Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Diseño e implementación de un marco de trabajo para el desarrollo de aplicaciones embebidas en el DSP del Sistema en Chip DM8168

Informe de Proyecto de Graduación para optar por el título de

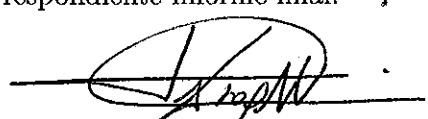
Ingeniero en Electrónica con el grado académico de Licenciatura

José Alberto López Mata

Cartago, Mayo, 2013

Declaro que el presente Proyecto de Graduación ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.



José Alberto López Mata

Cartago, 27 de junio de 2013

Céd: 1-1368-0657

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Proyecto de Graduación
Tribunal Evaluador

Proyecto de Graduación defendido ante el presente Tribunal Evaluador como requisito para optar por el título de Ingeniería en Electrónica con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal

Ing. Gabriela Ortiz León, MSc
Profesora Lectora

Ing. Arys Carrasquilla Batista, MSc
Profesora Lectora

Ing. Jorge Castro Godínez
Profesor Asesor

Los miembros de este Tribunal dan fe de que el presente trabajo de graduación ha sido aprobado y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica.

Cartago, 27 de junio de 2013

Resumen

Este documento presenta el diseño e implementación de un marco de trabajo para el desarrollo de aplicaciones embebidas en el Sistema en Chip DM8168. Este sistema perteneciente a *Texas Instruments* (TI) contiene un procesador digital de señales que posibilita cómputo dedicado como el de algoritmos de procesamiento digital de imágenes (PDI). El desarrollo e implementación de aplicaciones multimedia en el procesador de señales está bajo la mira del proyecto.

El marco de trabajo sirve como base de desarrollo para un programa de prueba que utiliza una técnica de procesamiento digital de imágenes. Este toma una imagen a la cual le aplica un operador Sobel. Este funciona como una máscara que opera sobre el contenido de la imagen para revelar sus bordes. El programa de prueba es ejecutado en una plataforma de desarrollo donde el Sistema en Chip se encuentra empotrado, esta plataforma ofrece los medios para desarrollar y probar aplicaciones embebidas.

Palabras clave: marco de trabajo, procesador digital de señales, Sistema en Chip, Sobel

Abstract

This document describes the design and implementation of a framework for embedded applications development for the DM8168 System-on-Chip. This system from Texas Instruments has a Digital Signal Processor for dedicated operations, such as digital image processing algorithms. The development and implementation of multimedia applications on the DSP is under the scope of the project.

The framework works as a base for the development of a test program that uses digital signal processing techniques. The application grabs an image and computes a Sobel operator over it, this operator works as a mask that computes data over the image content to expose its edges. The test program runs in a development platform where the System on Chip is embedded, this platform offers the means for the development and testing of embedded applications.

Keywords: framework, Digital Signal Processor, System-on-Chip, Sobel

a mi familia, especialmente mis padres; quienes me ofrecieron su apoyo constante e incondicional durante esta etapa de mi vida

Agradecimientos

En primer lugar agradezco a mi familia, por enseñarme el valor del esfuerzo que me permitió llevar a cabo el presente proyecto. Especialmente a mis padres Alicia y Roberto, quienes me enseñaron a siempre aspirar metas altas y por esforzarse día a día para ayudarme a completar esta etapa de mi vida. A mis amigos por ser fuente de motivación y apoyo en el transcurso de toda mi carrera.

A RidgeRun por darme el espacio para desarrollar mi proyecto, agradezco a su personal por la paciencia y ayuda técnica que brindaron en el transcurso del mismo. También, agradezco al Tecnológico de Costa Rica, por darme las oportunidades para realizar mi carrera y así crecer personal y profesionalmente.

José Alberto López Mata

Cartago, 27 de junio de 2013

Índice general

Índice de figuras iii

Índice de tablas v

1 Introducción 1
- 1.1 Sistemas empotrados en el mercado de tecnologías . . . 1
- 1.2 Implementación de soluciones en el DSP C6748 . . . 1
- 1.3 Marco de trabajo para la ejecución de procesos remotos . . . 2
- 1.4 Objetivos y estructura del documento . . . 3

2 Marco Teórico 5
- 2.1 Sistemas empotrados . . . 5
- 2.2 Sistema en chip DM8168 . . . 5
  - 2.2.1 Arquitectura . . . 5
  - 2.2.2 Esquema de operación cliente-servidor . . . 7
  - 2.2.3 Módulo de evaluación DM8168 . . . 7
- 2.3 Ambiente de desarrollo GNU/Linux . . . 8
  - 2.3.1 Shell . . . 8
  - 2.3.2 GNU Make . . . 9
- 2.4 SDK: Kit de Desarrollo de Software . . . 9
  - 2.4.1 Compilación cruzada . . . 10
  - 2.4.2 Sistema de integración de aplicaciones . . . 10
  - 2.4.3 Mapa de memoria . . . 11
- 2.5 Aplicaciones de procesamiento digital de señales . . . 12
  - 2.5.1 Operador Sobel como filtro de imagenes . . . 13
- 2.6 Interfaces de programación . . . 14
  - 2.6.1 Codec Engine . . . 14
  - 2.6.2 OSAL: Capa de abstracción de sistema operativo . . . 14
  - 2.6.3 CMEM: Controlador de memoria física . . . 15
  - 2.6.4 xDM: Interfaz de media digital . . . 15
- 2.7 RTSC: Componentes de Software de Tiempo Real . . . 16
  - 2.7.1 Servidor . . . 18
  - 2.7.2 Codec . . . 18
- 2.8 xdcTools . . . 19

2.8.1 xdc . . . 19
2.8.2 xs . . . 19

3 Marco de trabajo para la ejecución de procesos remotos 21
3.1 Integración con el SDK . . . 21
3.2 Esquema de operación del marco de trabajo . . . 22
3.2.1 Configuración de entorno de desarrollo . . . 22
3.2.2 Capa de construcción . . . 22
3.2.3 Generación de software . . . 23
3.3 Sistema de construcción . . . 24
3.4 Clase para la ejecución de procesos remotos . . . 25
3.4.1 Metaprogramas . . . 26
3.4.2 Comandos de construcción . . . 27
3.4.3 Flujo de operación . . . 28

4 Implementación de detector de bordes 33
4.1 Personalización de la interfaz de media digital . . . 33
4.1.1 Función *process* . . . 33
4.1.2 Expansión de argumentos de la interfaz IUNIVERSAL . . . 34
4.2 IMG_sobel_3x3 . . . 34
4.2.1 Formato de la imagen . . . 35
4.3 Aplicacion cliente . . . 36

5 Configuración del ambiente de ejecución 39
5.1 Partición de Linux . . . 39
5.2 Carga de módulos . . . 39

6 Resultados y Análisis 43
6.1 Factorización de secuencia de construcción . . . 43
6.2 Reducción del tiempo de construcción . . . 43
6.3 Detección de bordes en imagen de prueba . . . 45
6.4 Carga de procesamiento . . . 47
6.5 Mapa de memoria resultante . . . 47

7 Conclusiones 49
7.1 Recomendaciones . . . 51

Bibliografía 53

A Mapa de memoria del SDK para el DM8168 57

B Tuberias de GStreamer 59

Índice de figuras

1.1 Esquema del entorno de desarrollo del marco de trabajo 2

2.1 Arquitectura del sistema en chip DM8168 [25] 6
2.2 Diagrama de comunicación entre un cliente y un servidor 7
2.3 Módulo de evaluación DM8168 [19] 8
2.4 Sistema de comunicación por video que se implementa en el EVM DM8168 [10] 9
2.5 Sistema de integración de aplicaciones del SDK 11
2.6 Ejemplo de tuberia de GStreamer 11
2.7 Imagen original de una copa junto a su imagen filtrada con operador Sobel [6] 13
2.8 Estructura de *pools* y búferes de CMEM 15
2.9 Ciclo de vida de un algoritmo [42] 16
2.10 Contenido de un paquete RTSC 17
2.11 Diagrama de aplicaciones cliente - servidor en el SoC DM8168 19

3.1 Marco de trabajo junto al sistema de integración de aplicaciones del SDK . 22
3.2 Configuración del ambiente de desarrollo GNU/Linux 23
3.3 Construcción de software para un ambiente multiprocesador 23
3.4 Generación de contenido para arquitectura DM8168 24
3.5 Diagrama de flujo simplificado de operación del marco de trabajo 25
3.6 Estructura de metaprograma *config.bld* para construir paquetes RTSC . . . 27
3.7 Estructura de metaprograma ARM.cfg para configurar aplicación cliente . 27
3.8 Ejemplo de comando utilizado por la clase 28
3.9 Sequencia de comandos de la clase 29
3.10 Comandos para generar paquetes RTSC base con xdcTools 30
3.11 Diagrama de secuencia de construcción 31

4.1 Envoltura del operador Sobel en la función *process* de la interfaz xDM . . . 34
4.2 Expansión de estructura de IUNIVERSAL del codec Sobel 34
4.3 Diagrama de flujo de la operación del filtro Sobel 35
4.4 Diagrama de operación de aplicación cliente 37

5.1 Configuración de CMEM para alojar los búferes del DSP 40

5.2 Configuración del ambiente de ejecución para la aplicación del filtro de imágenes 41

6.1 Comparación de sistemas de construcción original y factorizado 44
6.2 Imagen de ejemplo con resolución de 640 x 480 convertida a escala de grises y procesada por el operador Sobel 46
6.3 Juego de imagenes filtradas por operador Sobel con diferentes niveles de umbral 46
6.4 Mapa de memoria para aplicación demo ARM + DSP 48

A.1 Mapa de memoria completo del SDK [16] 58

Índice de tablas

2.1 Bloques del mapa de memoria para la arquitectura DM8168 12

6.1 Reducción de tiempo de construcción de software 45
6.2 Carga de procesamiento de operador Sobel ejecutado en el procesador ARM y en los procesadores ARM y DSP 47

Nomenclatura

SoC: Sistema en Chip
DSP: Procesador Digital de Señales
PDI: Procesamiento Digital de Imágenes
SDK: Kit de desarrollo de software
RTSC: Componente de Software de Tiempo Real
XDAIS: Estándar de interoperabilidad de algoritmos
xDM: Interfaz de media digital
CE: Codec Engine
Kernel horizontal de operador Sobel

$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$$

Kernel vertical de operador Sobel

$$\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Capítulo 1

Introducción

1.1 Sistemas empotrados en el mercado de tecnologías

El proyecto se lleva a cabo en *RidgeRun – Embedded Solutions*, empresa que se especializa en software empotrado y ofrece soluciones hechas a la medida para sistemas basados en GNU/Linux, esto incluye herramientas de desarrollo de software, núcleos, controladores, aplicaciones y servicios de ingeniería [33].

La empresa participa en el mercado de productos y servicios multimedia en un ambiente muy competitivo, donde se tienen que generar soluciones utilizando tecnologías nuevas que mutan constantemente. Esto implica a su vez un reto para que los desarrolladores de software puedan aprovechar estas tecnologías e innovar soluciones. Un caso concreto es el del Sistema en Chip DaVinci Media 8168 de TI, este sistema está compuesto de múltiples unidades de procesamiento dedicado. Una de ellas es el Procesador Digital de Señales (*Digital Signal Processor*, DSP) C6748[25], donde un desarrollador que implementa software en esta unidad puede verse involucrado con distintos obstáculos técnicos; como técnicas de programación orientadas a componentes, tecnologías específicas de software del propietario, reglas de integración de algoritmos [21], técnicas de procesamiento digital de señales, entre otros.

