CAPÍTULO IV

INTRODUCCIÓN

Codewarrior (CW) IDE es un completo sistema para editar, compilar y enlazar, simular, programar y depurar para los DSC de Freescale.

La versión 8.0 del CW IDE está compuesto por un editor, ensamblador, compilador C/C++ optimizado, enlazador, simulador, programador y depurador.

En el paquete se incluye el UNIS Processor Expert, el cual es una herramienta que automatiza la generación de código, configuración de periféricos, dispositivos externos y algoritmos.

ENTORNO DE DESARROLLO INTEGRADO-IDE

El IDE es una aplicación de software que integra la mayoría de las herramientas de programación en una única pieza de software.

Más adelante se verá cómo crear, simular y depurar un proyecto en la familia DSP56F8XXX.

Luego de lanzar la aplicación aparece el IDE como la que se muestra en la Figura 1. Esta ventana de arranque le permite al usuario crear un nuevo proyecto, correr un ejemplo de demostración, cargar un proyecto ya existente o simplemente usar el IDE.

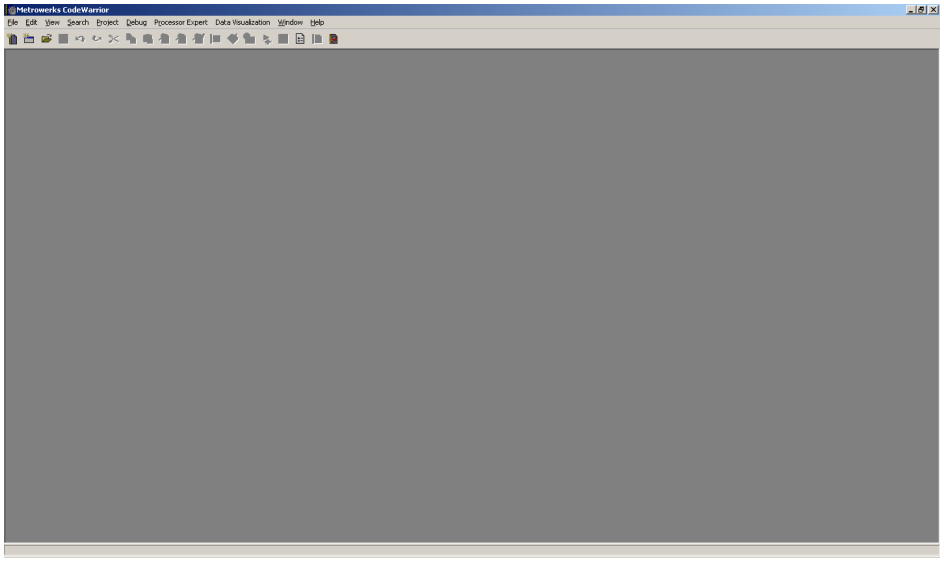


Figure 1 – CW IDE pantalla principal

CREANDO UN NUEVO PROYECTO

Para realizar un nuevo proyecto en CW, presiones seleccione la opción ***FILE->NEW*** del menú principal o  icono de la barra de herramienta. Una nueva ventana emergente aparece y es el asistente de proyecto, que le permite al usuario seleccionar, el tipo de proyecto, el dispositivo, el nombre y su ubicación.

Los diferentes tipos son (ver Figura 2):

- EABI Stationery: El mismo le permite crear un proyecto para las placas de referencia de Freescale con el estándar EABI para tener compatibilidad con OS principalmente.
- EVM Example Stationery: Esta opción le permite crear un proyecto simple para las boards de referencia de Freescale y obtenemos una aplicación funcionando es el mejor punto de partida.
- NEW Project Wizard: es un asistente para la creación de proyecto genérico.
- Empty Project: Crea un proyecto vacio sin definiciones específicas de una placa en particular.
- Makefile Importer Wizard: este tipo de proyecto se utiliza para pasar un proyecto basado en makefile a un proyecto manejado en el IDE.
- Processor Expert Examples Stationery: Crea proyectos con ejemplos listos para las placas de referencia dentro del entorno de Processor Expert.
- Processor Expert Stationary: Sirve para crear proyectos con Processor Expert no solamente para una placa de referencia.

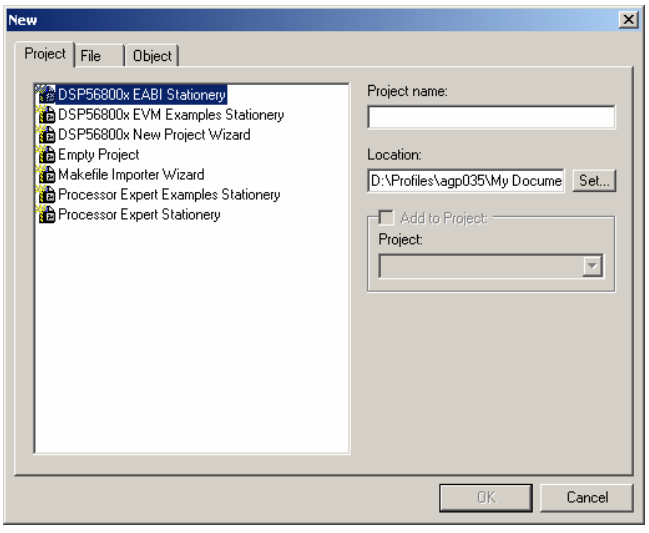


Figure 2- Creación de nuevo proyecto

Si seleccionamos ***DSP56800X New Project Wizard*** nos aparece la siguiente ventana, para seleccionar la subfamilia de DSC y el procesador especifico de esa subfamilia (ver Figura 3).

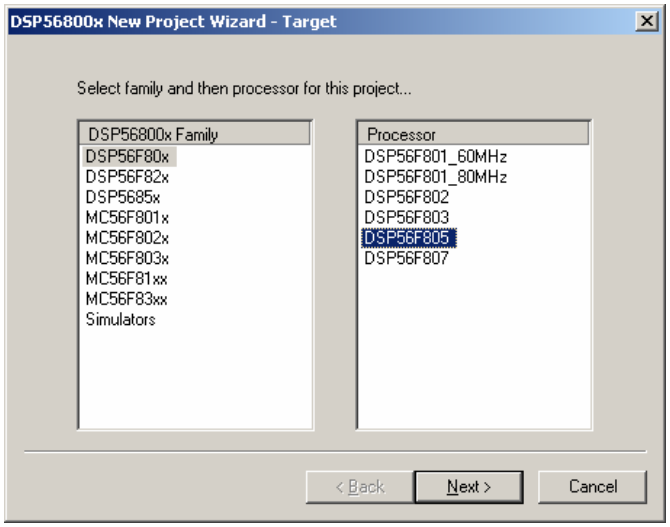


Figure 3- Selección del procesador para el proyecto.

El IDE de CW soporta tres tipos de lenguajes: Assembly, C y C++ también es posible hacer la mezcla de ellos en el mismo proyecto la opción de C++ está habilitada en la versión profesional (ver Figura 4).

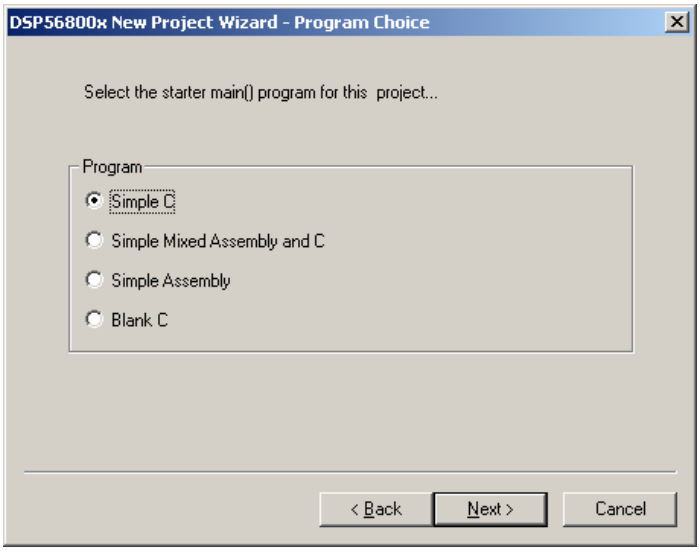


Figure 4 – Selección del Lenguaje a Utilizar en el proyecto.

Después de seleccionar el lenguaje del proyecto (o lenguajes si hay más de uno seleccionado), es necesario que definamos la locación del código generado si va a estar en memoria interna o memoria externa o dar la posibilidad de trabajar con ambas (ver Figura 5).

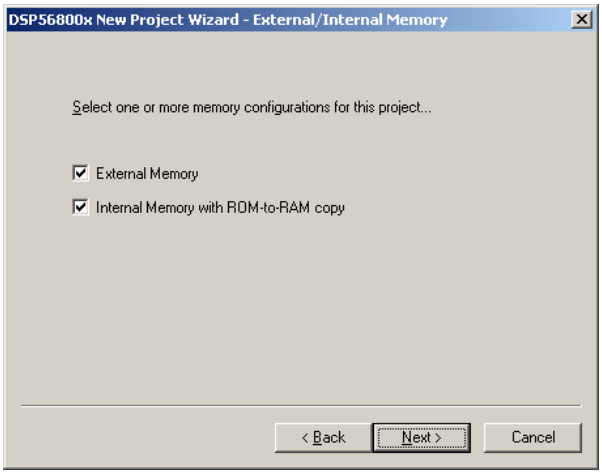


Figure 5 – Selección de Memoria Interna o Externa.

Ahora ya está listo para crear el proyecto y hay que presionar el botón de FINISH como se muestra en la siguiente Figura 6.

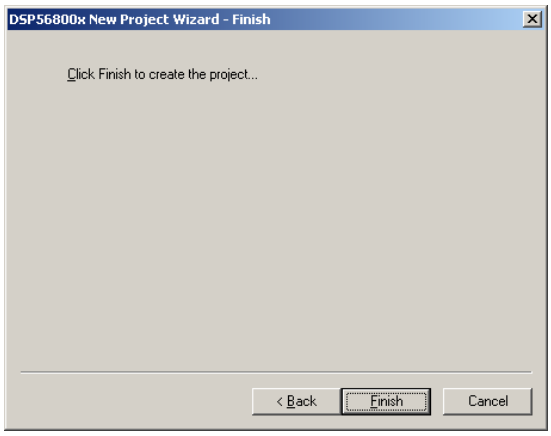


Figure 6 – Generar el proyecto de acuerdo a las opciones previamente seleccionadas.

