declaración del tipo de datos que se van a utilizar a lo largo del programa. Estos se almacenan en la memoria dependiendo del archivo *cmd* utilizado en el proyecto, como se explicó en la Sección 2.3 [11].

Existen diferentes directivas de inicialización de valores en un programa de ensamblador y el uso de cada una va a depender del tipo de dato que se requiera para el proceso. En el TSM320F28377s se pueden utilizar datos de 16 bits, 32 bits y flotantes. Para poder ser utilizadas es necesario realizar la declaración de la siguiente manera:

nombre ***mnemónico*** *valor o valores* *;comentarios*

En la Tabla 2.3 se muestran las directivas para inicializar un valor en lenguaje ensamblador.

Directivas que reservan espacio en la memoria

En algunos casos, los datos procesados en un programa no necesariamente requieren ser inicializados porque se pueden cargar directamente a la memoria de la tarjeta, son datos

Tabla 2.3: Directivas de lenguaje ensamblador para inicializar datos en la memoria.

Mnemónico	Descripción
.bits	Inicializa uno o más bits sucesivos en la sección actual.
.byte	Inicializa una o mas palabras sucesivas en la sección actual.
.char	Inicializa una o más palabras sucesivas en la sección actual.
.field	Inicializa un campo de bits de tamaño (1-32 bits) con valor.
.float	Inicializa una o más constantes de punto flotante de precisión IEEE de 32 bits.
.int	Inicializa uno o más enteros de 16 bits.
.long	Inicializa uno o más enteros de 32 bits.
.string	Inicializa uno o más cadenas de texto.
.ubyte	Inicializa uno o más palabras sucesivas sin signo.
.ulong	Inicializa uno o más enteros sin signo de 32 bits.
.uword	Inicializa uno o más enteros sin signo de 16 bits.
.word	Inicializa uno o más enteros signados de 16 bits.

resultado o son espacios de memoria para almacenamiento temporal, sin embargo, es necesario reservar el espacio de memoria correspondiente para poder utilizarlo. Por lo tanto, en la Tabla 2.4 se muestran las directivas que se pueden utilizar para reservar memoria.

Tabla 2.4: Directivas que reservan espacio en la memoria.

Mnemónico	Descripción
.bes	Reserva N bits en la sección actual; una etiqueta apunta al final del espacio reservado
.space	Reserva N bits en la sección actual; una etiqueta apunta al principio del espacio reservado

La sintaxis de estas directivas es la siguiente:

Nombre de variable **mnemónico** *Tamaño* *;Comentarios*

Declaración de símbolo como constante

La declaración de un símbolo constante es ampliamente utilizado en un programa fuente en ensamblador. Los símbolos constantes son valores que se declaran al inicio del programa y se mantienen constante a lo largo de todo el código, es decir, no se pueden cambiar, de tal manera que se pueden asignar nombres significativos a expresiones constantes como es el caso del número π o la longitud de una señal [11]. El mnemónico utilizado para asignar un

símbolo a una constante es **.set** y su sintaxis es la siguiente:

Nombre de la constante **.set** *valor* *;comentarios*

Directivas de símbolos y final del programa

En la siguiente Tabla 2.5 se muestran las directivas que se utilizan en los ejemplos a lo largo del presente trabajo para indicar la sección principal del código y el final del mismo.

Tabla 2.5: Directivas de símbolos y final de programa

Mnemónico	Descripción
.def	Identifica uno o más símbolos que son definidos en el módulo actual y que pueden ser utilizados en otros módulos.
.global	Identifica uno o más símbolos globales (externos)
.ref	Identifica uno o más símbolos utilizados en el modulo actual que son definidos en otro módulo.
.end	Fin de ensamblaje

La sintaxis utilizada para las directivas de símbolos es la siguiente:

mnemónico *símbolo*

2.5. Compilación y ejecución de un proyecto

Después de crear y emplear un algoritmo en código en CCS, el siguiente paso es compilar y verificar que éste no contenga errores. Se conoce como *Debugging* al proceso de depuración o identificación y corrección de errores. El CCS en éste proceso, verifica que se haya realizado el ensamble correctamente del código con la descripción de memoria (archivo *.cmd*), así como la configuración de puertos y errores de sintaxis. CCS genera el archivo ejecutable cuando se comprueba que tanto la configuración como el código son correctos.