La empresa, en miras a fortalecer sus soluciones embebidas busca conocer más a fondo esta arquitectura para aprovechar sus recursos tanto de hardware como de software y competir más en el mercado de aplicaciones multimedia.

1.2 Implementación de soluciones en el DSP C6748

La falta de las herramientas de software que faciliten la integración de aplicaciones embebidas en el Sistema en Chip (*System-on-Chip*, SoC) DM8168 es una necesidad presente.

El proyecto diseña e implementa un marco de trabajo que permita aprovechar los recursos ofrecidos por la arquitectura, específicamente el DSP C6748. De esa manera se abre un camino para incursionar en técnicas más sofisticadas para procesar datos multimedia, como lo son los algoritmos de PDI.

1.3 Marco de trabajo para la ejecución de procesos remotos

El aporte del proyecto corresponde a diseñar un marco de trabajo para el Kit de Desarrollo de Software de RidgeRun (*Software Development Kit*, SDK). El marco de trabajo se encarga de agrupar y enlazar los componentes del SDK necesarios para formar un sistema de construcción que genere contenido ejecutable para el SoC DM8168. Esto con la finalidad de facilitar el proceso de construcción de software que pueda ejecutarse en un ambiente multiprocesador y así formar una base para generar aplicaciones multimedia más complejas.

Para establecer una prueba base de las capacidades del marco de trabajo y del SoC se implementa una aplicación de detección de bordes en imágenes. Para ello se integra un operador Sobel en la arquitectura, este procesa cada sección de la imágen para calcular el gradiente de la misma y así exponer sus bordes.

La Figura 1.1 muestra un esbozo del entorno en el que el marco de trabajo se desarrolla, el cual está integrado con el SDK y utiliza el software y demás recursos que lo componen. El marco de trabajo se encarga de generar aplicaciones con contenido local y remoto. La primera es donde una aplicación cliente ejecutada en el procesador ARM administra un proceso remoto. La segunda es donde una aplicación servidor en el DSP ejecuta tareas de cómputo dedicado y entrega los datos procesados al cliente.

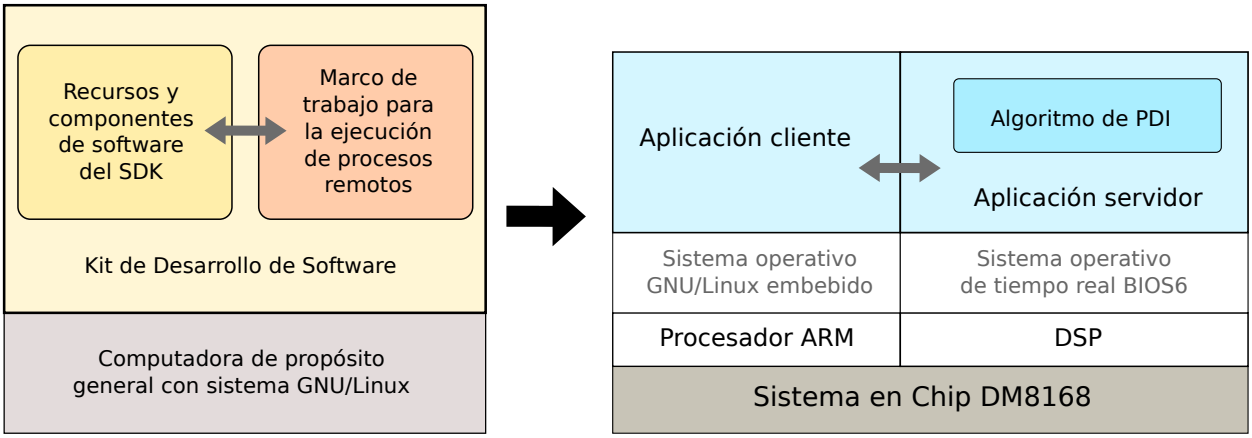


Figura 1.1: Esquema del entorno de desarrollo del marco de trabajo

1.4 Objetivos y estructura del documento

El proyecto tiene como objetivo integrar al SDK un marco de trabajo para facilitar el desarrollo de algoritmos que se ejecuten en el DSP de la arquitectura DM8168. Para conseguirlo se necesita que el marco de trabajo permita el funcionamiento en conjunto de componentes de software necesarios para ejecutar un proceso remoto.

El marco de trabajo sirve de base para el desarrollo de un programa de prueba que aplica un filtro Sobel sobre una imagen, el cual se ejecuta en el procesador digital de señales. Además para cumplir lo anterior hay que definir el ambiente de ejecución de la arquitectura para que la aplicación se ejecute en un ambiente multiprocesador.

El documento establece el marco teórico en el capítulo 2, así permite a los lectores familiarizarse con los conceptos técnicos necesarios para el desenvolvimiento de temáticas más específicas de los sistemas embebidos y el procesamiento digital de señales.

En el capítulo 3 se explican las características y el diseño del marco de trabajo. El capítulo 4 cubre la implentación del programa de prueba que lleva a cabo un filtro de imágenes en el DSP. La configuración del entorno de desarrollo en el SoC DM8168 es discutido en el 5. En el capítulo 6 se encuentran los resultados del proyecto junto a su análisis, por último se hacen las conclusiones y recomendaciones en el capítulo 7.

Capítulo 2

Marco Teórico

2.1 Sistemas empotrados

Un sistema empotrado (o embebido) es un sistema de computación que fue elaborado con la finalidad de cubrir necesidades específicas. Estos sistemas al ser fabricados con fines particulares, le permite a los ingenieros de diseño de hardware optimizarlos en reducción de tamaño y precio de producción; y a su vez aumentar su fiabilidad y desempeño. Los sistemas empotrados pueden formar parte de sistemas de cómputo de señales digitales, analógicas o mixtas; también pueden estar compuestos de sistemas de un solo procesador o multiprocesador [11, 38].

Estos dispositivos pueden emplearse en diferentes aplicaciones: en el área de multimedia como decodificadores de audio, en transportes como computadoras a bordo en trenes, en la industria como líneas de ensamblaje, entre muchas otras. Texas Instruments fabrica y da soporte a dispositivos afines a los sistemas empotrados como por ejemplo microprocesadores, procesadores digitales de señales y SoC.

2.2 Sistema en chip DM8168

2.2.1 Arquitectura

El SoC DM8168 pertenece a la serie DaVinci de TI, cuya característica principal es la eficiencia del procesamiento de aplicaciones multimedia. La Figura 2.1 muestra la arquitectura DM8168, donde resaltan diferentes unidades que se enuncian a continuación:

- Procesador ARM Cortex A8: Se encarga de control de procesos, cómputo de propósito general y administración de tareas.
- SGX: Unidad dedicada a procesamiento gráfico en 3D (tercera dimensión).
- Procesador de señales C6748: Toma la función de procesador auxiliar, es capaz de

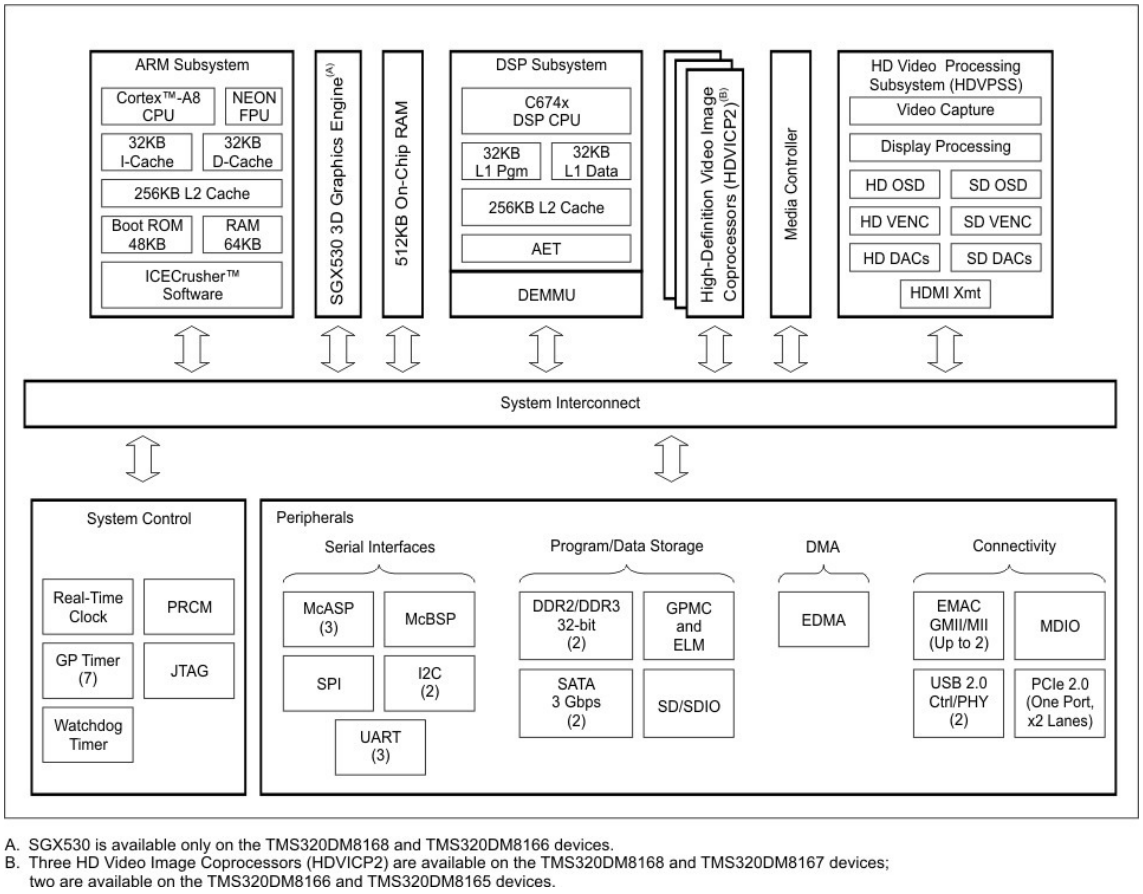
ejecutar eficientemente algoritmos dedicados de procesamiento digital de señales.

- Subsistema Ducati: Compuesto por diferentes unidades, como el controlador de media, coprocesadores y subsistema de video.

HDVICP2: Coprocesadores dedicados para imágenes y video en alta definición.

HDVPSS: Susbsistema de procesamiento de video para captura y despliegue de contenido.

Controlador de Media: Comprende dos procesadores ARM Cortex M3 que administran a los módulos HDVICP2 y HDVPSS [44].



A. SGX530 is available only on the TMS320DM8168 and TMS320DM8166 devices.
B. Three HD Video Image Coprocessors (HDVICP2) are available on the TMS320DM8168 and TMS320DM8167 devices; two are available on the TMS320DM8166 and TMS320DM8165 devices.

Figura 2.1: Arquitectura del sistema en chip DM8168 [25]

Todas estas unidades en conjunto amplian la gama de tareas que el SoC puede cubrir. Por ejemplo, para aplicaciones de codificación de video el DM8168 se comunica con el módulo HDVICP2 por medio del Controlador de Media. Para llevarlo a cabo se debe cargar un *firmware* a los procesadores M3, el *firmware* es en un código binario guardado en memoria que es interpretable únicamente por estos procesadores de media. Cuando la tarea de codificación es instanciada, el programa del Controlador de Media utiliza al módulo HDVICP2 para realizar el procesamiento del contenido de video.