La ventana de proyecto consiste de una lista de archivos pertenecientes al proyecto. Algunos archivos importantes son: DSP56F805_init.c que es un archivo automáticamente generado por el IDE y contiene toda la secuencia de inicialización de la CPU, DSP56F805_external_mem_linker.cmd que es el archivo que contiene todos los comandos asociados al enlazador describiendo todos los mapas de memoria, MSLC56800.lib y FP56800.lib son las librerías para el lenguaje C. En la Figura 7 se muestra una imagen de la ventana del proyecto.

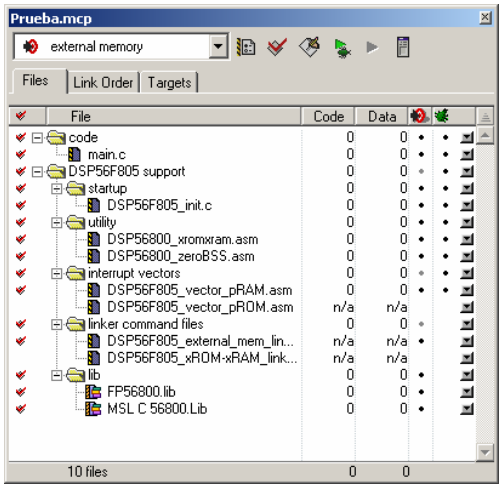


Figure 7 – Proyecto generado a partir del asistente de proyecto.

A continuación vemos los detalles de la ventana de proyecto y sus funcionalidades (Figura 8)

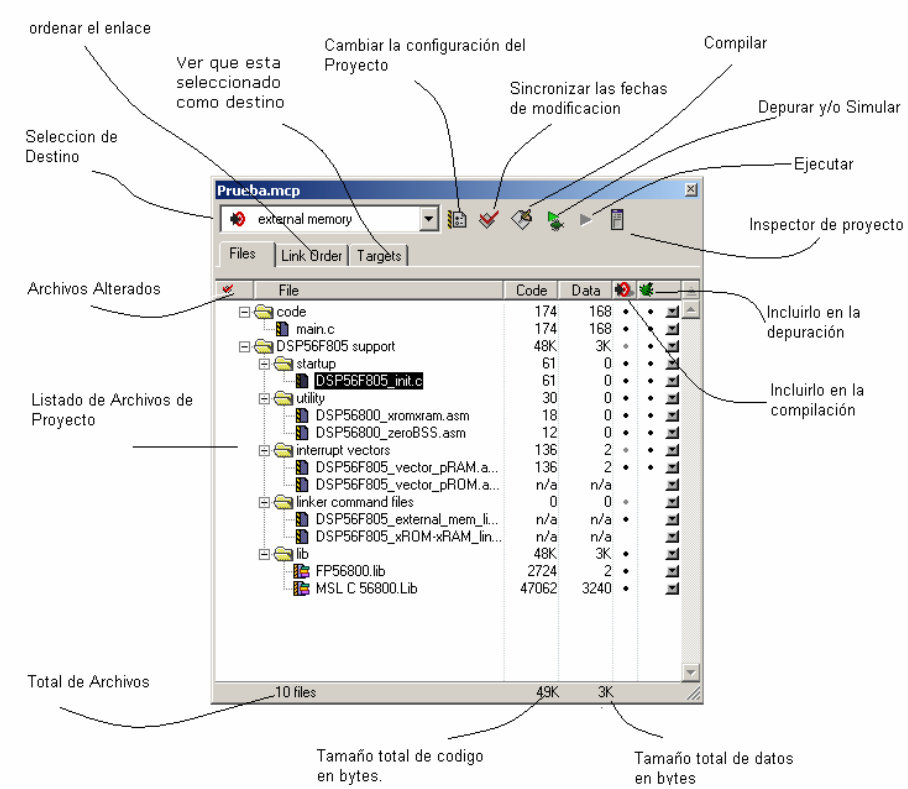


Figure 8 - Funcionalidades de la ventana de gerenciamiento de proyecto.

Para poder depurar un archivo, el mismo tiene que estar marcado, de lo contrario no se generan los símbolos para la depuración. A continuación se listan todas las extensiones de archivos que maneja el CW, por medio de la Tabla 1.

Tabla 1: Extensiones de archivos que maneja el CW

Type	Extension	Explanation
Minimum CodeWarrior Installation	.iSYM	CodeWarrior Intel® Symbols
	.mch	CodeWarrior Precompiled Header
	.mcp	CodeWarrior Project File
	.SYM	CodeWarrior Mac OS 68K Debug Symbols
	.xSYM	CodeWarrior Mac OS PPC Debug Symbols
	.dbg	CodeWarrior Debug Preferences
	.exp	Exported Symbol File
	.iMAP	CodeWarrior Link Map
	.MAP	CodeWarrior Link Map
Assembly	.a	Assembly Source File (Windows and Macintosh)
	.asm	Assembly Source File
	.dump	CodeWarrior Disassembled File
C and C++	.c++	C++ Source File
	.cc	C++ Source File
	.hh	C++ Header File
	.hpp	C++ Header File
	.i	C Inline Source File
	.icc	C++ Inline Source File
	.m	Object C Source File
	.mm	Object C++ Source File
Default C and C++	.c	C Source File
	.cp	C++ Source File
	.cpp	C++ Source File
	.h	C and C++ Header File
Default Java	.class	Java Class File
	.jar	Java Archive File
	.jav	Java Source File
	.java	Java Source File
Java	.JMAP	Java Import Mapping Dump
	.jpob	Java Constructor File
	.mf	Java Manifest File
Library	.a	(Static) Archive Library (Solaris and Linux)
	.lib	Library File
	.o	Object File (Windows and Macintosh)
	.o	Object (Relocatable) Library or Kernel Module (Solaris and Linux)
	.obj	Object File
	.pch	Precompiled Header Source File
	.pch++	Precompiled Header Source File
	.so	Shared Library (Linux)
Script	.sh	Shell Script (Linux)
	.psh	Precompile Shell Script (Linux)
	.pl	Perl Script (Linux)
Mac OS X	.dylib	Mach-O Dynamic Library
	.a	Mach-O Static Library
	.o	Mach-O Object File
	.plist	Property List

EDITANDO ARCHIVOS

Editaremos un archivo para empezar usaremos el main.c y para ello hacemos doble-click sobre el nombre del archivo en la ventana de proyecto, y nos aparecerá la ventana de edición de código como se muestra en la Figura 9.

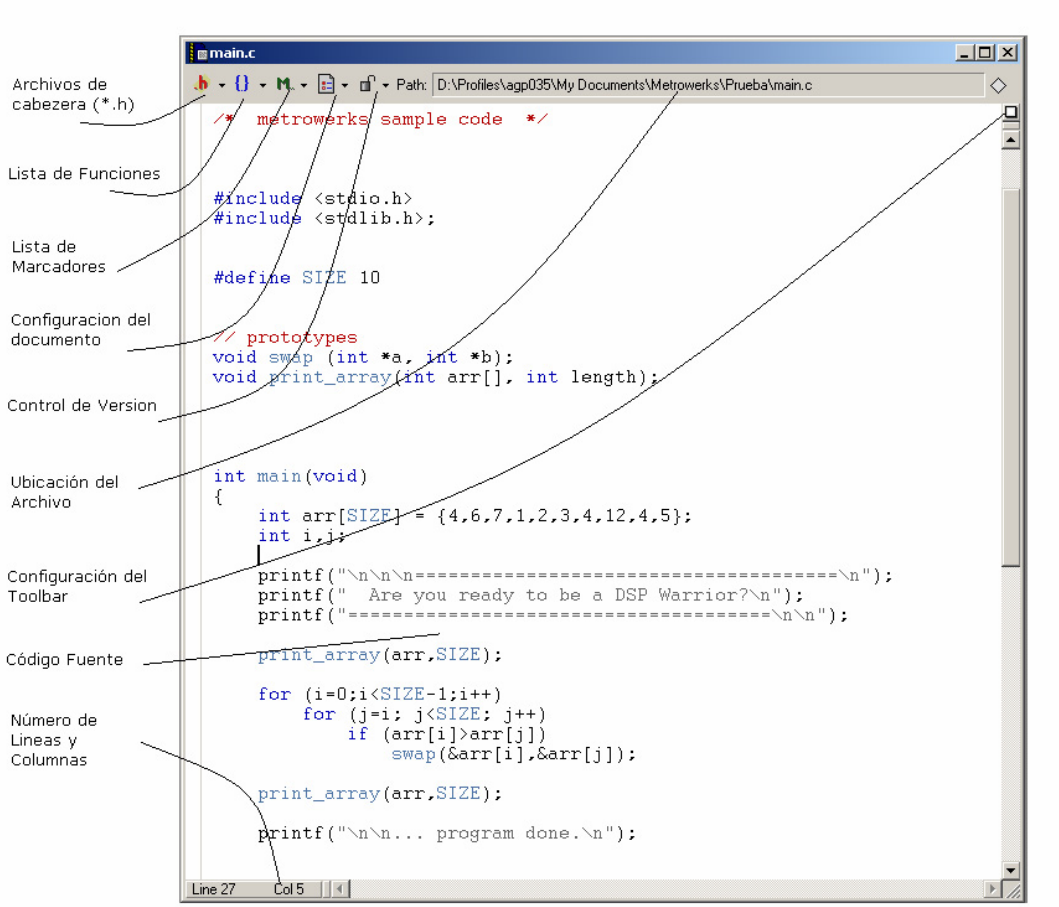


Figure 9 – Funcionalidades de la ventana de edición.

Luego de hacer una compilación completa obtenemos la ventana con los errores/advertencia/hallazgos con la cual nos va a permitir navegar para encontrar los problemas con una breve descripción de los mismos con su código de error para poder buscar su explicación más detallada además contando con un botón para ello y también botones de navegación como se ve en la Figura 10.