En la parte superior de la ventana *CCS Edit* del *Code Composer Studio* (ventana de edición del programa) se pueden observar dos iconos llamados *Build* y *Debug* como se muestra en la Figura 2.18. El icono *Build* sirve depurar el programa y construir el ejecutable del mismo, es decir, verifica que no exista algún error en el código, configuración y ensamble con el mapa de memoria. Si existen errores, estos se mostrarán en la ventana de notificaciones. Es necesario resolver los problemas que muestre el compilador, ya que de no ser así, el programa no se podrá ejecutar en el DSP. Después de realizar las correcciones, será necesario compilar

el proyecto nuevamente.

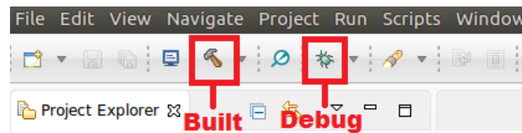


Figura 2.18: Herramienta *'Build'* y *'Debug'*

Una vez resueltos los problemas marcados por el compilador y obtener una compilación exitosa del proyecto, es necesario ejecutar el programa en la tarjeta mediante el icono de *Debug* (Figura 2.18), tomando en cuenta que se tuvo que compilar previamente parra que el compilador construya el ejecutable. Al presionar el botón, aparecerá una ventana emergente que notifica al usuario que se está cargando el proyecto e inmediatamente cambia la interfaz de *CCS Edit* a *CCS Debug* como la que se muestra en la Figura 2.19.