2.2.2 Esquema de operación cliente-servidor

En sistemas multiprocesador como el DM8168, las unidades pueden tener diferentes esquemas de operación como maestro-esclavo o cliente-servidor [14]. La Figura 2.2 muestra el funcionamiento del procesador ARM como un administrador que ejecuta aplicaciones cliente y el cliente invoca un proceso remoto en otra unidad. La aplicacíon servidor se ejecuta en el procesador esclavo, captura los datos a la entrada, los procesa y devuelve al procesador ARM.

En el SoC DM8168, se puede presentar la ejecución de una tarea en el procesador ARM y parte de ella se descarga a alguna unidad auxiliar (procesadores M3, DSP, o SGX). De esta forma la carga del procesador ARM se aligera y se aprovecha el uso de unidades dedicadas.

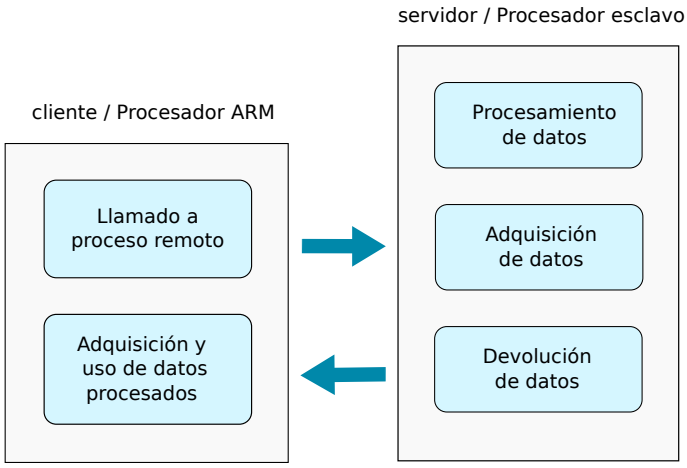


Figura 2.2: Diagrama de comunicación entre un cliente y un servidor

2.2.3 Módulo de evaluación DM8168

El sistema en chip DM8168 se usa en conjunto con una plataforma de desarrollo para poner en práctica soluciones completas. El módulo de evalución (*Evaluation Module*, EVM), como muestra la Figura 2.3 ofrece conectividad y expansiones como puertos de video *component* y HDMI (*High Definition Multimedia Interface*) de entrada y salida, comunicación serie y por red [35], entre otros.

Algunos ejemplos de aplicaciones que se pueden realizar con el EVM DM8168 son:

- Sistemas de transmisión de datos
- Sistemas de videoconferencia
- Visión por computadora
- Procesamiento digital de imágenes (PDI)
- Sistemas de seguridad
- Codificación y decodificación de audio

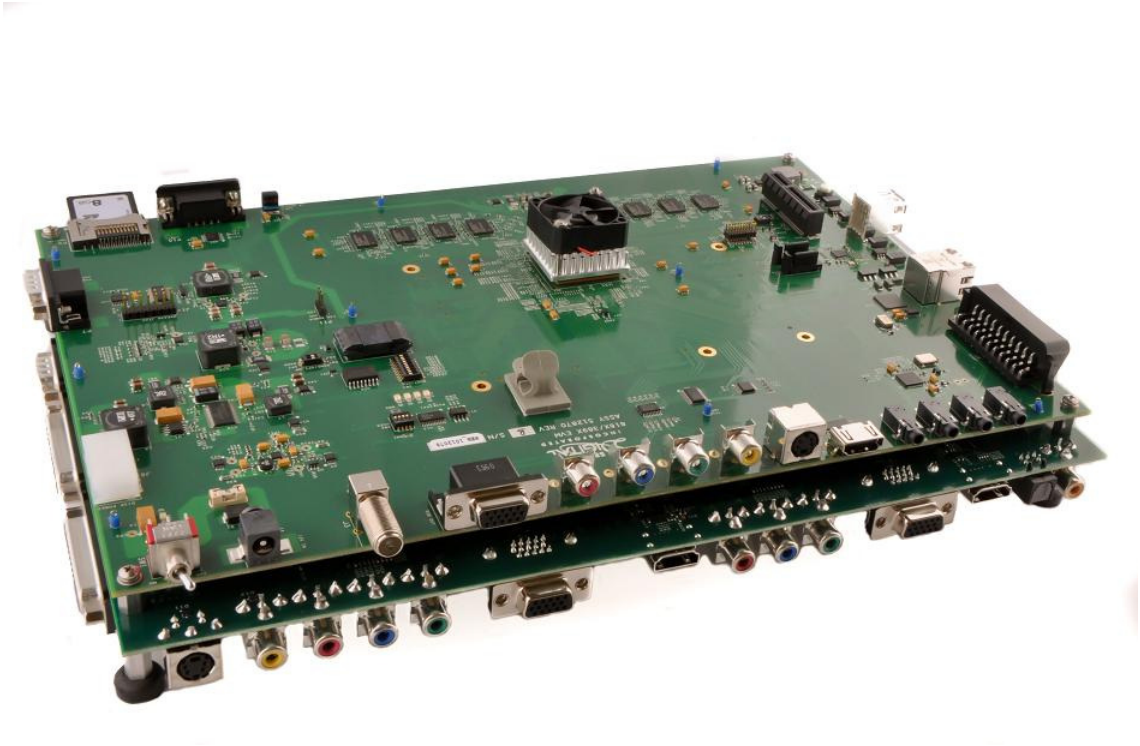


Figura 2.3: Módulo de evaluación DM8168 [19]

La Figura 2.4 muestra un diagrama de bloques para un sistema de comunicación por video que puede llevarse a cabo en el EVM. La plataforma captura video con los puertos de entrada, lo procesa con el SoC DM8168 y se entrega por medio de los puertos de salida a un medio que lo utilice. Como una pantalla de entrada analógica o un sistema de transmisión por red.

2.3 Ambiente de desarrollo GNU/Linux

GNU/Linux es un sistema operativo basado en Unix, caracterizado por ser desarrollado bajo un modelo de código abierto. Linux es un sistema central que administra y manipula recursos de *hardware* para que estos sean usados por programas de software, Linux en conjunto con GNU forman un sistema operativo completo basado en Unix [48].

Muchos tipos de programas se pueden ejecutar sobre GNU/Linux en una computadora de propósito general, esto incluye al SDK para el SoC DM8168 de *RidgeRun*.

2.3.1 Shell

El *Shell* es un intérprete de comandos que permite a un usuario interactuar con el sistema GNU/Linux, en otras palabras es un programa de usuario o un entorno que permite la interacción entre dicho usuario y el sistema [39]. En el desarrollo de aplicaciones es

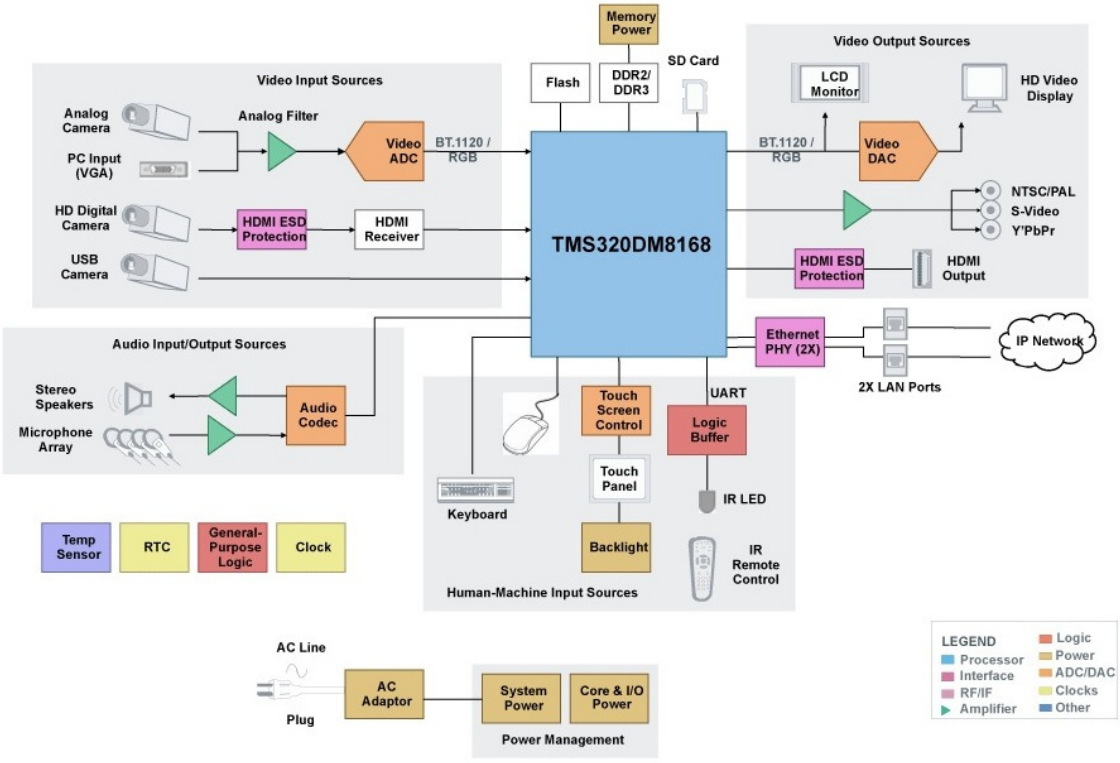


Figura 2.4: Sistema de comunicación por video que se implementa en el EVM DM8168 [10]

frecuentado utilizar el Shell por medio de la linea de comandos, el SDK lo utiliza como un medio para accesar a diferentes recursos de programación.

2.3.2 GNU Make

GNU Make es una utilidad que facilita y organiza la construcción de software. Los programas informáticos pueden volverse muy grandes, esto conlleva a que su construcción se vuelva más compleja y más difícil de manejar. Debido a esas razones, este programa utiliza comandos que permiten al código fuente ser construido de forma ordenada.

La utilidad GNU Make se utiliza por medio de archivos especiales típicamente llamados *Makefiles*, dentro de ellos se definen comandos de construcción de software que son invocados por GNU Make [9].

2.4 SDK: Kit de Desarrollo de Software

El SDK de *RidgeRun* es un sistema integrado que facilita la construcción y ejecución de software en sistemas empotrados. El SDK contiene software con los componentes necesarios para generar e implementar soluciones completas, dentro de los que se encuentran: archivos de configuración para arquitecturas específicas, software de integración para

compiladores y dispositivos empotrados, paquetes de software de código abierto como de software propietario (por ejemplo de TI) [32].

Por medio del SDK es posible programarle a los SoC rutinas de inicialización en memoria, que al momento de encendido del dispositivo permiten la ejecución de un sistema operativo GNU/Linux embebido, uso del sistema de archivos por red, comunicación con PC por medio del puerto serie, entre otros. Todo esto para darle al desarrollador un soporte completo de construcción, implementación y supervisión a las aplicaciones embebidas.

2.4.1 Compilación cruzada

La compilación cruzada es el proceso de compilar una aplicación desde una máquina con un tipo de arquitectura, y el producto es código ejecutable para otra máquina o dispositivo con arquitectura distinta [24]. Los procesadores ARM que pertenecen a un sistema embebido no tiene los recursos de hardware y de software suficientes para propiciar un ambiente de desarrollo completo y funcional como el del SDK, por medio de la compilación cruzada es posible desarrollar software con el SDK desde una computadora de propósito general con arquitectura x86 o x64 hacia un sistema embebido de otra arquitectura.

Existen distintas herramientas de construcción que permiten la compilación cruzada. Para procesadores ARM, el SDK la lleva a cabo con *CodeSourcery*. *CodeSourcery* hace compilación cruzada para arquitecturas ARM y distintos sistemas empotrados basados en Linux [34].