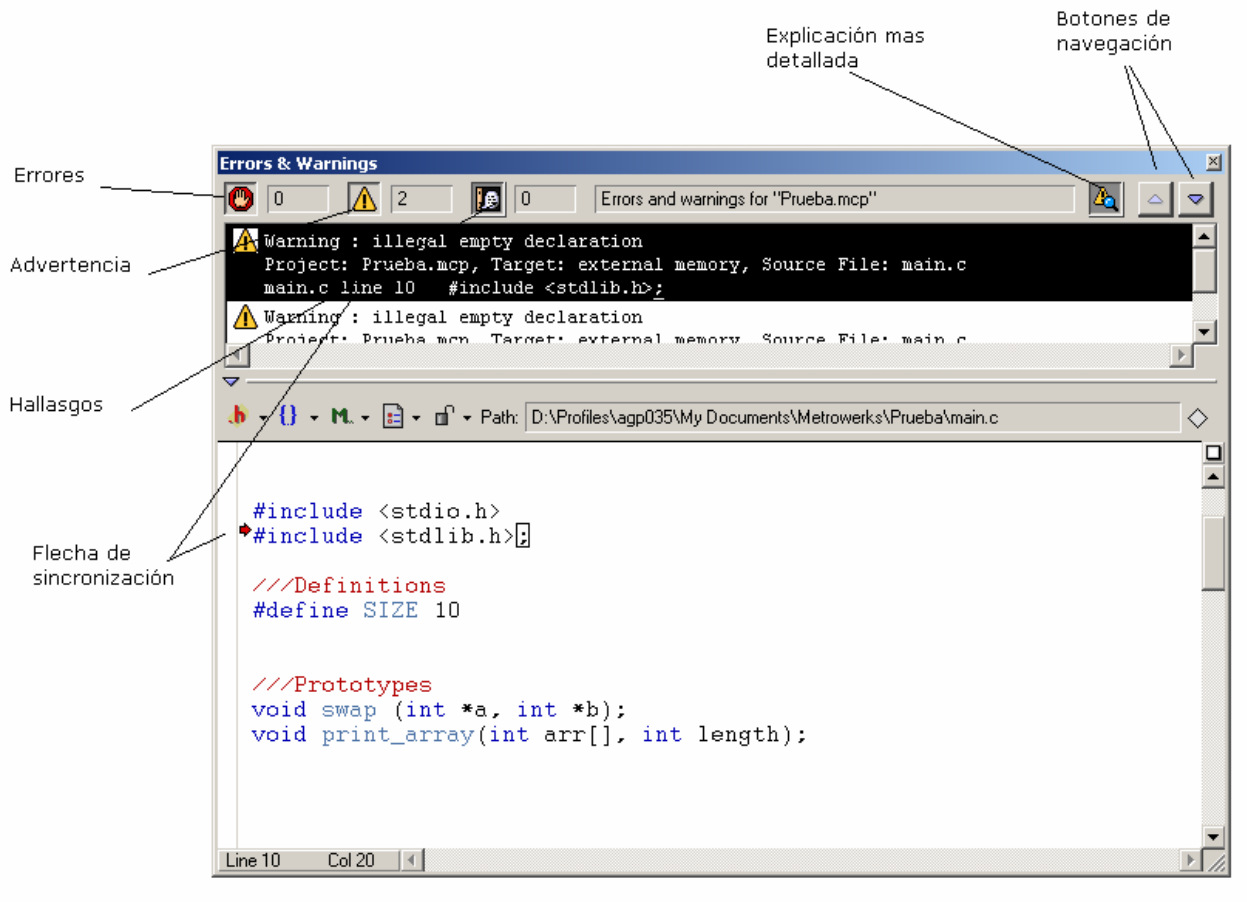


Figure 10 – Ventana de navegación de errores.

Si se detectan errores no se generan los archivos binarios. El compilador también genera las advertencias y los hallazgos si algún código dudoso es encontrado. Si no se detectaron errores el archivo .elf es generado en el subdirectorio “output” dentro de la estructura del proyecto.

CONFIGURANDO PROYECTOS

Ahora se verán las ventanas de configuración de proyecto como se muestran en la Figuras 11 a 21. Como puede observarse hay muchas opciones que se tendrán que revisar y seleccionar de acuerdo a la naturaleza y necesidades especificas del proyecto. La principales son Target Settings, M56800 Target, C/C++ Language, C/C++ Warning, M56800 Assembler, ELF Disassembler, M56800 Processor, Global Optimization, M56800 Linker, Remote Debugging and M56800 Target Settings.

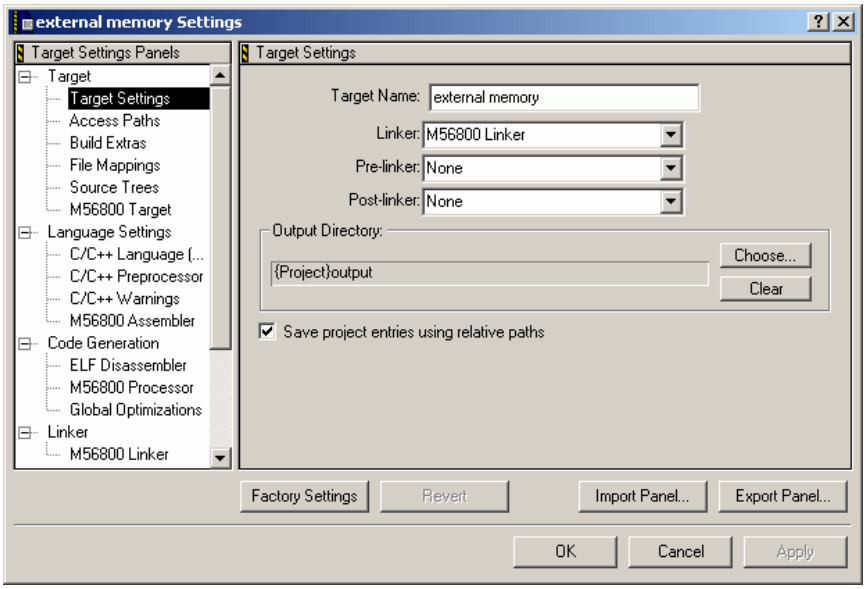


Figure 11- Ventana inicial de configuración del proyecto

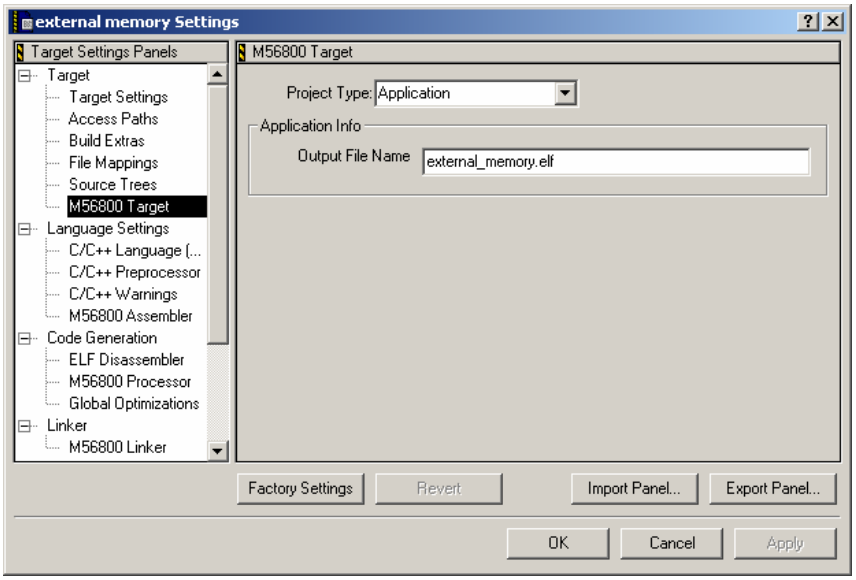


Figure 12- Ventana de configuración para seleccionar si lo que vamos a compilar es una aplicación o una librería.

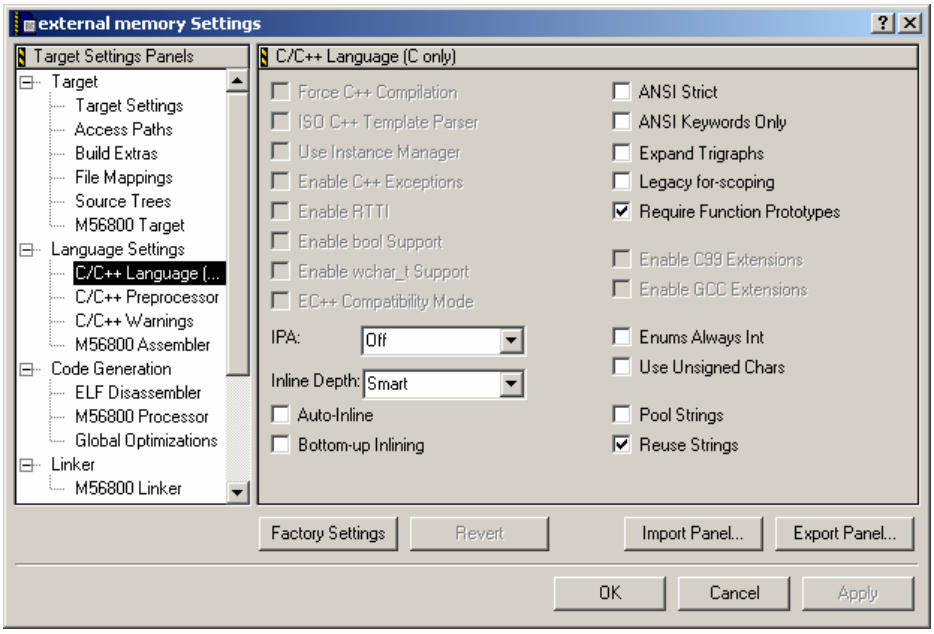


Figure 13- Ventana de configuración del lenguaje C

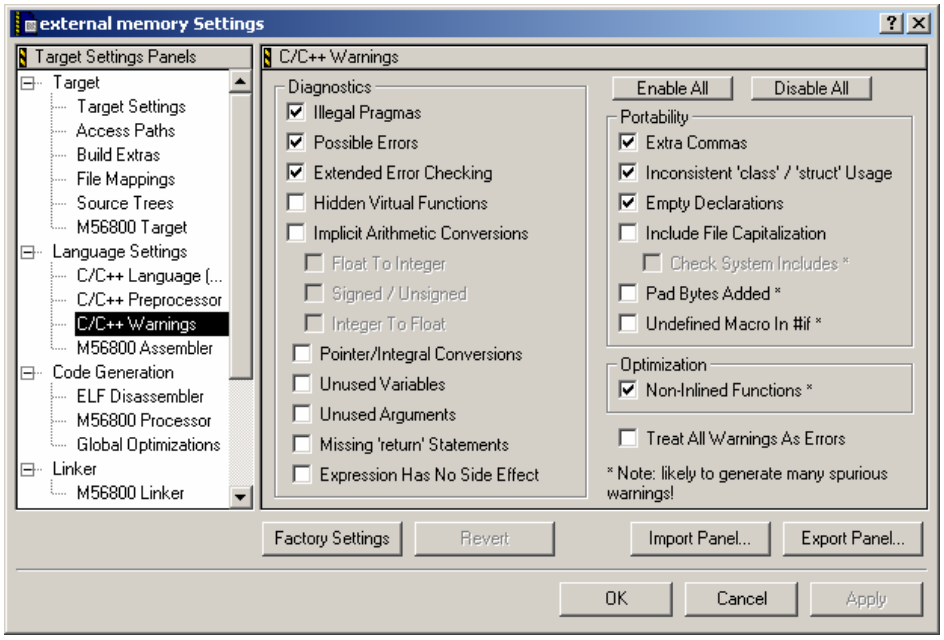


Figure 14-Ventana de configuración de los reportes de advertencias.