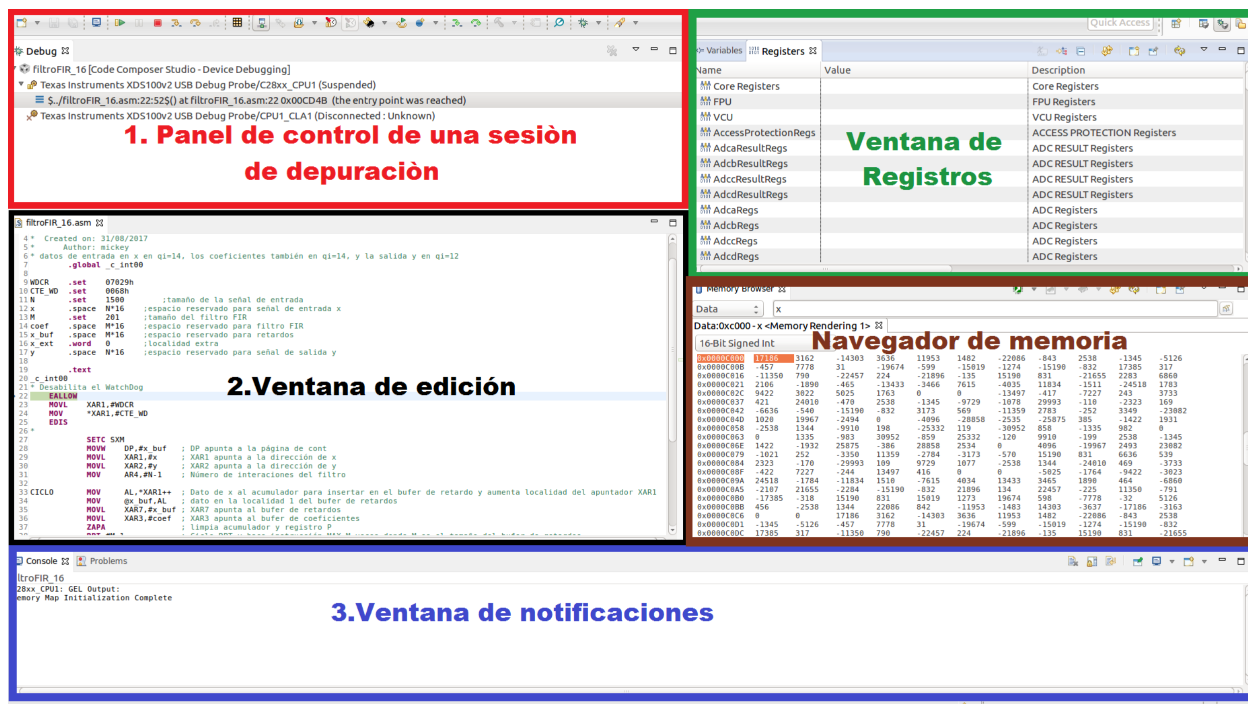


Figura 2.19: Interfaz *CCS Debug* al cargar el programa

Se pueden observar cinco diferentes secciones en la interfaz de *CCS Debug* de la Figura 2.19, las cuales son necesarias para monitorear el comportamiento del sistema, las cuales se

componen de:

1. Panel de control de una sesión de depuración. Cuando se está en una sesión de depuración o *debug*, permite manipular la forma de ejecución del programa en el DSP pausando, reiniciando, ejecutando paso a paso, etcétera.

2. Ventana de edición. Es la ventana donde se tiene acceso a los archivos del proyecto y durante la ejecución de algún programa paso a paso, muestra el lugar en donde el programa se va ejecutando.

3. Ventana de notificaciones. Muestra la actividad del CCS y notificaciones respecto a la compilación de código o la comunicación con el DSP.

4. Ventana de registros. Permite monitorear, y en algunos casos modificar los registros del DSP ya sean del CPU, interrupciones, módulo ADC, módulo PWM, entre otros.

5. Ventana de memoria. Muestra el mapa del contenido de la memorial del DSP.

Finalmente, el programa se puede ejecutar de dos maneras, instrucción por instrucción mediante el icono *Step Into* o presionando la tecla *F5*, o bien puede ejecutarse de forma continua mediante el icono *Resume* o presionando la tecla *F8*. Estos íconos se encuentran en la parte frontal de la ventana *CCS Debug* como se muestra en la Figura 2.20

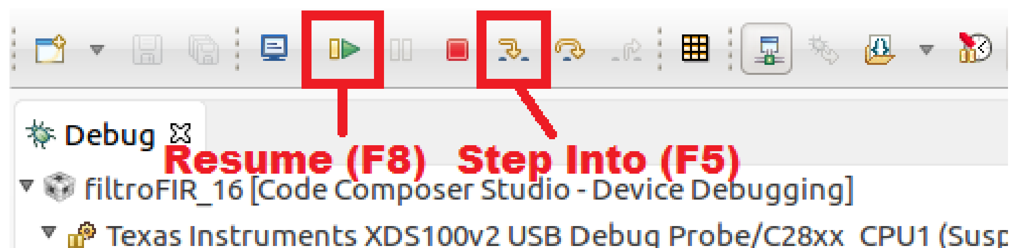


Figura 2.20: Herramienta *'Resume'* (F8) y *'Step Into'* (F5) para ejecución de código

2.6. Herramientas de acceso a memoria y visualización de datos en CCS

El *Code Compose Studio* cuenta con diferentes herramientas que facilitan al programador visualizar datos y el proceso de los mismos. Las dos herramientas más utilizadas en la ejecución de los ejemplos en el presente trabajo son el acceso a los datos de la memoria y

visualización por medio de gráficas una secuencia de datos alojados en la memoria.

El acceso a los datos de la memoria del dispositivo se hace por medio del *Memory Browser* y se puede abrir por medio de la ruta *View* → *Memory Browser* desde la ventana *CCS Debug*. Esto permite visualizar los datos contenidos a lo largo de la memoria, ingresando la dirección que se desea visualizar. De la misma manera se puede seleccionar la forma de visualizar dichos datos, ya sea en palabras de 16 o 32 bits en formatos binario, hexadecimal, flotante o entero signado o sin signo, además del Qi de interés cuando se trabaja en formato de punto fijo.