2.4.2 Sistema de integración de aplicaciones

RidgeRun adquiere software propietario o de código abierto con miras a integrarlo al SDK para llevarlo luego a un sistema empotrado. La Figura 2.5 esboza el sistema de integración de aplicaciones del SDK. Este está compuesto de un sistema de configuración que permite personalizar el entorno de desarrollo con las opciones requeridas para integrar aplicaciones. También tiene un sistema de construcción que ofrece definiciones, herramientas para compilación cruzada y software que unifica las secuencias de construcción de contenido.

El SDK además dispone de un sistema de *clases*, estas permite la integración de aplicaciones con sistemas de construcción específicos. Por ejemplo si una aplicación tiene un sistema de construcción basado en *autotools* (un conjunto de herramientas para la construcción automática de paquetes de software [28]), el SDK ofrece una clase de *autotools* que se usa para integrar aplicaciones que implementen este esquema [18].

GStreamer es una biblioteca de software que permite construir componentes de manejo multimedia y provee los medios de operación de estos [27]. La Figura 2.6 enseña como un *elemento* de GStreamer puede formar parte de una *tubería*, la cual permite al elemento la interconexión con otros de entrada y salida. Un posible escenario es usar una tubería que tiene un elemento que captura video, seguido del elemento de conversión de espacios

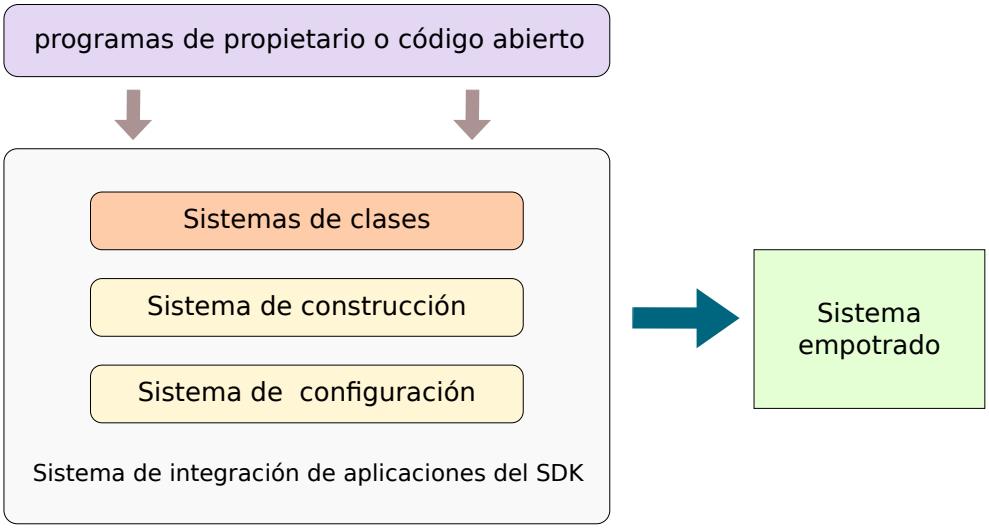


Figura 2.5: Sistema de integración de aplicaciones del SDK

de color y finalmente por un elemento que despliega el video convertido en un monitor de entrada analógica.

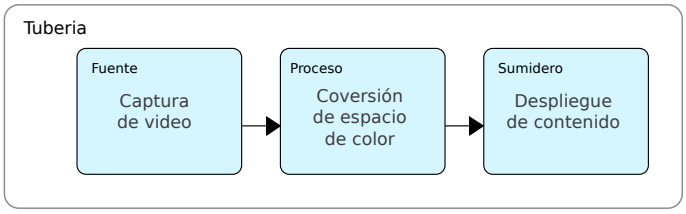


Figura 2.6: Ejemplo de tuberia de GStreamer

El sistema de integración de aplicaciones del SDK también ofrece una clase para generar elementos de GStreamer, dando así más funcionalidad a las aplicaciones con las que la empresa trabaja y facilitando su uso en distintos escenarios multimedia.

2.4.3 Mapa de memoria

Un mapa de memoria describe la distribución de los datos en una tabla de memoria [17]. La arquitectura DM8168 usa un mapa de memoria de tamaño 1GB, el cual es accesado y manejado por diferentes unidades bajo un esquema de memoria compartida.

Por ejemplo, la Tabla 2.1 enseña como la memoria del SoC es asignada al controlador de memoria de Linux, al controlador de memoria física, a regiones de memoria compartida y a código ejecutable de procesadores esclavos [16].

Tabla 2.1: Bloques del mapa de memoria para la arquitectura DM8168

Segmento	Longitud [MB]	Dirección base	Uso
LINUX_MEM_1	364	0x80000000	Partición del sistema Linux
CMEM	20	0x96C00000	Segmento de memoria contiguo
DSP_ALG_HEAP	20	0x98000000	Para reserva de memoria para algoritmos
DSP_DATA	12	0x99500000	Datos del ejecutable del DSP
IPC_SR_VIDEO_M3_VPSS_M3	1	0x9A100000	Memoria compartida interna para el Controlador de Media
Reserved MCHDVICP2 Firmware	7	0x9DD00000	Código ejecutable y datos para el Controlador de Media - HDVICP2
Reserved MCHDVPSS Firmware	17	0x9E600000	Código ejecutable y datos para el Controlador de Media - HDVPSS
UNUSED	3	0x9F900000	Bloque de memoria libre

Búferes de datos

El mapa de memoria del SDK es utilizable a través de búferes de datos. Estos son contenedores de información que se guardan en memoria para el almacenamiento temporal de datos [20], con los búferes se puede transmitir información entre aplicaciones o unidades que tengan acceso a la memoria del sistema.

2.5 Aplicaciones de procesamiento digital de señales

El procesamiento digital de señales (PDS) genera soluciones en la actualidad en diversos campos. El PDS es una ciencia que se encarga del estudio y modelado de señales captadas del mundo real para procesar sus datos por medios digitales [15].

Con el PDS se pueden hacer ecolocalización para sistemas de RADAR/SONAR en aeronaves y submarinos, estudios sismológicos e imágenes médicas como tomografías para diagnóstico [13]. Una rama del PDS es el procesamiento digital de imágenes, en el cual se frecuentan técnicas como la detección y extracción de características.

2.5.1 Operador Sobel como filtro de imagenes

Este operador funciona como una máscara que calcula sobre la imagen el gradiente de su contenido. La Figura 2.7 es un ejemplo de una imagen filtrada con un operador Sobel, donde se revela información de contraste lúminico entre puntos contiguos y esto hace que se expongan zonas donde existen bordes.

El vector gradiente ∇f, se define como la tasa máxima a la que se presenta un cambio de intensidad para una función x,y [47].

$$\nabla f = (\delta(f)/\delta(x))i + (\delta(f)/\delta(y))j \ (1)$$

donde la intensidad se refiere a intensidad lumínica y cada una de las derivadas parciales hace referencia a los cambios lumínicos en los espacios x y y.

Figura 2.7: Imagen original de una copa junto a su imagen filtrada con operador Sobel [6]

La detección se logra utilizando *Kernels*, estos son matrices matemáticas que operan sobre el pixel de la imagen y sus pixeles circundantes. El operador Sobel está compuesto de dos Kernels, uno que detecta los contrastes lumínicos de manera horizontal y el otro de manera vertical. [1]

- Kernel horizontal: este Kernel procesa la aproximación digital de la derivada parcial respecto a la dirección x, $\delta(f)/\delta(x)$

$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$$

- Kernel vertical: procesa la aproximación digital de la derivada parcial respecto a la dirección y, $\delta(f)/\delta(y)$.

$$\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Cada uno de ellos se posiciona sobre el pixel y calcula el cambio de intensidad entre el pixel y sus pixeles circundantes. El resultado del filtro es el cálculo de la magnitud del gradiente $|\nabla f|$, esta se obtiene a partir de la raíz cuadrada de la suma de la potencia al cuadrado de las derivadas parciales [47],

$$|\nabla f| = \sqrt{(\delta(f)/\delta(x))^2 + (\delta(f)/\delta(y))^2} \quad (2)$$

La aproximación digital de $|\nabla f|$ se reduce a la suma de los resultados de las derivadas parciales.

2.6 Interfaces de programación

Las interfaces de programación son medios y recursos de programación para llevar a cabo distintos tipos de tareas [30]. Estas cubren diferentes propósitos para darle funcionalidad a las aplicaciones, por ejemplo una aplicación que corre en un sistema GNU/Linux embebido puede reservar memoria para guardar datos y por medio de funciones transmitir esos datos a un proceso que es llevado en otra unidad.

2.6.1 Codec Engine

Desde la perspectiva de un desarrollador de aplicaciones, Codec Engine (CE) es un conjunto de interfaces que instancian y ejecutan algoritmos estandarizados. Particularmente con CE se pone en práctica el esquema de operación cliente - servidor, este sirve como marco de trabajo que permite a la aplicación cliente comunicarse con contenido remoto del DSP [23]. Por medio de esta interfaz se puede:

- Configurar parámetros y argumentos de operación
- Inicializar algoritmos y procesos
- Ejecutar procesos remotos

Codec Engine es un producto de software Texas Instruments. Este ofrece archivos y secuencias de construcción que de forma manual, se pueden utilizar junto al SDK para construir paquetes ejecutables para el DSP y el SoC.

2.6.2 OSAL: Capa de abstracción de sistema operativo

La capa de abstracción de sistema operativo (*Operating System Abstraction Layer*, OSAL) es una interfaz de programación que permite a la aplicación cliente hacer uso de recursos

del sistema operativo, como registro e inicialización de tareas y manejo de memoria para futuro uso por parte del algoritmo [46].

2.6.3 CMEM: Controlador de memoria física

CMEM es un módulo que facilita el manejo de uno o varios bloques de memoria física contigua y ofrece servicios de traducción de memoria virtual a física. Este controlador evita fragmentación de memoria y asegura bloques de memoria contigua grandes para sistemas que se ejecutan por tiempo prolongado [12].

Por medio del SDK se le asigna a CMEM un espacio en el mapa de memoria. Este sector contiene a los búfer de datos que utiliza el procesador ARM y el DSP para intercambiar datos. El procesador de señales no utiliza una unidad que le permita direccionar o interpretar memoria virtual, por eso el DSP opera con memoria física y CMEM le ofrece estos bloques en un formato en que los pueda accesar.

CMEM trabaja bajo una configuración de bloques llamados *pools*, la Figura 2.8 ilustra como se compone esta estructura. Por medio de comandos se le asigna a CMEM un número de *pools* definido a lo largo de una dirección de memoria y que cada uno de ellos almacene un número particular de búferes los cuales tienen un tamaño específico. Estos se definen con base en las necesidades de memoria de la aplicación y del sistema en el cual se ejecuta.

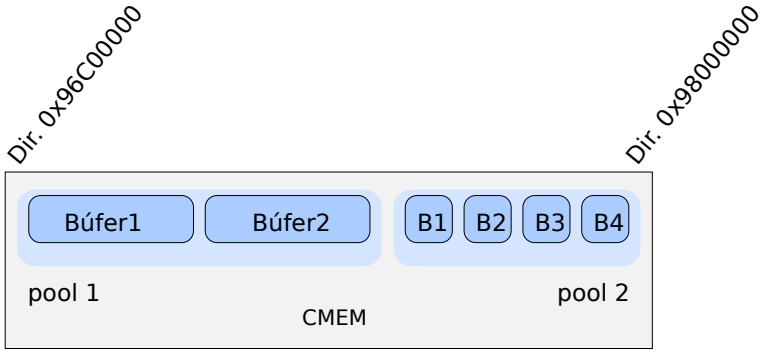


Figura 2.8: Estructura de *pools* y búferes de CMEM

2.6.4 xDM: Interfaz de media digital

Para programar aplicaciones en un ambiente de tiempo real, TI ofrece un conjunto de reglas y recomendaciones para implementar aplicaciones llamado XDAIS, el cual denota estándar de interoperabilidad entre algoritmos. Por medio de reglas de programación y convenciones de uso de recursos XDAIS se encarga de que diferentes algoritmos sean construidos en cierto tipo formato y puedan convivir en tiempo de ejecución [7].