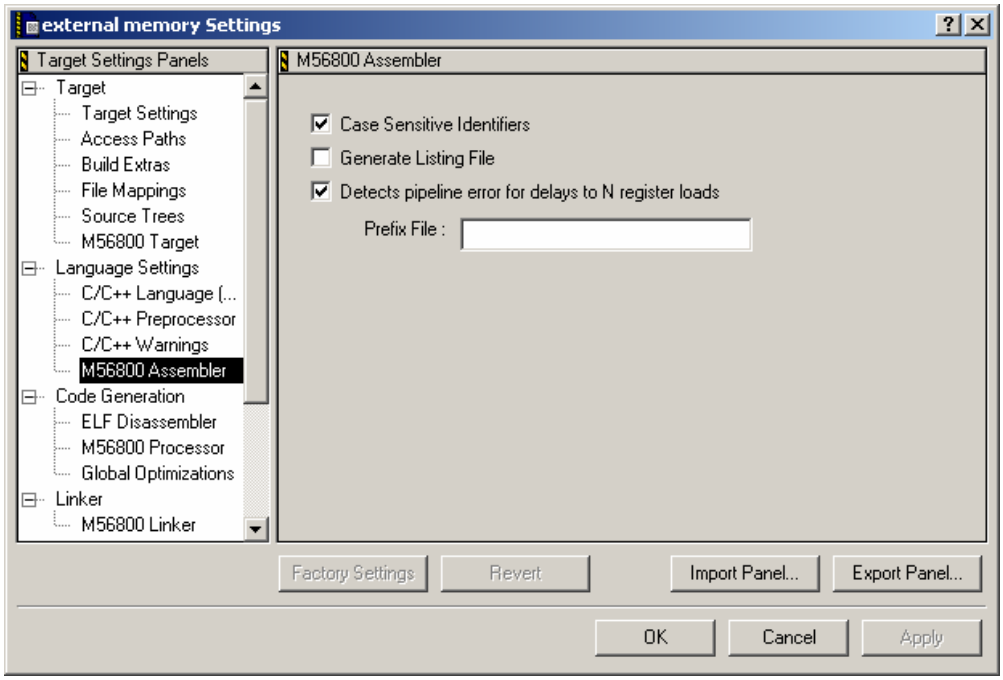


Figure 15-Ventana de configuración del Assembler y lo importante es la generación del listing file.

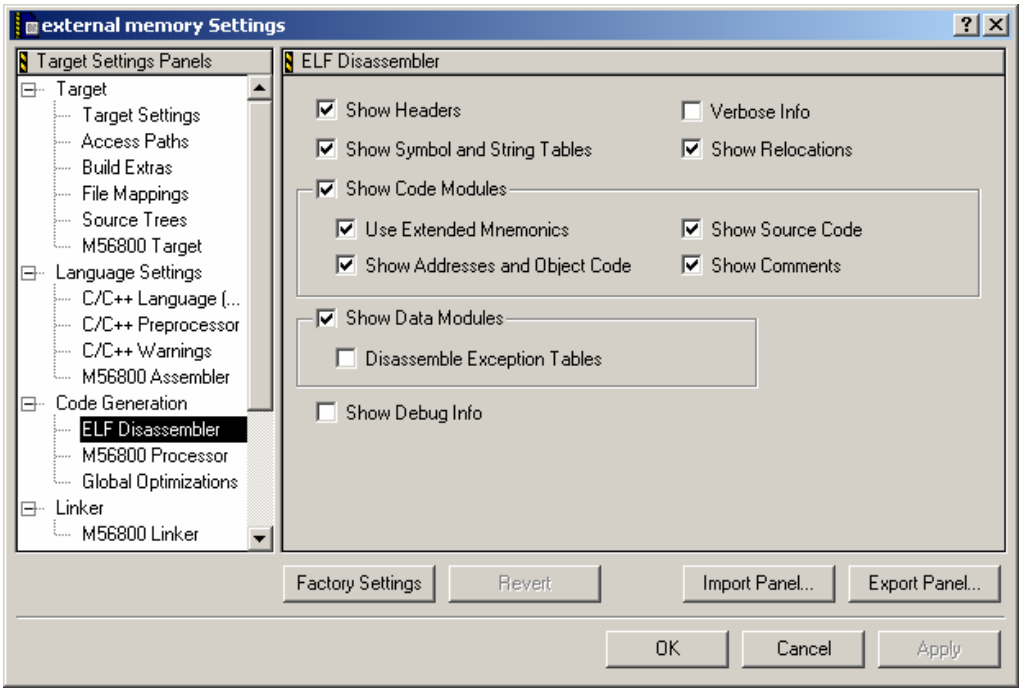


Figure 16 – Ventana de configuración del binario generado.

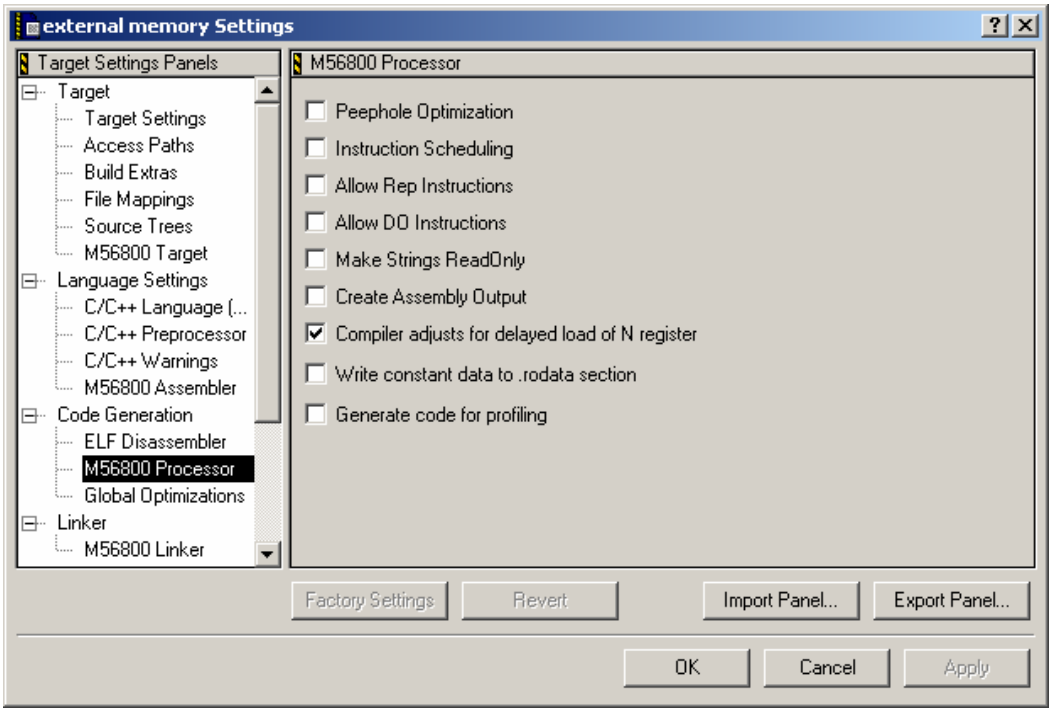


Figure 17-Ventana de configuración del procesador para hacer uso de recursos específicos.