Además de visualizar datos de la memoria, también una herramienta muy importante en *Code Composer Studio* es poder cargar datos a la memoria del dispositivo desde un archivo de texto con extensión *.dat* y de la misma manera exportar los datos alojados en la misma. A continuación se explica como importar datos externos a la memoria del dispositivo y las características que debe contener el archivo para que sea compatible.

2.6.1. Importación de datos a la memoria

En algunos de los programas de ejemplo de este trabajo se emplean secuencias de datos (señales de prueba, coeficientes de filtros, etc.) para probar el funcionamiento de las operaciones y algoritmos de procesamiento digital programados, por esta razón, a continuación se aborda la forma de importar datos externos a la memoria del DSP TMS320F28377S.

En primera instancia, todos los archivos de datos que se deseen importar deben tener el encabezado como el mostrado a continuación:

No. clave	**Formato**	**Dir. de inicio**	**No. de página**	**No. de datos**

donde:

No. clave; es un indicador de *Texas Instruments* igual a 1651, este número es necesario para que Code Composer Studio identifique que el archivo contiene datos a cargar en la memoria.

Formato; indica el formato en el cual están los datos a cargar en la memoria, este campo toma valores del 1 al 4 dependiendo del tipo de formato como se muestra en la Tabla 2.6.

Dir. de inicio; es la dirección de la localidad de memoria, donde se comenzarán a escribir los datos contenidos en el archivo. Este campo depende directamente de la definición del mapa de memoria que haya sido declarado en el archivo *.cmd* y del orden en que hayan sido declaradas las variables de tipo arreglo, para alojar los datos del archivo. Este campo debe escribirse en hexadecimal.

Tabla 2.6: Formatos de datos soportados por CCS para importarse a la memoria del DSP.

Formato	Tipo	Descripción
1	hex	Datos en formato hexadecimal
2	int	Datos en formato entero con longitud de palabra de 16 bits
3	long	Datos en formato entero con longitud de palabra de 32 bits
4	float	Datos en formato flotante

No. de página; es la página de memoria donde se alojan las localidades destinadas a contener cada dato del archivo *.dat*. La memoria del DSP se distribuye en páginas y posteriormente en secciones.

No. de datos; es la cantidad total de datos a cargar en la memoria, este campo debe escribirse en hexadecimal.

A continuación se muestra un ejemplo de como debe estar escrita la primer línea de un archivo *.dat* para cargar los datos contenidos a la memoria del DSP. Entre cada campo existe un espacio de separación.

1651 2 1A034 1 041A

Una vez listo el archivo con el encabezado anterior, para cargar los datos a la memoria se tiene que tener listo un programa compilado y tener conectada la tarjeta Launchpad-F28377S a un puerto USB de la PC, para poder entrar al modo *Debug*, dando click derecho indicado en el icono cuyo dibujo es un escarabajo. Finalizada la escritura del programa en el DSP, como parte del entorno del modo *Debug* deberá aparecer a la derecha de la pantalla, una ventana llamada *Memory Browser* la cual permite ver el contenido de la memoria y escribir datos en ella. Ubicada la ventana *Memory Browser*, se debe de dar click derecho al icono resaltado en la Figura 2.21, para desplegar el menú de opciones y seleccionar *Load Memory* para que se despliegue la ventana mostrada en la Figura 2.22a.