Una expansión de XDAIS es XDAIS-xDM o solamente *xDM*, que se inclina al uso del mismo estándar pero orientado hacia aplicaciones de multimedia. xDM se utiliza a través

de métodos y recursos escritos en el lenguaje de programación C. Este ofrece variables, funciones, estructuras y otros recursos de programación para implementar algoritmos multimedia [42].

IUNIVERSAL

IUNIVERSAL provee los métodos para adherir un algoritmo a la interfaz de multimedia, así aprovechar a la arquitectura DM8168 con procesos que corren en el DSP llamados desde el procesador ARM [8]. Esta interfaz permite a los desarrolladores integrar algoritmos personalizados al DSP como por ejemplo algoritmos de segmentación de imágenes.

El ciclo de vida de un algoritmo se ilustra en la Figura 2.9, desde una aplicacion cliente se hace un llamado a las funciones de la interfaz para llevar a cabo el siguiente ciclo de acciones:

- algAlloc: Se reserva memoria para el algoritmo
- algInit: Se inicializa el proceso que maneja el algoritmo
- algActivate: Activación del algoritmo
- process: Procesamiento de datos
- control: Control de comportamiento del algoritmo en tiempo de ejecución
- algDeactivate: Desactivación del algoritmo
- algFree: Se libera la memoria reservada

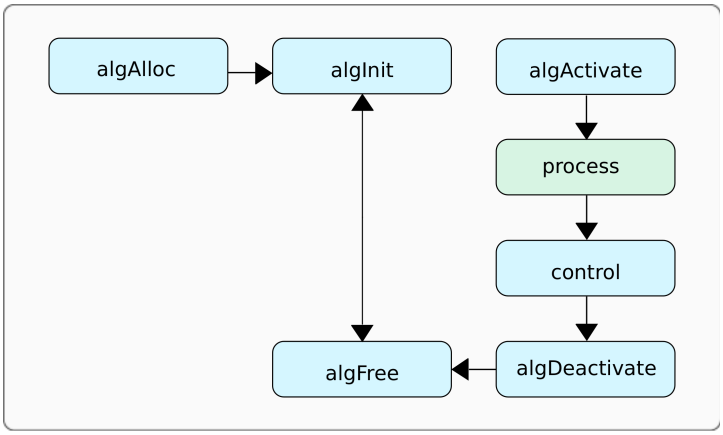


Figura 2.9: Ciclo de vida de un algoritmo [42]

2.7 RTSC: Componentes de Software de Tiempo Real

Los Componentes de Software de Tiempo Real (*Real Time Software Components*, RTSC) es un proyecto perteneciente a *The Eclipse Foundation* que ofrece contenido de bajo nivel usando el lenguaje de programación C, fue hecho con el propósito de dirigirse a todas las plataformas embebidas. Con los RTSC es posible ejecutar procesos remotos en unidades dedicadas como los procesadores de señales C6748.

Los RTSC son desarrollados bajo el paradigma de programación orientada a componentes. A diferencia de la programación orientada objetos este paradigma abarca el ciclo de vida completo del software, estandariza su abstracción de tal manera que cualquier software definido e implementado por un tercero puede ser manipulado e implementado por la empresa [31].

Además, las herramientas que utilizan esta estructura estandarizada le facilita a los desarrolladores la construcción, uso y reutilización de estos componentes, con la finalidad de acelerar el tiempo de creación de aplicaciones embebidas. Por medio de los RTSC se puede manejar el ciclo de vida completo de un componente de software, desde su construcción hasta la monitorización de su operación en tiempo real en un ambiente de ejecución [26].

Una de las características especiales de los RTSC es su portabilidad. El modelo RTSC permite el desarrollo de componentes escritos en C a través de cualquier herramienta de compilación, en cualquier entorno de desarrollo y para cualquier plataforma embebida. Esto revela que además de portabilidad entre sistemas, los Componentes de Tiempo Real tienen además un atributo de agnosticidad.

Los RTSC se manejan en un formato de *paquetes* como el que enseña la Figura 2.10. Los paquetes RTSC pueden agrupar distintos tipos de software, desde código fuente, bibliotecas, archivos de configuración y construcción u otros paquetes RTSC. En un contexto práctico, un paquete es un directorio que contiene el software necesario para ser construido y ejecutado en un ambiente de desarrollo:

Figura 2.10: Contenido de un paquete RTSC

- package.xdc: Este es un archivo que le da un nombre al paquete, por medio de este archivo todo el paquete y su contenido pueden ser manipulados por otro paquete RTSC que quiera utilizarlo.

- package.bld: Este archivo opera como un programa para construir el paquete, desde un punto de vista sencillo este archivo define cuales componentes estarán involucrados a la hora de construir el paquete y que contenido distribuir cuando el paquete haya sido construido.

Los archivos package.xdc y package.bld forman una base para un paquete RTSC, teniendo estos se puede agregar más contenido que define características al paquete, qué es lo que hace y otros atributos.

2.7.1 Servidor

En el desarrollo de aplicaciones para sistemas multiprocesador como el DM8168, se implementa software bajo el modelo de aplicaciones cliente - servidor. La Figura 2.11 muestra como el DSP ejecuta una aplicación servidor que procesa datos provenientes del procesador ARM, esta opera sobre un ambiente de ejecución que cuenta con controladores e interfaces del sistema. Una aplicación servidor es un paquete RTSC, que anida a otro paquete RTSC llamado *codec*, el cual contiene la implementación del algoritmo para procesar los datos [41].

SYS/BIOS6

SYS/BIOS™ 6.x es un sistema operativo de tiempo real avanzado que puede ser implementado en un procesador ARM, un DSP o un microcontrolador. Está diseñado para uso en aplicaciones embebidas que necesitan calendarización, sincronización e instrumentación de tiempo real [36].

La aplicación servidor del DSP es corrida por este sistema operativo, debido a esto es necesario tener este paquete disponible para que el marco de trabajo lo utilice en su flujo de construcción de la aplicación servidor.

2.7.2 Codec

Codec en este contexto, no debe reducirse al concepto de codificador-decodificador, sino a uno más amplio que denota la implementación y empaquetado de un algoritmo. Un codec es un Componente de Tiempo Real que utiliza la interfaz xDM con la finalidad de llevar a cabo una tarea de procesamiento de datos, como un algoritmo de PDI. En el codec se implementa un algoritmo particular que debe operar bajo el estándar XDAIS y el formato de la interfaz xDM [40].

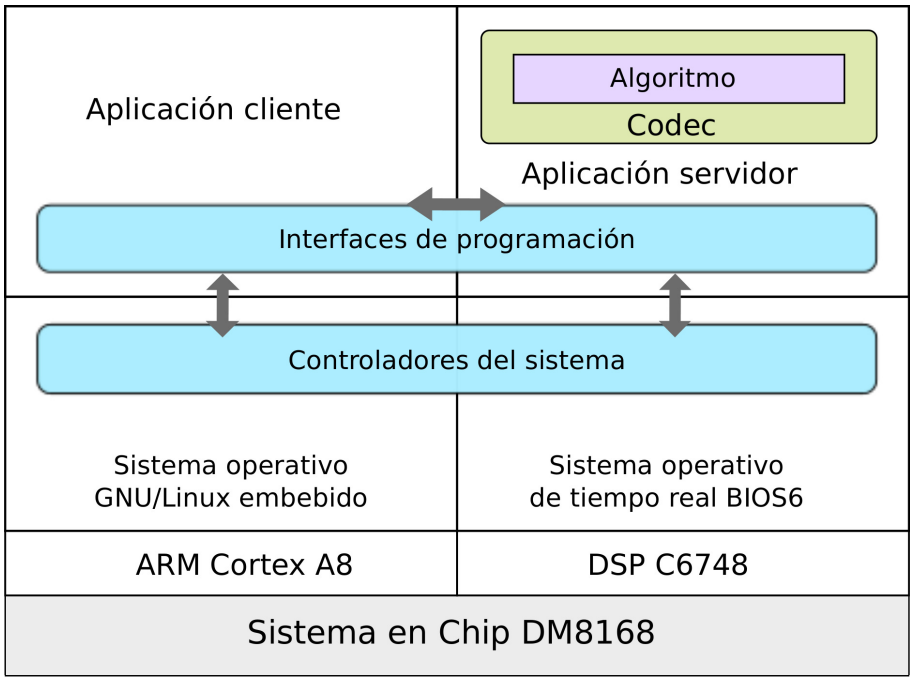


Figura 2.11: Diagrama de aplicaciones cliente - servidor en el SoC DM8168

2.8 xdcTools

xdcTools es un producto que contiene todas las herramientas necesarias para crear, transportar, probar y ejecutar a los RTSC [5]. A partir de xdcTools, el SDK puede generar paquetes de software de tiempo real y contenido ejecutable para el DSP C6748.

2.8.1 xdc

xdc es un comando pertenecienta a xdcTools. Este es utilizado para construir paquetes RTSC y ejecutables que usan estos paquetes [2]. El comando es una envoltura de GNU Make, la cual le provee definiciones, opciones y variables propias de los paquetes RTSC que son necesarias para construir el software.

2.8.2 xs

xs es un comando que interpreta y ejecuta programas de xdcTools [4]. El comando permite lanzar herramientas de xdcTools que generan paquetes RTSC y crean definiciones para el sistema de construcción.

Capítulo 3

Marco de trabajo para la ejecución de procesos remotos

El diseõ del marco de trabajo toma como punto de partida el reconocimiento de los paquetes de software necesarios para construir aplicaciones que se ejecuten en el DSP, esto involucra la utilización de Componentes de Software de Tiempo Real esenciales que generen código que sea ejecutado en un ambiente multiprocesador.

A partir de esto, se elaboran los métodos que incorporen las tecnologías de los RTSC y las interfaces de software necesarias al SDK de la plataforma. Esto para asistir al desarrollador de software con el proceso de construcción de aplicaciones embebidas, bajo el formato en el que el SDK integra aplicaciones al DM8168. Para ello se agrupa e implementa componentes de construcción especializados, contenido de software específico para la arquitectura y secuencias de construcción que permiten entregar contenido ejecutable para el SoC.

3.1 Integración con el SDK

Las soluciones para aplicaciones multimedia que brinda la empresa en su mayoría se desarrollan utilizando al SDK. Por lo tanto, es importante que el marco de trabajo se adapte a los métodos y formatos utilizados por el SDK con la intención de que a los desarrolladores de software les sea rápido y fácil construir aplicaciones embebidas para el SoC, además de darle portabilidad al marco de trabajo y facilitar su mantenimiento.

La Figura 3.1 enseña como el marco de trabajo funciona en conjunto con el SDK y su sistema de integración de aplicaciones. El marco de trabajo se apega a los sistemas de configuración y construcción existentes, también da una clase para la generación de contenido remoto que amplia las posibilidades de construir software, en este caso código ejecutable para el DSP de la arquitectura DM8168.