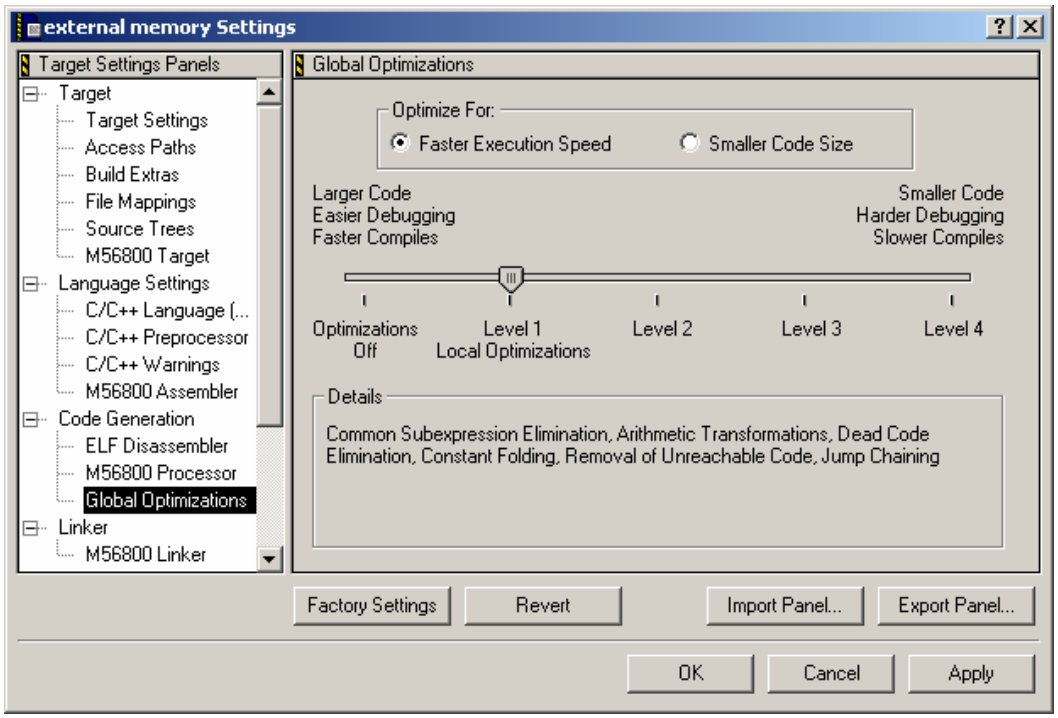


Figure 18-Ventana de configuración de las optimizaciones globales

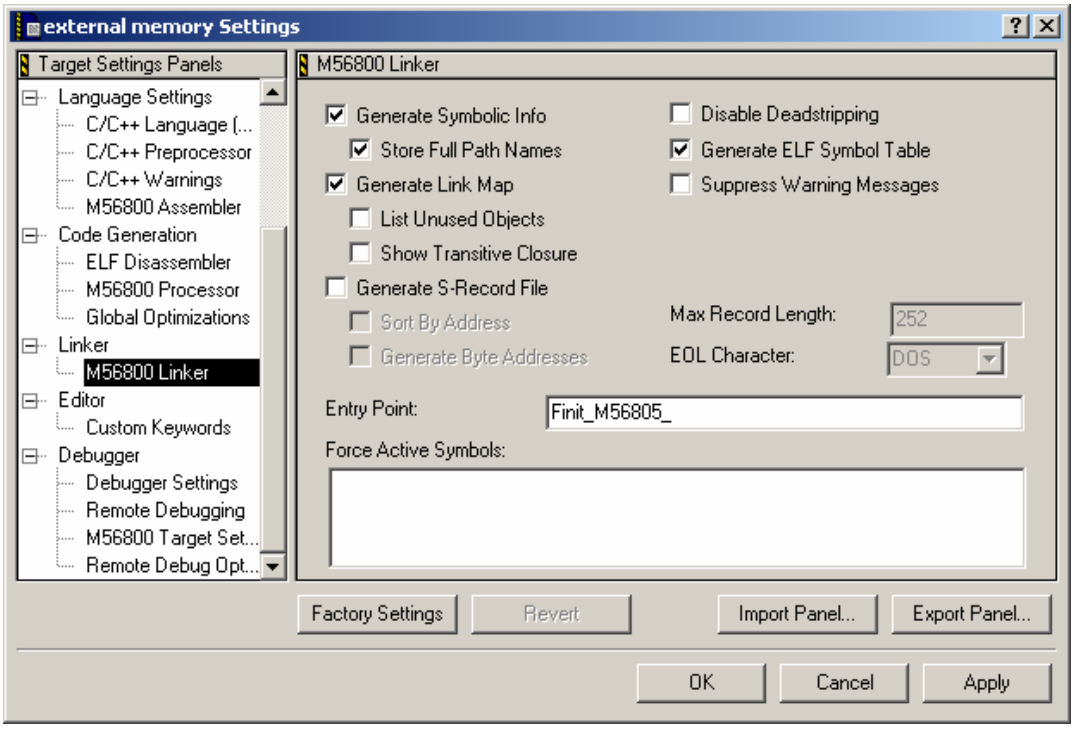


Figure 19-Ventana de configuración del enlazador acá tenemos que activar si queremos generar el archivo S19.

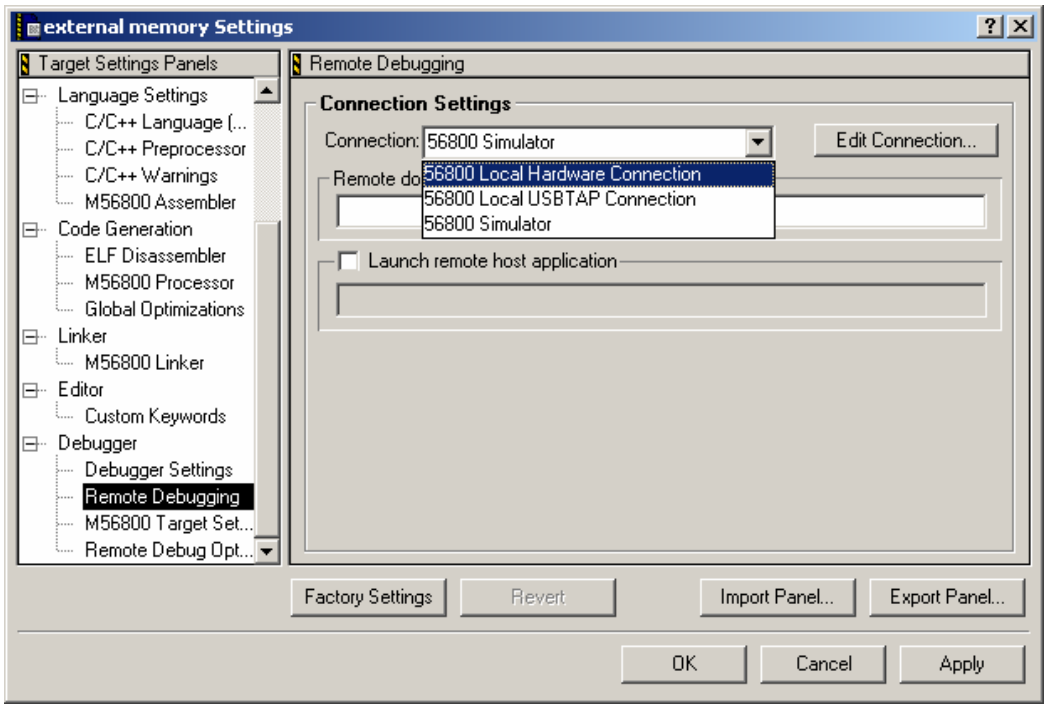


Figure 20- Ventana de configuración de la conexión del depurador o si usaremos el simulador.

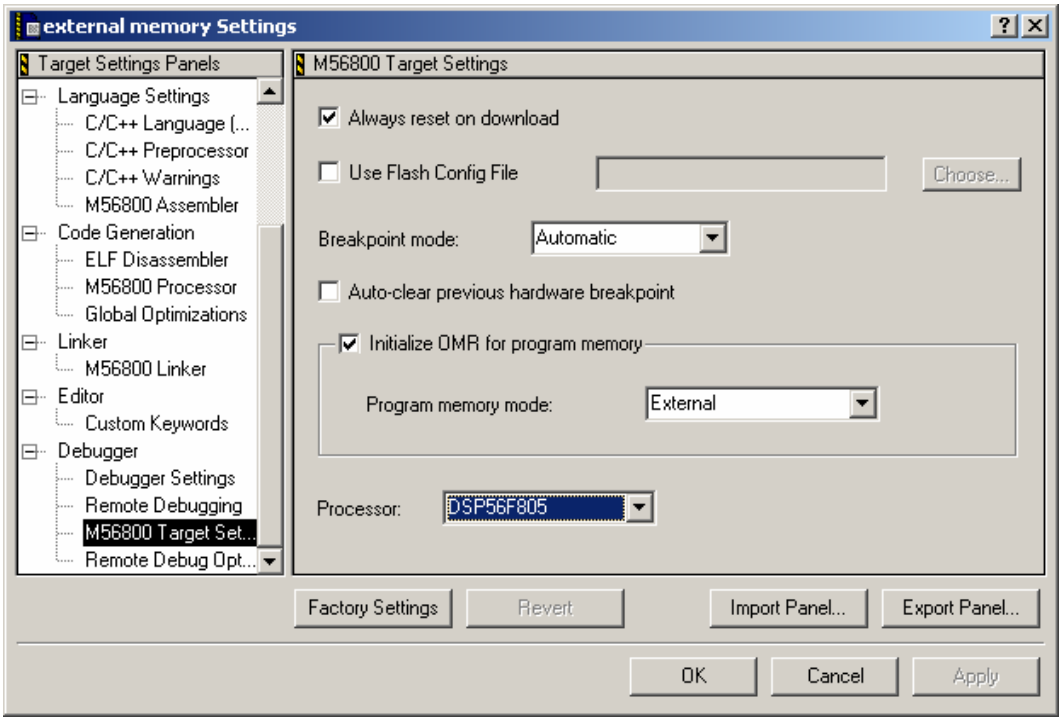


Figure 21 – Ventana de configuración del Target.

DEPURANDO UN PROYECTO

Para depurar una aplicación lo que tenemos que hacer es apretar el  icono de depuración en la ventana de proyecto o a través del menú principal ***PROJECT->DEBUG*** y se dispara la compilación y enlace de todo el proyecto y si no hay errores entonces se abre la ventana de depuración como la que se muestra en Figura 22, en la cual puede observarse la cantidad de recursos con los cuales se van a interactuar en la tarea de depuración del día a día.