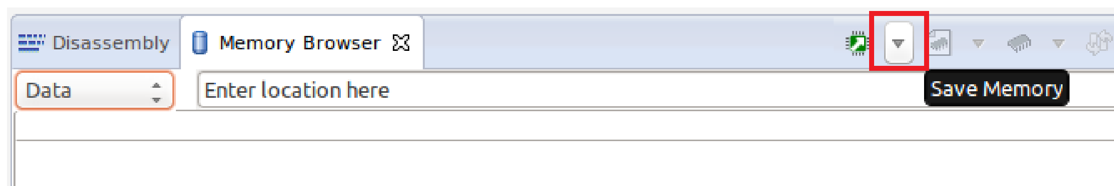
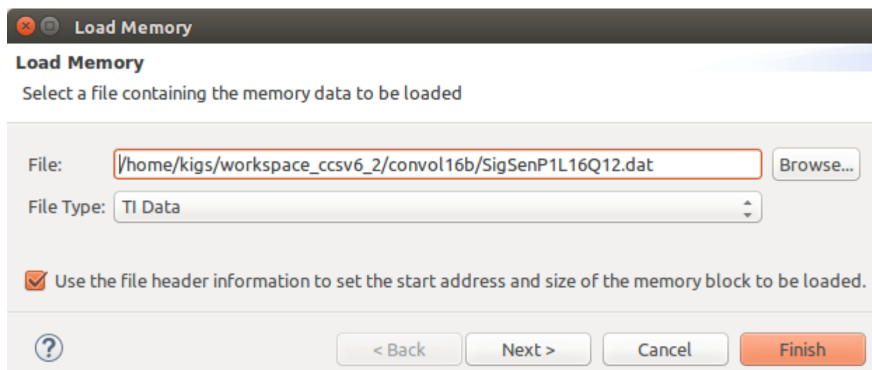
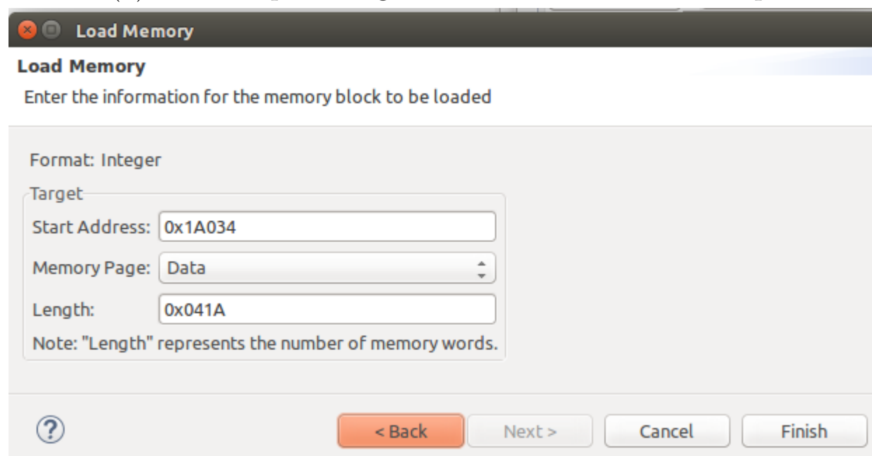


Figura 2.21: Visualizador y editor del contenido de la memoria.

Con el botón de *Browser* se accede al explorador de archivos para poder buscar el archivo que nos interesa, y adicionalmente en la parte inferior izquierda tenemos la opción de habilitar para que lea la información de la primera línea de nuestro archivo y configure la exportación de archivos, en caso de habilitarla, al presionar el botón *Next* nos mostrará la configuración leída del archivo, en caso de no habilitar dicha opción, manualmente se tendrá que ingresar la configuración.



(a) Ventana para cargar datos a la memoria del dsp.



(b) Ventana para cargar datos a la memoria del DSP.

Figura 2.22: Cargar datos a la memoria del DSP.

En la Figura 2.22b se muestra la configuración leída del encabezado del archivo de datos *.dat* seleccionado. Se puede observar que los datos son de tipo entero, la dirección de inicio, la página en donde se encuentra la dirección inicial y la cantidad de datos a cargar. Al presionar el botón *Finish* la IDE CCS cargará los datos a la memoria del DSP.

2.6.2. Visualización de datos

Cuando se carga un programa al DSP y se ejecuta desde *Code Composer Studio*, es posible construir gráficas para visualización de los datos almacenados en la memoria del DSP. La comunicación entre la tarjeta y el software es por medio de la conexión USB. Las gráficas se generan a partir de una cantidad de datos seleccionados en la memoria del DSP especificando el formato de datos utilizado. El Software puede generar hasta seis tipos de gráficas diferentes tanto en el dominio del tiempo, como en el dominio de la frecuencia.