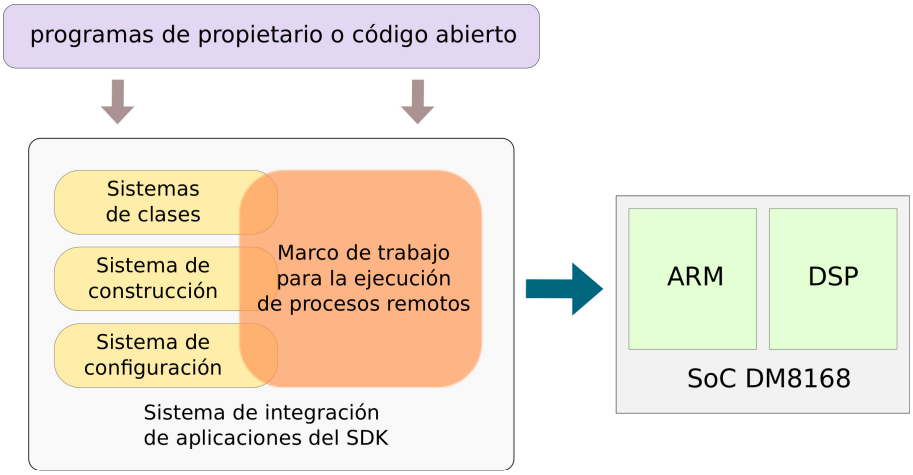


Figura 3.1: Marco de trabajo junto al sistema de integración de aplicaciones del SDK

3.2 Esquema de operación del marco de trabajo

El marco de trabajo tiene como propósito simplicar el proceso de construcción de software que se ejecuta en un ambiente multiprocesador, esto involucra la compilación de paquetes de codecs y servidores para el DSP, además de aplicaciones cliente para el procesador ARM. Para llegar a esto se puede definir un esquema de operación general.

3.2.1 Configuración de entorno de desarrollo

Para que el marco de trabajo construya software debe contar con un entorno de desarrollo configurado, esto signfica contar con rutas, variables, paquetes y utilidades de software reconocidas y listas para usar.

La configuración del entorno se ilustra en la Figura 3.2. Por medio de definiciones del SDK se cargan al entorno las variables y rutas de los paquetes necesarios para construir contenido para el Sistema en Chip, entre los que resaltan: Codec Engine, OSAL, XDAIS, xDM y IUNIVERSAL. Con el entorno configurado, la capa superior encargada de construir software, puede usar estos componentes y generar contenido ejecutable.

3.2.2 Capa de construcción

El entorno de desarrollo configurado es interpretado por la capa de contrucción como un conjunto de definiciones, como rutas y variables que son accesadas cuando se necesiten.

La Figura 3.3 muestra diferentes utilidades y medios del SDK que usa el marco de trabajo para construir software. Por un lado se presentan las herramientas de construcción dedicadas, como CodeSourcery, xdcTools, y CodeGenTools. Este último compila software para la familia de procesadores de señales C6000 entre ellos el C6748 del SoC [45]. La capa de construcción usa estas herramientas junto al sistema de construcción del SDK y

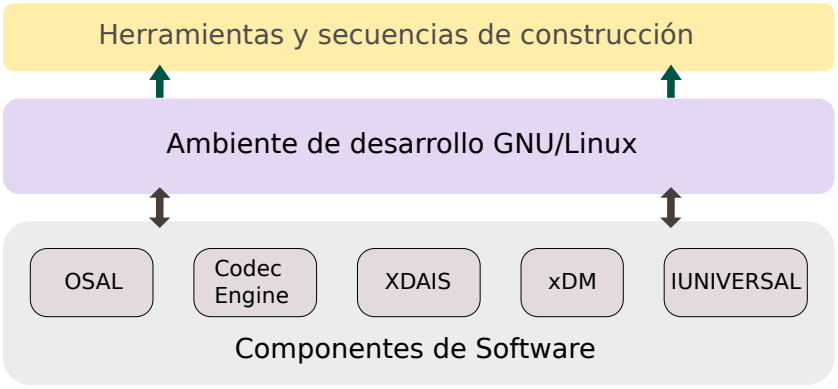


Figura 3.2: Configuración del ambiente de desarrollo GNU/Linux

comandos del marco de trabajo para generar contenido ejecutable para el SoC.

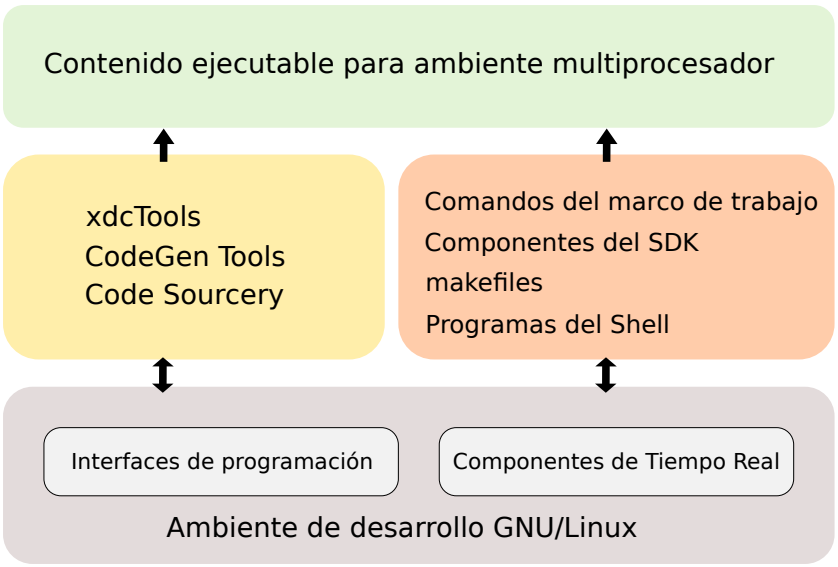


Figura 3.3: Construcción de software para un ambiente multiprocesador

3.2.3 Generación de software

Por medio de especificaciones internas y de usuario el marco de trabajo construye contenido remoto y local. El contenido remoto corresponde al codec y al servidor del DSP C6748, el contenido local es la aplicación cliente que corre en el procesador ARM Cortex A8. La Figura 3.4 muestra como a partir de la capa de construcción se genera este tipo de contenido. Se define el comportamiento que debe de tener el sistema GNU/Linux embebido para que la aplicación ARM+DSP pueda ser ejecutada en el ambiente del sistema empotrado.

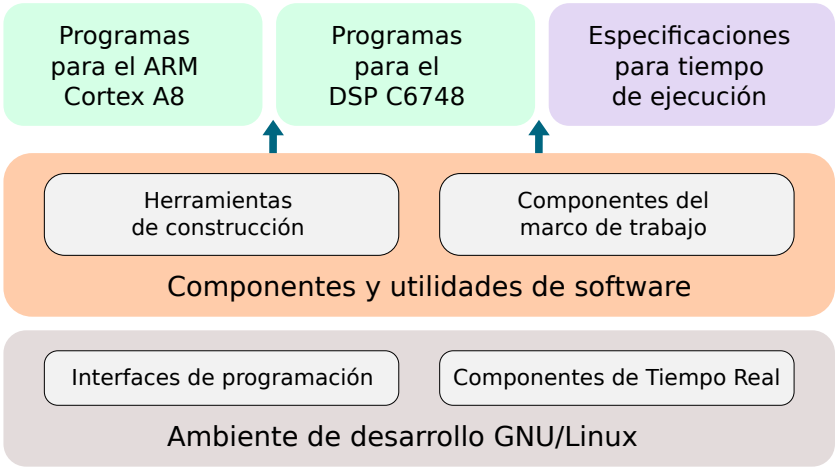


Figura 3.4: Generación de contenido para arquitectura DM8168

3.3 Sistema de construcción

El sistema de construcción consiste en el conjunto de secuencias que lleva a cabo el marco de trabajo para generar software para la arquitectura.

El sistema de construcción utiliza definiciones del usuario y definiciones internas para ejecutar alguna secuencia en particular. A partir de estas definiciones, durante el flujo de construcción el marco de trabajo accesa contenido interno para construir paquetes de software que tienen las características determinadas por el usuario y que tienen las especificaciones apropiadas para el empotrado.

La Figura 3.5 muestra una simplificación de los pasos que necesita el marco de trabajo para construir contenido, los cuales son:

- Configuración

A partir de especificaciones de usuario en un archivo *Makefile*, se definen los nombres y a su vez las rutas de los paquetes que se pretende construir.

- Modos de uso del marco de trabajo

El marco de trabajo tiene secuencias de construcción que utiliza por defecto, por medio del Shell y la utilidad GNU Make el usuario puede interactuar con el marco de trabajo y así determinar esquemas de construcción alternativos.

El modo de uso por defecto es el de integrar al SDK código propietario o código abierto existente. Por ejemplo, si se desea integrar al SDK un codec de codificación de imágenes JPEG, el marco tiene rutinas que permite manipularlo, construirlo y ejecutarlo en el SoC. Este tipo de uso tiene la ventaja de permitir la integración de codecs compatibles con la arquitectura hechos por algún tercero.

Otro uso es la opción de generar paquetes desde cero, el marco invoca un programa de xdcTools que automatiza la creación de paquetes RTSC base. De esta manera se da espacio a los desarrolladores de generar software nuevo para el DSP.

Además de construir el software, se tienen un modo para destruirlo. El modo, usa secuencias para eliminar el contenido ejecutable que el marco de trabajo genera. Esto sin afectar archivos fuente ni archivos base del paquete que son necesarios para reconstruirlo.

- Construcción e instalación de contenido

Finalmente, se construye el software con diferentes herramientas y comandos. Luego se instala el contenido en el sistema de archivos del SoC DM8168.

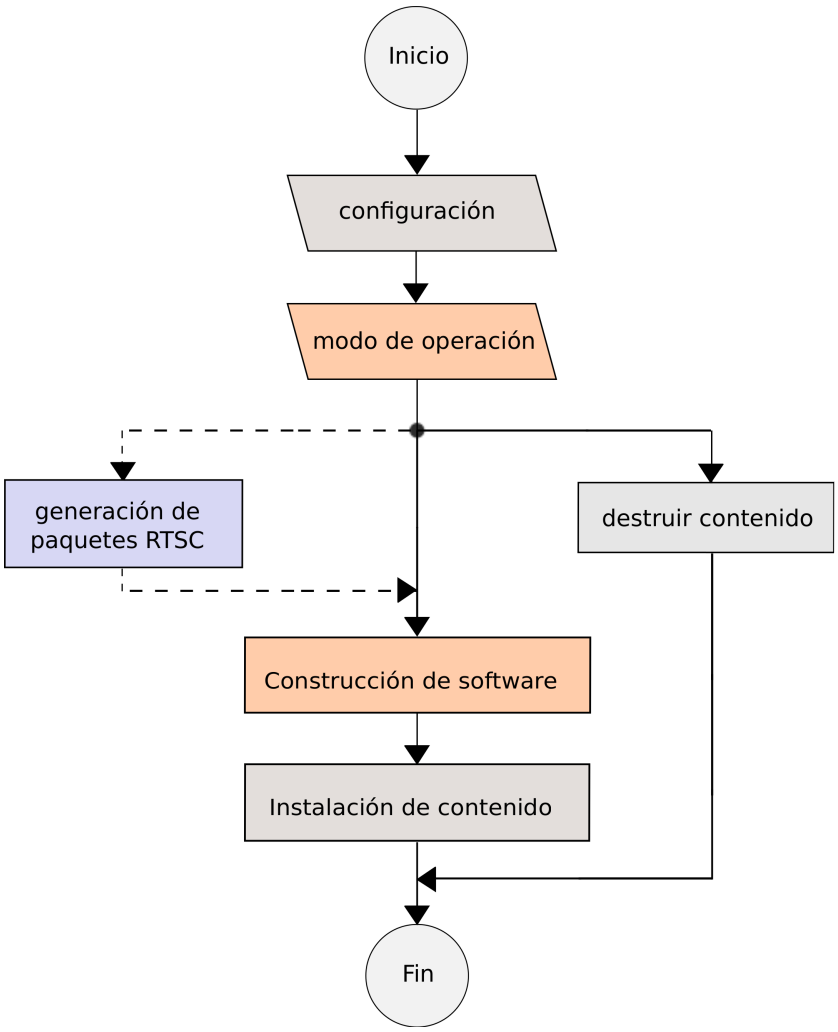


Figura 3.5: Diagrama de flujo simplificado de operación del marco de trabajo

3.4 Clase para la ejecución de procesos remotos

El SDK maneja un sistema de construcción de aplicaciones basado en *clases*, estas son archivos de la aplicación GNU Make que tienen la información necesaria para generar aplicaciones embebidas bajo algún esquema de construcción.