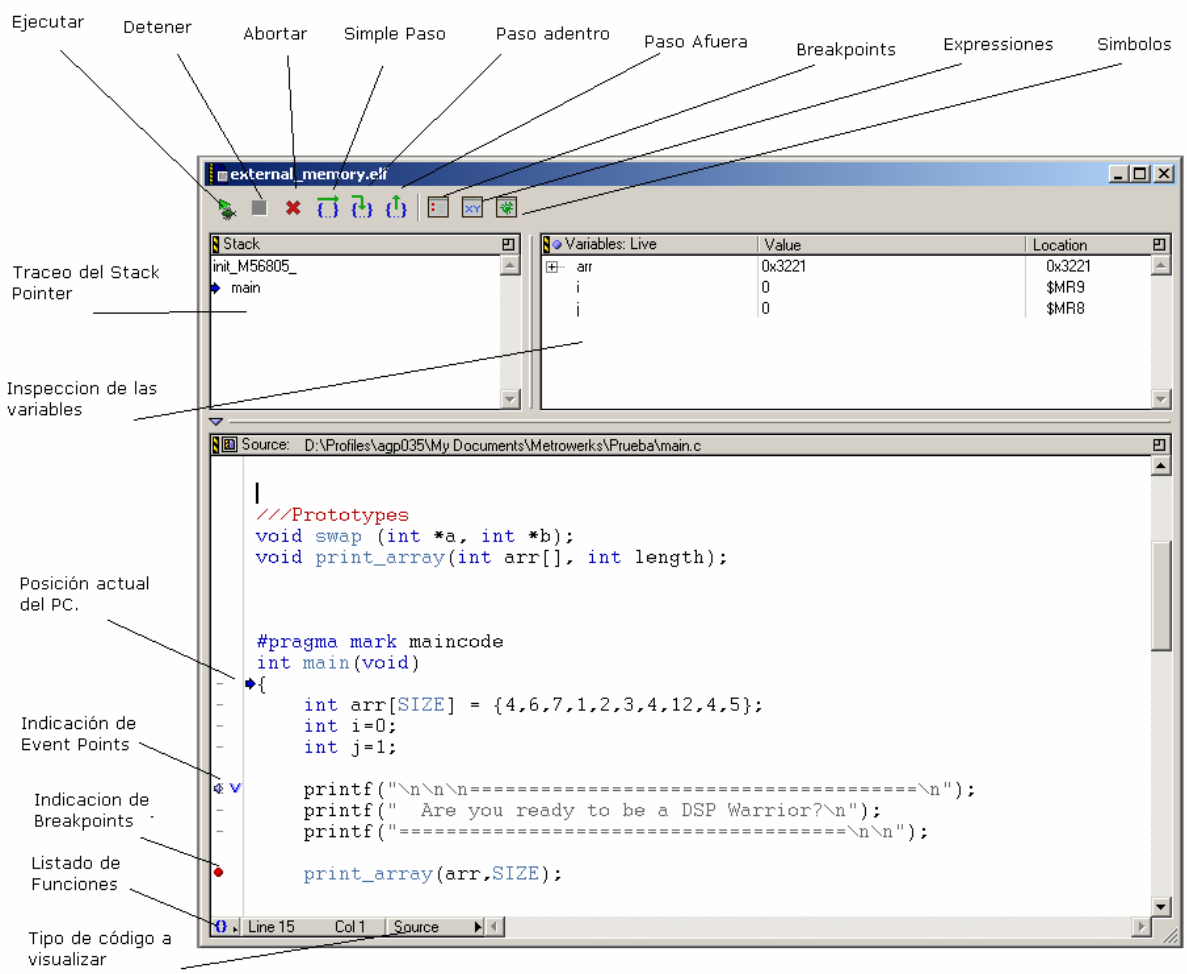


Figure 22 – Ventana de Depuración

Para ver los registros del procesador se selecciona la función en el menú del programa VIEW->REGISTERS, lo que permite visualizar la ventana cuyo aspecto se muestran en la Figuras 23 y 24.

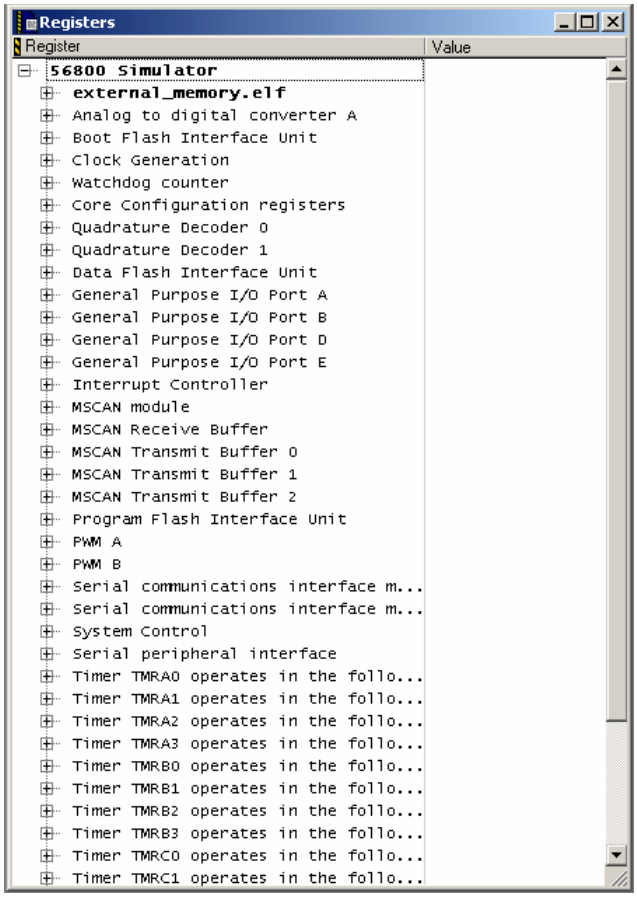


Figure 23-Ventana de Registros Generales

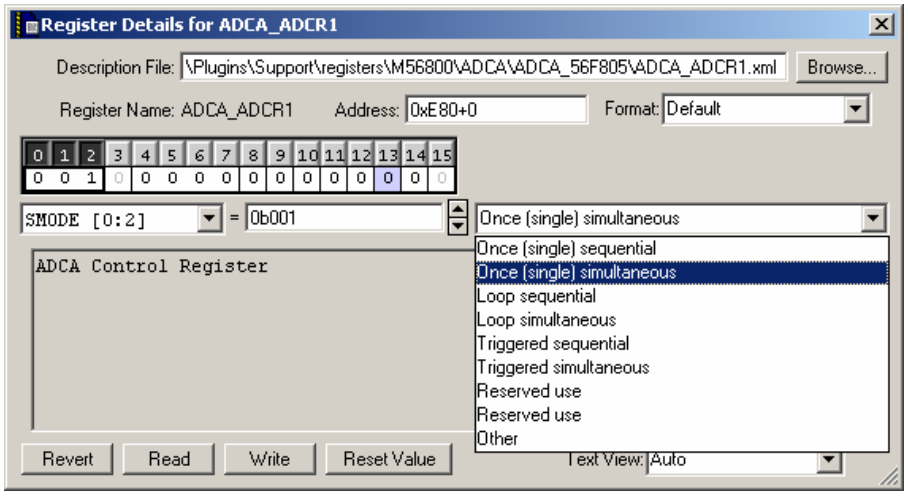


Figure 24-Ventana de Detalle de Registro del Conversor tiene definidos todos sus bits y sus combinaciones facilitando la depuración y consulta de valores al momento de la configuración de un periférico.

ARCHIVO DE COMANDOS DE ENLACE

Los archivos CMD son los archivos de que le define al enlazador como reservar memoria para todas las piezas de código y dato que componen el proyecto.

Los archivos CMD tienen su propio lenguaje completo, con palabras claves, directivas, y expresiones que son usadas para crear las especificaciones para su código de salida. La sintaxis y estructura del archivo de enlace es similar que otro lenguaje de programación.

Hay tres secciones importantes en un archivo CMD:

1. Segmentos de Memoria – (Memory Segments)
2. Segmentos de Sección – (Segment Sections)
3. Bloques de Cierre – (Closure Blocks)

Un archivo CMD debe contener un segmento de memoria y segmento de secciones, los bloques de cierre son opcionales. Las Funciones, palabras claves, directivas y comandos de los archivos de enlace se listan a continuación:

- . (location counter)
- ADDR
- ALIGN
- ALIGNALL
- FORCE_ACTIVE
- INCLUDE
- KEEP_SECTION
- MEMORY
- OBJECT
- REF_INCLUDE
- SECTIONS
- SIZEOF
- SIZEOFW
- WRITEB
- WRITEH
- WRITES
- WRITEW

Para mayor detalle referirse al manual de Targeting correspondiente.

A continuación se muestra un ejemplo de código correspondiente a un archivo CMD con Memoria Externa:

```
# ----------------------------------------------------

# Metrowerks sample code

# linker command file for DSP56805

# using
#     external pRAM
#     external xRAM
#     internal xRAM (0x30-40 for compiler regs)
#         mode 3
#          EXT 0

# ----------------------------------------------------

# see end of file for additional notes
# additional reference: Motorola docs

# for this LCF:
# interrupt vectors --> external pRAM starting at zero
#      program code --> external pRAM
#         constants --> external xRAM
#      dynamic data --> external xRAM

# stack size is set to 0x1000 for external RAM LCF


# requirements: Mode 3 and EX=0
# note -- there is a mode OB but any Reset or COP Reset
#         resets the memory map back to Mode 0A.


# DSP56805EVM board settings:
#    OFF --> jumper JG7 (mode 0 upon exit from reset)
#    ON  --> jumper JG8 (enable external board SRAM)


# CodeWarrior debugger Target option settings
#   OFF --> "Use Hardware Breakpoints"
#    ON --> "Debugger sets OMR at Launch" option

# note: with above option on, CW debugger sets OMR as
# OMR:
#     0 --> EX bit (stay in Debug processing state)
#     1 --> MA bit
#     1 --> MB bit



# DSP56805
# mode 3 (development)
```

```
# EX = 0

MEMORY
{
    .p_interrupts_ext_RAM (RWX) : ORIGIN = 0x0000, LENGTH = 0x0080
    .p_external_RAM       (RWX) : ORIGIN = 0x0080, LENGTH = 0x0000
    .x_compiler_regs_iRAM (RW)  : ORIGIN = 0x0030, LENGTH = 0x0010
    .x_internal_RAM       (RW)  : ORIGIN = 0x0040, LENGTH = 0x07C0
#   .x_reserved                 : ORIGIN = 0x0800, LENGTH = 0x0400
    .x_peripherals        (RW)  : ORIGIN = 0x0C00, LENGTH = 0x0400
    .x_flash_ROM          (R)   : ORIGIN = 0x1000, LENGTH = 0x1000
    .x_external_RAM       (RW)  : ORIGIN = 0x2000, LENGTH = 0xDF80
    .x_core_regs          (RW)  : ORIGIN = 0xFF80, LENGTH = 0x0000
}



# we ensure the interrupt vector section is not deadstripped here

KEEP_SECTION{ interrupt_vectors.text }



# place all executing code & data in external memory

SECTIONS {
        .interrupt_vectors_for_p_ram :
        {
            # from 56805_vector.asm
            * (interrupt_vectors.text)

        } > .p_interrupts_ext_RAM


        .executing_code :
        {
                # .text sections

                * (.text)
                * (rtlib.text)
                * (fp_engine.text)
                * (startup.text)
                * (user.text)
        } > .p_external_RAM


        .data :
        {
                # .data sections

                * (.const.data)
                * (fp_state.data)
                * (rtlib.data)
                * (.data)
```

```
        # .bss sections

        * (rtlib.bss.lo)

        __bss_start = .;

        * (.bss)

        __bss_end   = .;
        __bss_size = __bss_end - __bss_start;



        # setup the heap address

        __heap_addr = .;
        __heap_size = 0x1000; # larger heap for hostIO
        __heap_end = __heap_addr + __heap_size;

        . = __heap_end;



        # setup the stack address

        _min_stack_size = 0x0200;
        __stack_addr = __heap_end;
        __stack_end  = __stack_addr + _min_stack_size;
        . = __stack_end;



        # set global vars

        # MSL uses these globals:
        F_heap_addr  = __heap_addr;
        F_heap_end   = __heap_end;
        F_stack_addr = __stack_addr;


        # stationery init code globals

        F_bss_size      = __bss_size;
        F_bss_addr      = __bss_start;

        # next not used in this LCF
        # we define anyway so init code will link
        # these can be removed with removal of rom-to-ram
        # copy code in init file

        F_data_size     = 0x0000;
        F_data_RAM_addr = 0x0000;
        F_data_ROM_addr = 0x0000;

    F_rom_to_ram    = 0x0000; # zero is no rom-to-ram copy

    } > .x_external_RAM
}
```

```
# ------------------------------------------------------
# additional notes:


# about the reserved sections
# for this external RAM only LCF:

# p_interrupts_RAM -- reserved in external pRAM
# memory space reserved for interrupt vectors
# interrupt vectors must start at address zero
# interrupt vector space size is 0x80

# x_compiler_regs_iRAM -- reserved in internal xRAM
# The compiler uses page 0 address locations 0x30-0x40
# as register variables. See the Target manual for more info.



# notes:
# program memory (p memory)
# (RWX) read/write/execute for pRAM
# (RX) read/execute for flashed pROM

# data memory (X memory)
# (RW) read/write for xRAM
# (R)  read for data flashed xROM

# LENGTH = next start address - previous
# LENGTH = 0x0000 means use all remaining memory



# revision history
# 011020 R4.1 a.h. first version
# 030220 R5.1 a.h. improved comments
```