Para generar una gráfica a partir de los datos de la memoria es necesario asegurarse de que el programa se esté ejecutando, posteriormente seleccionar la ruta *Tools*→ *Graph* desde las opciones que se encuentran en la parte superior del software, como se muestra en la Figura 2.23.

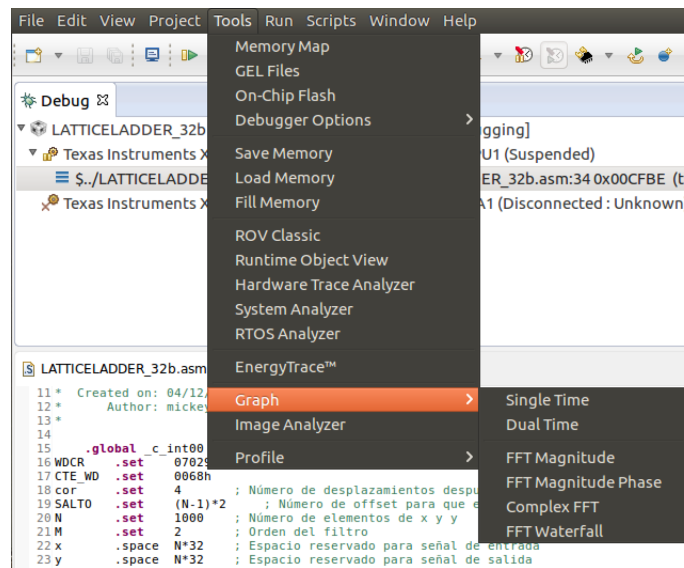


Figura 2.23: Creación de gráfica.

Para graficar una secuencia de tiempo es necesario seleccionar la opción *Single Time*, posteriormente emergerá una ventana como la que se muestra en la Figura 2.24a donde el usuario asigna las características de los datos que contendrá dicha gráfica. Los datos más importantes a ingresar son:

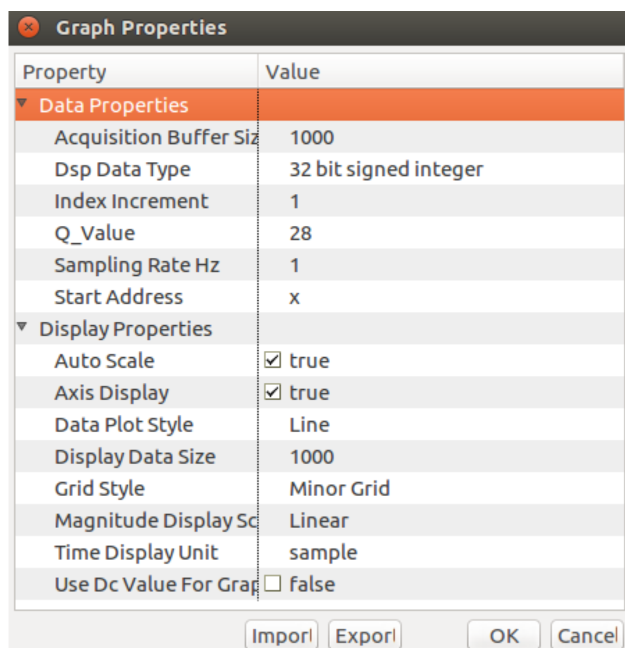
1. Tamaño del buffer de los datos a graficar.
2. Tipo de datos, éstos pueden ser de 64, 32, 26 y 8 bits, enteros signados, sin signo o flotantes.
3. El valor del Qi para el caso de números de punto fijo.