La clase para la ejecución de procesos remotos es creada de manera que conviva con las otras clases y pueda ser invocada de manera externa, como por ejemplo una llamada

hecha por el código fuente del DSP.

3.4.1 Metaprogramas

Para construir software para un ambiente de tiempo real, específicamente paquetes RTSC, la clase es asistida por metaprogramas. Los metaprogramas tambien pueden componer un paquete RTSC, solo que estos en vez de estar orientados a ser un producto final, se encargan de proveer información que asista al proceso de construcción de otros paquetes of software.

Un paquete RTSC que contenga un archivo *package.xdc* y *package.bld*, se construye a partir de definiciones de un archivo llamado *config.bld*. Este especifica cual herramienta de compilación se utiliza, para cual o cuales dispositivos construir el paquete y con cuales opciones.

La Figura 3.6 muestra el contenido de este archivo que utiliza el marco de trabajo para construir paquetes RTSC. A partir de ellos se puede:

- Definir para cual procesador de la arquitectura DM8168 se va a construir código
- Cual herramienta va a construir el contenido
- Bajo que perfil y cuales opciones
- Para cual plataforma o máquina (módulo de evaluación DM8168)
- Cual mapa de memoria va a utilizarse

A partir de los metaprogramas se pueden obtener definiciones para construir aplicaciones que manejen otro esquema de construcción del SDK. En el caso de aplicaciones cliente que son ejecutadas en el procesador ARM, estas pueden ser construidas con otra clase del SDK como la de *autotools*. Los demás sistemas de construcción que se implementan en las clases del SDK no manipulan Componentes de Tiempo Real como xdcTools, esto conlleva a generar una interfaz entre el sistema de construcción que maneja RTSC y otro sistema de construción que no lo maneje.

Para esto, un metaprograma llamado *ARM.cfg* es utilizado por la clase para generar banderas de compilación y enlazado que son entregadas al sistema que construye aplicaciones cliente. En la Figura 3.7 el metaprograma define variables que abstraen información de los paquetes necesarios para construir la aplicación del procesador ARM. A partir de la variable A, se usa una función de xdcTools llamada *xdc.useModule* para obtener las definiciones necesarias del paquete OSAL.

La variable B obtiene las definiciones de la misma manera y utiliza una función especial de Codec Engine llamada *createFromServer()*. Con esta función se abstrae el mapa de memoria que utiliza la aplicación del DSP e información del codec que anida. El uso de este metaprograma y sus variables internas por parte de la clase, permite la generación de banderas de compilación y enlazado que ocupa el sistema de construcción de la aplicación cliente.

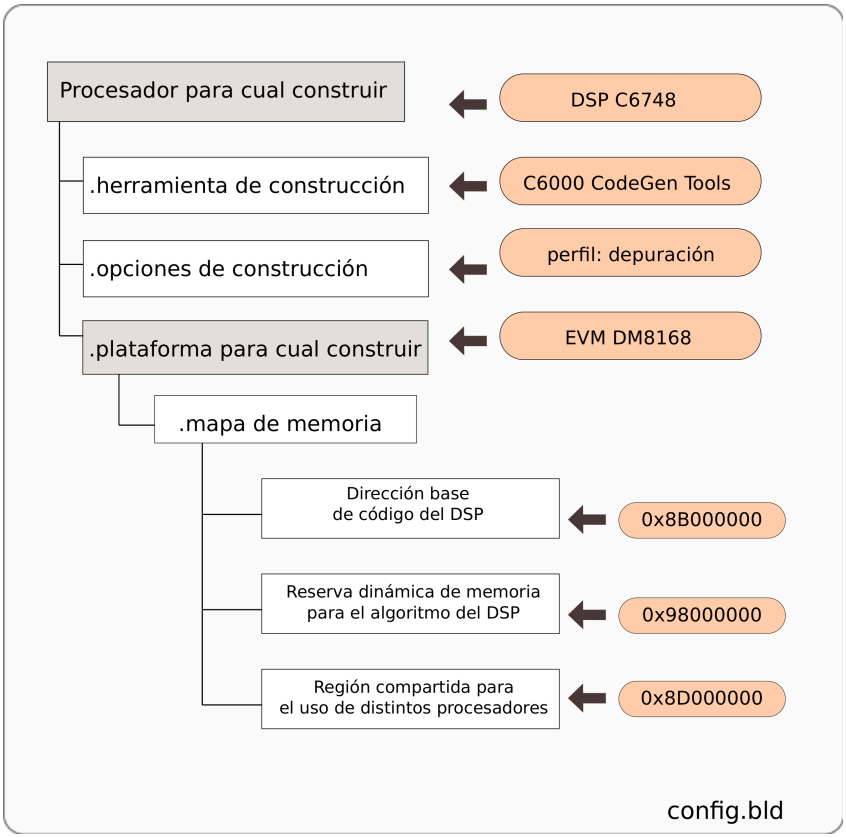


Figura 3.6: Estructura de metaprograma *config.bld* para construir paquetes RTSC

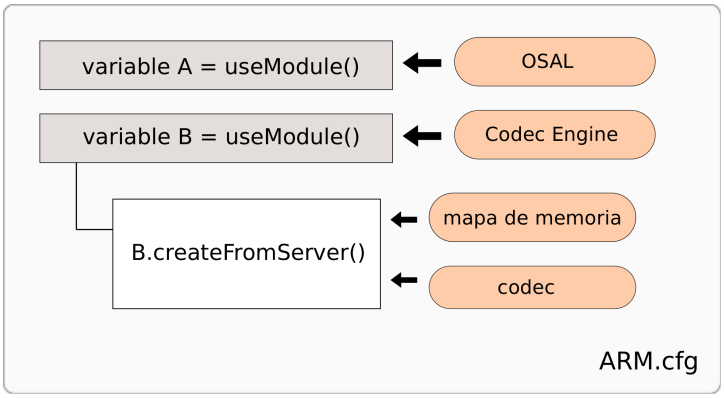


Figura 3.7: Estructura de metaprograma ARM.cfg para configurar aplicación cliente

3.4.2 Comandos de construcción

La clase es utilizada por medio de comandos, estos pertenecen a los archivos *Makefile*. En la Figura 3.8 se ilustra el esquema de un comando típico para construir paquetes, el cual está compuesto por diferentes partes:

- Comando: Invoca una herramienta de construcción
- Componentes: Software que necesita el comando para funcionar
- Argumentos y opciones
- Especificación del contenido que se construye

El comando forma un bloque base que utiliza la clase en diferentes etapas de construcción. Una agrupación de comandos definen las secuencias que usa el marco de trabajo para construir software para el procesador ARM y el DSP.

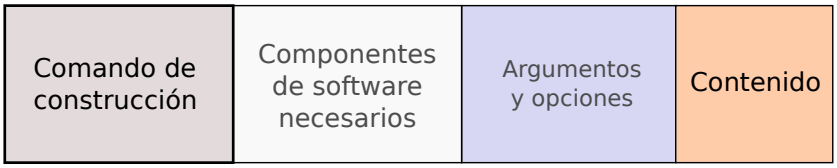


Figura 3.8: Ejemplo de comando utilizado por la clase

3.4.3 Flujo de operación

La clase lleva un orden en el cual construye el contenido, esto se debe a una relación de dependencias entre paquetes. Por ejemplo, para que una aplicación servidor anide un codec, este primero debe de encontrarse construido. Ocurre de esta manera, porque la aplicación del DSP ocupa anidar un codec que tenga las capacidades para ejecutarse en un ambiente de tiempo real.

Teniendo en cuenta la relación de dependencia entre el paquete del servidor y el paquete del codec, se introduce otra dependencia. Como la aplicación cliente se construye a partir de definiciones de la aplicación servidor interpuestas por el metaprograma *ARM.cfg*, es un requisito que la aplicación servidor se encuentre construida.

Esto abre paso a que el flujo de construcción típico sea de primero el codec, después el servidor y por último la aplicación cliente. La Figura 3.9 enseña este flujo de construcción para paquetes existentes que se integran al SDK. Los comandos de xdcTools, *xdc* y *xs* ocupan un conjunto de rutas que definen los componentes para generar software llamadas *Rutas XDC*. Para el caso del codec, estas rutas apuntan a las interfaces XDAIS y IUNIVERSAL ya que el algoritmo se implementará a través de estas. Además de las rutas, xdcTools opera con *Argumentos XDC*, como argumento se introduce al metaprograma *config.bld* el cual tiene las definiciones de la arquitectura y la herramienta de compilación. En las *Opciones XDC* se utiliza la opción de *reporte*, la cual detalla en la línea de comandos del Shell todas las etapas de construcción. Otra de las opciones es *entrega*, esto especifica que al finalizar la construcción del paquete se genere una versión comprimida de este. Con esto el paquete comprimido puede ser transportado a otro entorno de desarrollo y ser reutilizado. Al final del comando se especifica el codec que se desea construir.

Cuando el codec termina de construirse, la clase delega la construcción al siguiente comando que construye a la aplicación servidor. Este trabaja bajo el mismo formato del codec, las *Rutas XDC* contienen ahora más componentes para involucrarlos en la construcción, como el paquete BIOS6 y el codec construido en la etapa anterior.

Consecuentemente con el comando *xs* se especifican las rutas necesarias, como la del paquete del servidor. El comando invoca a un programa llamado *configuro* [3]. Junto

a los metaprogramas introducidos como argumentos, el comando abstrae la información necesaria para generar banderas de compilación y enlazado que puedan ser interpretadas para el sistema de construcción de la aplicación cliente. La aplicación que se ejecuta en el procesador ARM es contruida con alguna de las clases que ofrece el sistema de construcción del SDK.