LENGUAJE C DE CW

A continuación damos una breve descripción del lenguaje C para la familia DSP568XX

- ✓ El lenguaje C++ no es soportado en la familia DSP 56800
- ✓ Las funciones matemáticas en punto flotante no están soportadas (ej: Sin, Cos, Sqrt).
- ✓ La función ***sizeof*** del lenguaje C no es lo mismo que la función ***SIZEOF*** del enlazador. En C, la función ***sizeof*** retorna el número de tipos ***SIZE_T***, el cual el compilador declara ser el tipo ***unsigned long int***. La función ***sizeof*** en C retorna el número de palabras, mientras que la función ***SIZEOF*** del enlazador retorna el número de bytes.

Formatos de Números

Se explicará como el compilador CW implementa los tipos de numerous enteros y de punto flotante para los procesadores DSP56800 (ver Figuras 25 a 27). Para más información sobre los tipos integer mirar el archivo limitis.h y para los tipos float mirar el archivo float.h que están explicados en el manual MSL C Reference Manual.

Type	Option Setting	Size (bits)	Range
bool	n/a	16	true or false
char	**Use Unsigned Chars** is disabled in the **C/C++ Language (C Only)** settings panel	16	-32,768 to 32,767
	Use Unsigned Chars is enabled	16	0 to 65,535
signed char	n/a	16	-32,768 to 32,767
unsigned char	n/a	16	0 to 65,535
short	n/a	16	-32,768 to 32,767
unsigned short	n/a	16	0 to 65,535
int	n/a	16	-32,768 to 32,767
unsigned int	n/a	16	0 to 65,535
long	n/a	32	-2,147,483,648 to 2,147,483,647
unsigned long	n/a	32	0 to 4,294,967,295

Figure 25- Tipos de Datos Enteros.

Type	Size (bits)	Range
float	32	1.17549e-38 to 3.40282e+38
short double	32	1.17549e-38 to 3.40282e+38
double	32	1.17549e-38 to 3.40282e+38
long double	32	1.17549e-38 to 3.40282e+38

Figure 26- Tipos de datos de punto flotantes

Type	Declared As	Size (bits)	Range
fixed	__fixed__	16	(-1.0 <= x < 1.0)
short fixed	__shortfixed__	16	(-1.0 <= x < 1.0)
long fixed	__longfixed__	32	(-1.0 <= x < 1.0)

Figure 27- Tipos de datos de punto fijo.

Convenciones de Llamadas y Estructura de la Pila

El IDE de CW para los DSP56800 almacena los datos y las llamadas a funciones de maneras que pueden ser diferentes a otras plataformas de destino.

Convenciones de Llamadas

Los registros A, R2, R3, Y0 e Y1 pasan los parámetros a las funciones. Cuando una función es llamada, la lista de parámetros es barrida de izquierda a derecha. Los parámetros son pasados en la siguiente forma:

1. El primer valor de 32-bits es colocado en A
2. Los primeros dos valores de 16-bits son colocados en Y0 e Y1 respectivamente
3. Las primeras dos direcciones de 16-bits son colocadas en R2 y R3 respectivamente.
4. Todos los parámetros restantes son enviados a la pila, comenzando de parámetro más hacia la derecha. Parámetros de múltiples-palabras tienen la palabra menos significativas enviada primero a la pila.
5. Cuando se llama a una rutina que retorna una estructura, el llamador pasa una dirección en R0 el cual especificará donde copiar la estructura.

Los registros A, R0, R2 e Y0 son usados para retornar los resultados de las funciones como sigue:

1. Valores de 32-bits son retornados en A.
2. Valores de direcciones de 16-bits son retornados en R2.
3. Todos los valores de 16-bits que no sean direcciones son retornados en Y0.

Registros No-Volátiles y Registros Volátiles

Los registros no-volátiles son registros que pueden ser usados para almacenar valores a través de las llamadas a funciones. Estos registros son también llamados registros de guarda a través de llamadas de funciones (Save Over Call registers **SOCs**).

Los registros volátiles son registros que no pueden almacenar valores a través de las llamadas a funciones. Estos registros son también llamados **non-SOCs** .

En la Figura 28 se muestra un listado de los registros volátiles y no-volátiles

Unit	Register Name	Size	Type	Comments
Arithmetic Logic Unit (ALU)	Y1	16	Volatile (non-SOC)	
	Y0	16	Volatile (non-SOC)	
	Y	32	Volatile (non-SOC)	
	X0	16	Volatile (non-SOC)	
	A2	4	Volatile (non-SOC)	
	A1	16	Volatile (non-SOC)	
	A0	16	Volatile (non-SOC)	
	A10	32	Volatile (non-SOC)	
	A	36	Volatile (non-SOC)	
	B2	4	Volatile (non-SOC)	
	B1	16	Volatile (non-SOC)	
	B0	16	Volatile (non-SOC)	
	B10	32	Volatile (non-SOC)	
	B	36	Volatile (non-SOC)	
Address Generation Unit (AGU)	R0	16	Volatile (non-SOC)	
	R1	16	Volatile (non-SOC)	
	R2	16	Volatile (non-SOC)	
	R3	16	Volatile (non-SOC)	
	N	16	Volatile (non-SOC)	
	SP	16	Volatile (non-SOC)	

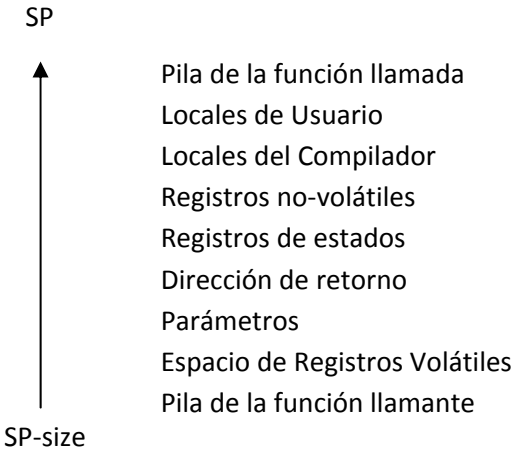
Unit	Register Name	Size	Type	Comments
Address Generation Unit (AGU) *(continued)*	M01	16	Volatile (non-SOC)	In certain registers, values must be kept for proper C execution. Set to 0xFFFF for proper execution of C code.
Program Controller	PC	21	Volatile (non-SOC)	
	LA	16	Volatile (non-SOC)	
	HWS	16	Volatile (non-SOC)	
	OMR	16	Volatile (non-SOC)	In certain registers, values must be kept for proper C execution. For example, set the CM bit. (See OMR Settings.)
	SR	16	Volatile (non-SOC)	
	LC	16	Volatile (non-SOC)	
Page 0	MR0	16	Volatile (non-SOC)	
	MR1	16	Volatile (non-SOC)	
	MR2	16	Volatile (non-SOC)	
	MR3	16	Volatile (non-SOC)	
	MR4	16	Volatile (non-SOC)	
	MR5	16	Volatile (non-SOC)	
	MR6	16	Volatile (non-SOC)	
	MR7	16	Volatile (non-SOC)	
	MR8	16	Non-volatile (non-SOC)	
	MR9	16	Non-volatile (non-SOC)	

Unit	Register Name	Size	Type	Comments
	MR10	16	Non-volatile (non-SOC)	
	MR11	16	Non-volatile (non-SOC)	
	MR12	16	Non-volatile (non-SOC)	
	MR13	16	Non-volatile (non-SOC)	
	MR14	16	Non-volatile (non-SOC)	
	MR15	16	Non-volatile (non-SOC)	

Figure 28- Listado de la clasificación de los registros volátiles y no-volátiles.

Estructura de la Pila

La estructura de la pila es generada como se muestra a continuación. La pila crece en orden ascendente, significando esto que poner un dato en la pila, incrementa la dirección del apuntador de pila:



El registro puntero de pila (SP) es un registro de 16-bits usado implícitamente en todas las instrucciones de PUSH y POP. La pila de software soporta estructuras programadas, tales como el pasaje de parámetros a subrutinas y variable locales. Si está programando con ambos lenguajes Assembly y lenguaje C de alto nivel use las técnicas de pila para mantener compatibilidad y entendimiento entre los lenguajes. Vea que es posible soportar el pasaje de parámetros y variables locales a una subrutina al mismo tiempo con el uso de la estructura de la pila.

Ubicación de la Pila de Usuario

El compilador DSP56800 construye estructuras para llamadas de funciones en forma jerárquicas usando el apuntador de pila (SP) para localizar la próxima locación de memoria disponible en la cual vamos a ubicar la estructura de información de las funciones llamadas. No hay usualmente registros apuntadores para estructuras explicitas. Normalmente el tamaño de las estructuras son fijados a tiempo de compilación. La cantidad total del espacio de pila requerida para los argumentos de vienen, las variables locales, el retorno de información de la función, las posiciones para salvar registros (incluyendo aquellos en los pragmas de las funciones de interrupción) es calculado y la estructura de la pila es ubicada al comienzo de la función llamada.

Algunas veces se puede necesitar modificar el SP en ejecución de programa para ubicar espacio de almacenamiento local temporario usando llamadas de assembler en línea. Esto invalida todos los desplazamientos de la estructura de la pila para ser usado con el SP para acceder a las variables locales, los argumentos en la pila, etc. Con la funcionalidad de ubicación de la pila de usuario, se puede usar instrucciones de assembler en línea (con ciertas restricciones) para modificar el SP, mientras mantenemos la precisión de las variables locales, datos temporarios del compilador y desplazamientos

de los argumentos, ej: esas variables pueden aun ser accedidas puesto que el compilador conoce que van a modificar el apuntador de la pila.