4. La dirección inicial de los datos en la memoria del DSP.

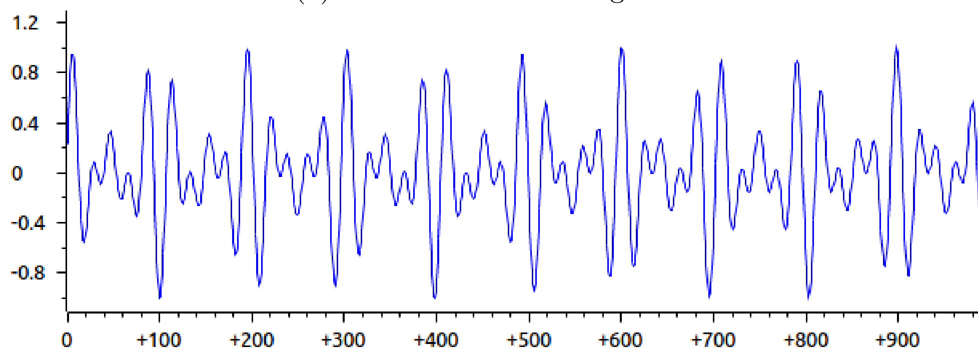
5. Número de datos que el usuario desea visualizar.

6. Cuadrícula y su tamaño.

En la Figura 2.24b se muestra el ejemplo de una gráfica generada con una señal con 1000 localidades de memoria, con datos de 32 bits enteros signados en Qi=28.



(a) Características de la gráfica.



(b) Gráfica de una secuencia de tiempo en *Code Composer Studio*.

Figura 2.24: Gráficación de una secuencia de tiempo en *Code Composer Studio*.

Cuando la interfaz del *Code Composer Studio* genera una gráfica y la ubica en una ventana localizada en la parte inferior de la interfaz, es decir, en la misma sección de notificaciones (ver Figura 2.19). Sin embargo el usuario puede acomodar las ventanas como lo desee.

Para el caso de las gráficas con la opción *Dual time*, genera dos secuencias de tiempo con las mismas características indicando las direcciones de inicio de los datos de cada secuencia.

Para las gráficas en el dominio de la frecuencia el software puede generar las gráficas *FFT Magnitude FFT*, *FFT Magnitude Phase*, *Complex FFT* (real e imaginario de la FFT) y *FFT Waterfall*. Al generar alguna de estas gráficas se requiere la misma información base, es decir:

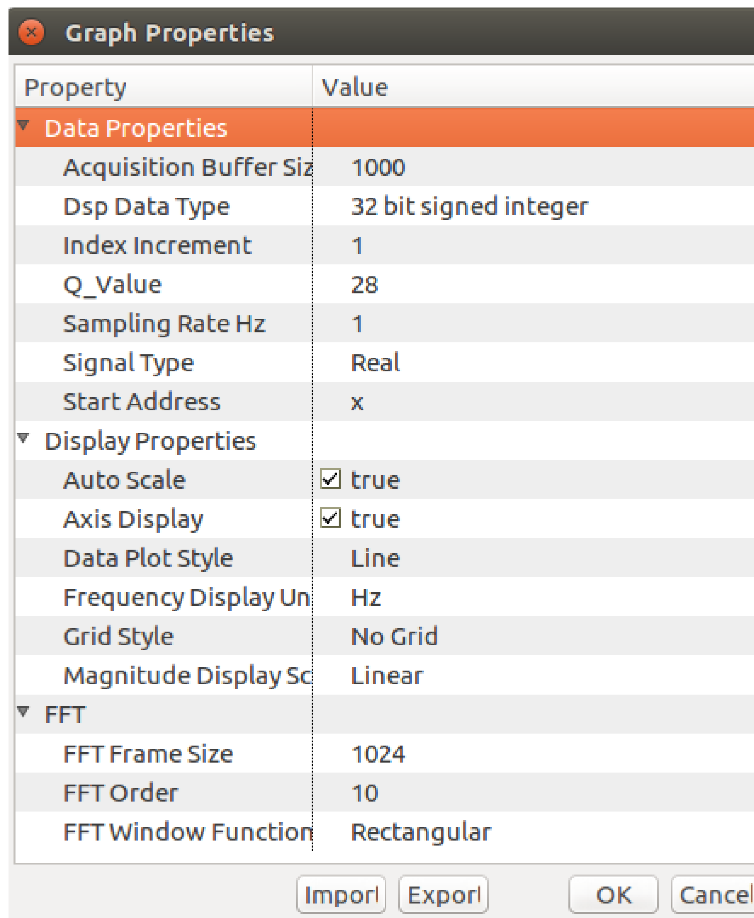
1. Tamaño del buffer de los datos para la gráfica.
2. Tipo de datos, éstos pueden ser de 64, 32, 26 y 8 bits, enteros signados, sin signo o flotantes.
3. El valor del Q_i para el caso de números de punto fijo.
4. Tipo de señal.
5. Dirección de inicio de los datos (para el caso de las gráficas de magnitud y magnitud y fase).
6. Dirección de inicio de los datos reales y dirección de inicio de los datos imaginarios (solo en caso de la gráfica *Complex FFT*).
7. Orden de la FFT.
8. Tipo de ventana para la FFT.