La funcionalidad de ubicación de pila de usuario es habilitada con la configuración del siguiente pragma:

```
#pragma check_inline_sp_effects [on|off|reset]
```

El pragma puede ser usado en funciones individuales. Por defecto el pragma está deshabilitado al comienzo de la compilación de cada archivo en el proyecto.

La funcionalidad de ubicación de pila de usuario permite simplemente agregar líneas de assembler en línea para modificar el SP en cualquier lugar de la función. Estas restricciones son de carácter directo:

1. El SP debe ser modificado por la misma cantidad en todos los caminos de ida al punto de control de flujo de mezcla.
2. El SP debe ser modificado por una cantidad constante. Esto es un modo de direccionamiento tal como “(SP)+N” y escrituras directas sobre el SP no son manejadas.
3. El alineamiento del SP debe ser mantenido apropiadamente.
4. No debe sobre-escribir la ubicación de la pila del compilador por el decremento del SP en el espacio de la pila ubicada por el compilador.

El punto uno es requerido cuando se piensa con instrucciones del tipo “if-then-else”. Si una rama del punto de decisión modifica el SP en un solo lado y las otras ramas lo modifican de otra manera, luego el valor de SP depende del tiempo de ejecución y el compilador no es capaz de determinar donde las variables básicas están ubicadas en tiempo de ejecución. Para prevenir esto a que suceda la funcionalidad de ubicación de pila de usuario cruza el grafico de control de flujo, grabando las modificaciones del SP por las instrucciones de assembler en línea a través de todos los caminos del programa. Esto luego controla todos los puntos de mezcla para asegurarse que las modificaciones que se hicieron sobre el SP son consistentes en cada rama, convergiendo al punto de mezcla. Si esto no sucede una advertencia es emitida anunciando la inconsistencia entre las ramas.

Una vez que el compilador determina que las modificaciones en línea del SP son consistentes en el grafo de control de flujo, los desplazamientos del SP usados para referenciar las variables locales, argumentos de funciones o valores temporales son reparados al conocer las modificaciones en línea del SP. Advierta que puede libremente ubicar espacio local para almacenamiento en la pila:

1. Se puede modificar tanto como se quiera el SP a lo largo de todas las ramas al punto de control de mezcla de flujo.
2. El SP está alineado correctamente. El SP debe ser modificado por una cantidad que el compilador pueda determinar en tiempo de compilación.

Un nuevo pragma es definido:

```
#pragma check_inline_sp_effects [on|off|reset]
```

generará una advertencia si el usuario introduce assembler en línea que modifican el SP una cantidad dependiente del tiempo de ejecución. Si el pragma no es especificado, entonces los desplazamientos de la pila usados para acceder a las variables bases de la pila serán incorrectos. Es responsabilidad el usuario habilitar `#pragma check_inline_sp_effects`, si se desea modificar el SP con assembler en línea y acceder a variables locales en la pila. Tenga en cuenta que este pragma no tiene efecto en funciones a nivel de assembler o archivos de assembler puro (.ASM).

En general, assembler en línea puede ser usado para crear grafos de flujos arbitrarios y no todos ellos pueden ser detectados por el compilador.

Por ejemplo:

```
REP #3
LEA (SP)+
```

Este ejemplo modificaría el SP en un valor de 3, pero el compilador vería solamente la modificación de un solo valor. Otros casos como estos pueden ser creados por el usuario usando saltos o derivaciones con assembler en línea. Estas son construcciones peligrosas y no son detectadas por el compilador. En el caso de que el SP es modificado por una cantidad dependiente del tiempo de ejecución, una advertencia es emitida.

En la Figura 29 se muestra el código de modificación permita por el SP usando assembler en línea.

```
#define EnterCritical() { asm(lea (SP)+);\
                          asm(move SR,X:(SP)+); \
                          asm(bfset  #0x0300,SR); \
                          asm(nop); \
                          asm(nop);}

#define ExitCritical()  { asm(lea (SP)-;\
                          asm(move X:SP) ,SR); \
                        asm(nop);\
                          asm(nop);}

#pragma check_inline_sp_effects on
```

```
int func()
{
        int a=1, b=1, c;

        EnterCritical();

        c = a+b;

        ExitCritical();

}
```

Figure 29 - Modificación permitida del SP usando assembler en línea.

En este caso funcionará porque no hay puntos de control de flujo de mezcla. El SP es modificado consistentemente a lo largo de todos las ramas de comienzo a final de las funciones y es propiamente alineado.

```
#define EnterCritical() { asm(lea (SP)+);\
                          asm(move SR,X:(SP)+); \
                          asm(bfset  #0x0300,SR); \
                          asm(nop); \
                          asm(nop);}

#define ExitCritical()  { asm(lea (SP)-;\
                          asm(move X:SP) ,SR); \
                        asm(nop);\
                          asm(nop);}

#pragma check_inline_sp_effects on

int func()
{
          int a=1, b=1, c;

          if (a)
{
                  EnterCritical();

                  c = a+b;

          }
          else {
                  c = b++;
          }

          ExitCritical();

          return (b+c);
}
```

Figure 30 – Modificación NO permitida del SP usando assembler en línea.

En el ejemplo de la Figura 30, se generará el siguiente mensaje de advertencia porque la entrada del SP al macro “ExitCritical” es diferente dependiendo de la rama que tome el if. Por lo tanto el acceso a las variables a, b y c pueden no ser las correctas:

```
Warning : Inconsistent inline assembly modification of SP in this
function.
M56800_main.c line 29 ExitCritical();
```

```
#define EnterCritical() { asm(move n,SP);\
                          asm(move SR,X:(SP)+); \
                          asm(nop); \
                          asm(nop);}

#define ExitCritical()  { asm(lea (SP)-;\
                          asm(move X:(SP) ,SR); \
                          asm(nop);\
                          asm(nop);}

#pragma check_inline_sp_effects on
int func()
{
          int a=1, b=1, c;

          if (a)
          {
                  EnterCritical();

                  c = a+b;

          }
          else {
                  EnterCritical();
                  c = b++;
          }



          return (b+c);
}
```

Figure 31 – Modificación del SP por una cantidad dependiente del tiempo de ejecución.

El ejemplo de la Figura 31, generará el siguiente mensaje de advertencia:

```
Warning : Cannot determine SP modification value at compile time
M56800_main.c line 20 EnterCritical();
```

Este ejemplo es inválido puesto que el SP es modificado por una cantidad dependiente en tiempo de ejecución.

Si todas las modificaciones de assembler en línea al SP a lo largo de todas las ramas son igualmente implementadas a la salida de la función, no es necesario explícitamente desalojar el espacio incrementado de la pila. El compilador limpiará las líneas extras de assembler en la pila automáticamente al final de la función.

El ejemplo de la Figura 32, no necesita llamar al macro “ExitCritical” porque el compilador automáticamente limpiará las líneas extras de assembler en la pila.

```
#pragma check_inline_sp_effects on
int func()
{
            int a=1, b=1, c;

            if (a)
            {
                        EnterCritical();

                        c = a+b;

            }
            else {
                        EnterCritical();
                        c = b++;
            }



            return (b+c);
}
```

Figure 32 - Borrado automático del assembler en línea en la pila

Secciones Generadas por el Compilador

El compilador crea ciertas secciones por defecto cuando compila archivos fuentes de C. Estas secciones por defecto son todas manejadas por los comandos de default en el archivo CMD y son las siguientes:

- .text (el compilador coloca el código ejecutable acá por defecto)
- .data (el compilador coloca los datos inicializados acá por defecto)
- .bss (el compilador coloca los datos no inicializados acá por defecto)

Nota: Estas secciones son secciones generadas por el compilador por defecto. Otras secciones definidas por el usuario pueden ser generadas a través del uso de:

```
#pragma define_section
```

Si el proyecto tiene la opción habilitada “write constant data to .rodata” en M56800 Processor de los Target Settings, entonces el compilador generará la sección .rodata para datos constantes. Esta opción es sobre-escrita por el uso de the #pragma use_rodata.

Nota: La sección .rodata no es manejado por default por el archivo CMD. Por lo tanto si ud. necesita el uso de esta sección debe colocarla dentro del archivo CMD dentro de una de las secciones del mapa de memoria (ver Figura 33).

Por defecto los datos inicializados a cero son puestos en la sección .bss por el compilador. Esto es hecho para reducir el tamaño de carga de la aplicación. El tamaño de carga es reducido porque en vez cargar el depurador una secuencia de ceros en la sección .data (la sección cargable), el compilador simplemente mueve los datos inicializados a cero a la sección .bss (sección no cargable), la cual es inicializada a cero por el código de arranque. Este comportamiento puede ser sobre-escrito por el uso del #pragma explicit_zero_data o por el uso de #pragma use_rodata, el cual pone todos los datos constantes en una sección especial .rodata.

Section	Size	Range (Hexadecimal)
PROGRAM	64K x 16 bit	0000 - FFFF
DATA	64K x 16 bit	0000 - FFFF

Figure 33 – Mapa de Memoria

Configuración OMR

El registro de operación de modo (OMR) es parte del controlador de programa del núcleo del DSP56800. Este registro es el responsable de la mayoría de los modos en como el núcleo opera.

Nota: Para los detalles generales del OMP ver el DSP56800 Family Manual. Para los detalles específicos del registro ver el manual del chip por lo general el DSP56800 User Manual. El compilador CW tiene ciertos requerimientos sobre el valor contenido dentro del registro y el modo en que el núcleo del DSP56800 opera. Estos requerimientos están expresados en la Figura 34.

Bit Number	Bit Name	Requirements
4	Saturation or SA bit	This bit must be cleared for the compiled code to work properly.
5	Rounding or R bit	This bit must be cleared for the compiled code to work properly.
8	Condition code or CC bit	This bit must be set for the compiled code to work properly.

Figure 34 – Bits de requerimientos sobre el OMR

BIBLIOGRAFIA

DSP56800 Family Manual. 16-Bit Digital Signal Controllers. DSP56800FM Rev. 3.1 11/2005. Freescale.

DSP56800 User Manual. 16-Bit Hybrid Controllers. DSP56F801-7UM Rev. 6 07/2004. Freescale.

CodeWarrior™ Development Studio for Freescale™ 56800/E Hybrid Controllers: MC56F83xx/DSP5685x Family Targeting Manual. Freescale.

MSL C Reference 10.0. Freescale.