En la Figura 2.25a se muestra la ventana de selección de características para generar una gráfica de la Magnitud de la FFT de un buffer de datos, mientras que en la Figura 2.25b se muestra un ejemplo de la gráfica de magnitud de la FFT de la señal mostrada en la Figura 2.24b por medio de *Code Composer Studio*.

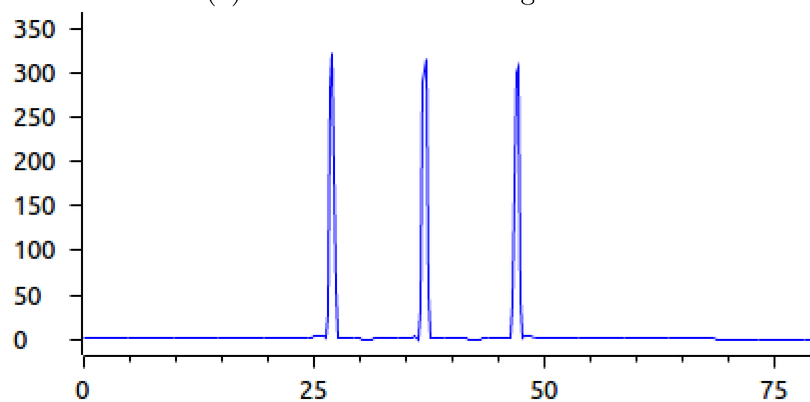
Dependiendo de la configuración de la gráfica, el eje de las abscisas puede verse en Hertz, para ello es necesario indicar la frecuencia de muestreo de la señal en las configuraciones de la gráfica (Figura 2.25a), de no ser así, la frecuencia que representa dicha gráfica es la frecuencia normalizada, de tal manera que esta dada por:

$$\omega = \frac{2\pi f}{f_s} \tag{2.1}$$

donde f es la frecuencia en Hz, y f_s es la frecuencia de muestreo.



(a) Características de la gráfica.



(b) Magnitud de la FFT de una señal en *Code Composer Studio*.

Figura 2.25: Gráficación de la magnitud de la FFT en *Code Composer Studio*.

2.7. Resumen del capítulo

El DSP empleado a lo largo del presente trabajo fue diseñado por *Texas Instruments*, de tal manera que el Software utilizado como herramienta de programación es el *Code Composer Studio*. En este capítulo se ha realizado una explicación de las funciones básicas de dicho software, mostrando la creación de un proyecto utilizando la LAUCHXL-F28377S tanto en lenguaje ensamblador como en lenguaje C y utilizando un mapa de memoria (archivo .cmd). Además, se mostró la utilidad y configuración de CONSTROLSUITE como herramienta de configuración para periféricos y programación en lenguaje C.

Se han mostrado las herramientas con las que cuenta el programa y los dos entornos que éste contiene (CCS Edit y CCS Debug), además de las herramientas para visualización de la memoria (Memory Browser), registros, variables y graficación en el dominio del tiempo y de la frecuencia. En muchas aplicaciones es necesario ingresar los datos de señales o coeficientes a la memoria del DSP, para después acceder a estos datos y realizar el procesamiento respectivo, para ello se explicó las características que debe contener un archivo que contiene los datos a importar en la memoria del DSP.