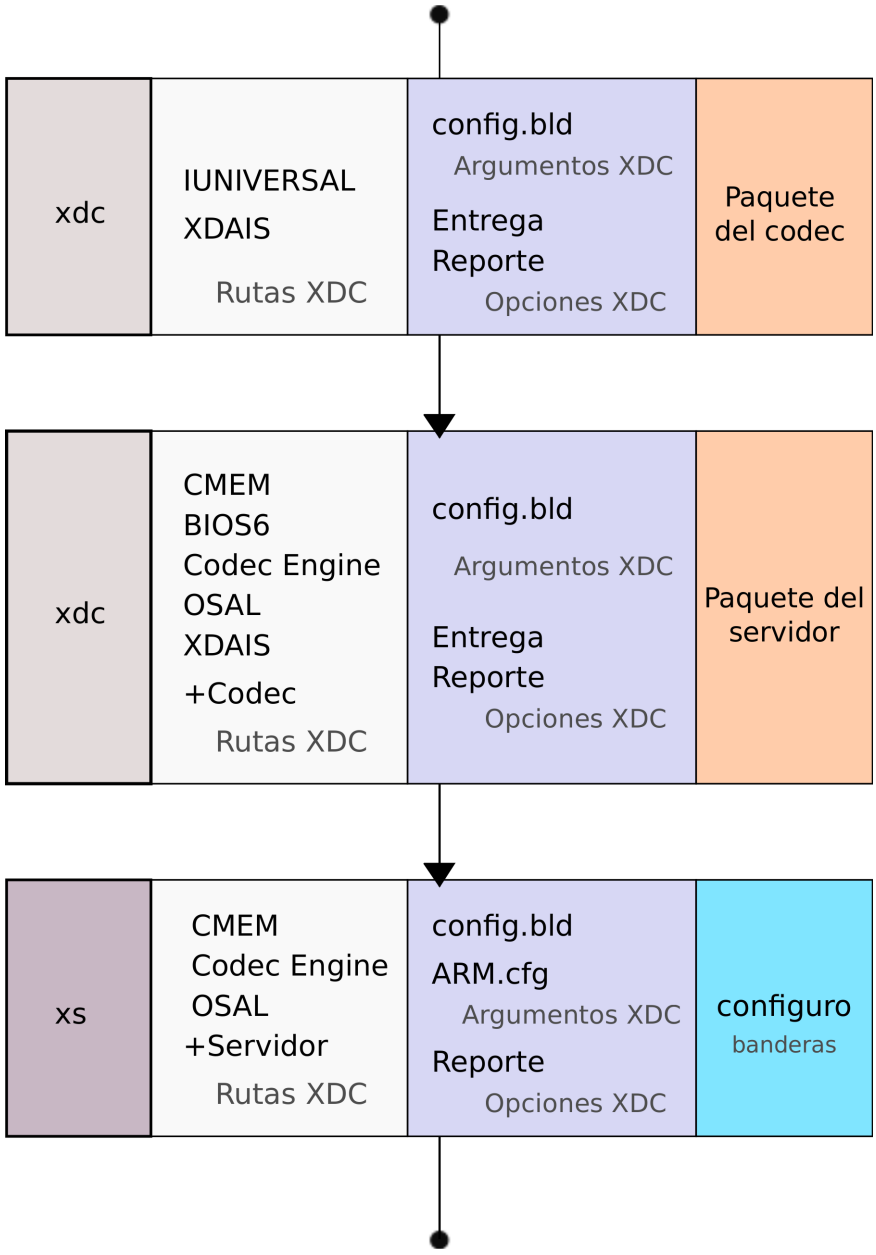


Figura 3.9: Sequencia de comandos de la clase

A parte del flujo típico, el conjunto de comandos de la *clase* permite manejar la construcción del contenido de manera individual o por medio de distintas combinaciones. Adicionalmente, esto revela que el marco de trabajo por cada tarea que realice, debe de conducir el flujo de construcción de tal manera que si un paquete depende de otro, el paquete requerido se encuentra construido y sea localizable por los demás paquetes que lo necesiten en el entorno de desarrollo.

Además del flujo de construcción de paquetes existentes, la clase permite crear paquetes RTSC desde cero como una utilidad del marco de trabajo. La Figura 3.10 muestra los comandos para generar un codec y una aplicación servidor desde cero. La clase utiliza el comando *xs* para lanzar programas de xdcTools que automatizan la generación de paquetes. Con las rutas y opciones especificadas el comando ejecuta la aplicación *gencodecpkg* que genera automáticamente un paquete base para el codec. A parte del codec, se puede lanzar la aplicación *genserverpkg* que genera una aplicación servidor con las especificaciones del metaprograma *config.bld*.

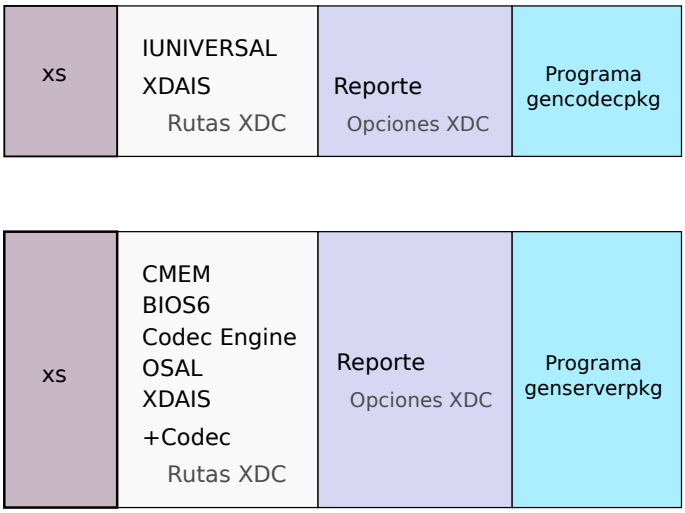


Figura 3.10: Comandos para generar paquetes RTSC base con xdcTools

Hasta este punto se conoce el orden que sigue el marco de trabajo y los comandos que puede utilizar, se elabora un diagrama 3.11 con una secuencia de operación más detallada. Esto involucra el contenido necesario para correr una aplicación cliente-servidor en el SoC. Primero se define cual modo de uso conduce el marco de trabajo, a partir de esto la clase tendrá una secuencia de operación definida. Después se inicia la construcción de cada componente en orden; primero el codec, de segundo el servidor y por último el cliente. Finalmente cuando todo el contenido termina de construirse, este se instala en el sistema de archivos para formar una aplicación ARM+DSP.

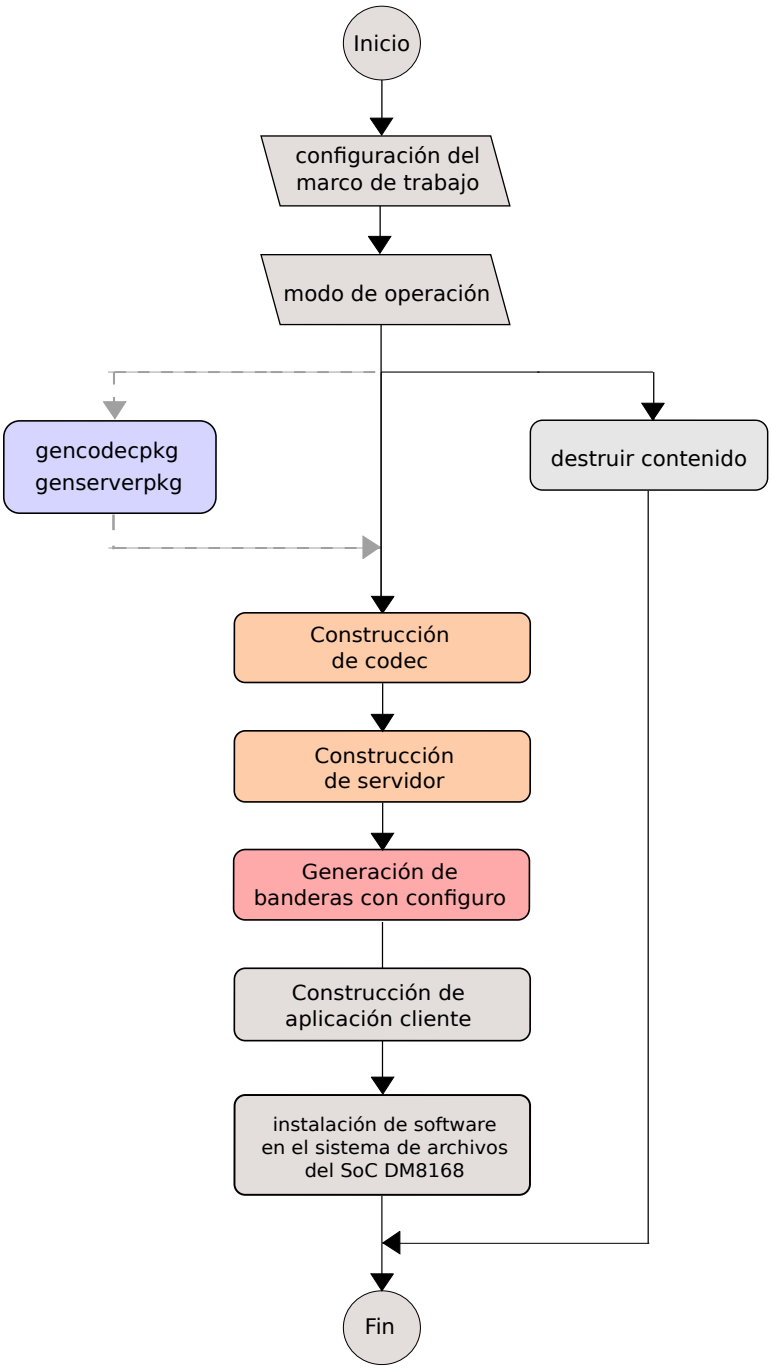


Figura 3.11: Diagrama de secuencia de construcción

Capítulo 4

Implementación de detector de bordes

El marco de trabajo se utiliza para integrar al SDK una aplicación de prueba para filtrar imágenes. TI tiene una biblioteca de procesamiento digital de imágenes llamada IMGLIB [37]. Esta ofrece una función del lenguaje de programación C que implementa un operador Sobel utilizando dos Kernels de imágenes.

4.1 Personalización de la interfaz de media digital

Para procesar datos en el DSP el operador Sobel es llamado a través de la interfaz xDM, esto implica que la función de IMGLIB se encuentre apegada al formato y métodos de la interfaz.

4.1.1 Función *process*

La función process de xDM es llamada para que tome los datos provenientes del la aplicación cliente, que los procese con el detector de bordes y que devuelva la imagen filtrada. La Figura 4.1 enseña como la biblioteca de IMGLIB está envuelta por la función *process*. Además, con argumentos de esta función, se le pasa al operador Sobel información del tamaño de la imagen. Esto permite que la imagen sea procesada de la forma correcta. Existen algoritmos que al finalizar devuelven argumentos de salida, la función *process* permite manejar estos argumentos. Sin embargo para este caso particular el operador Sobel de IMGLIB no los necesita y por lo tanto no son devueltos.

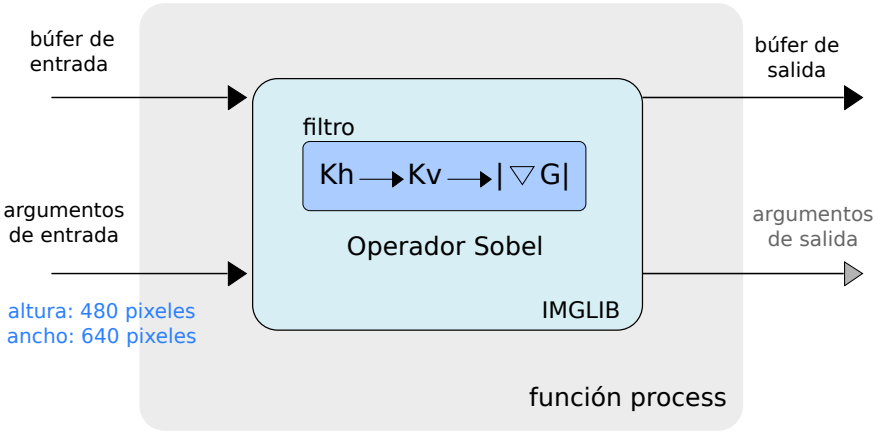


Figura 4.1: Envoltura del operador Sobel en la función *process* de la interfaz xDM

4.1.2 Expansión de argumentos de la interfaz IUNIVERSAL

IUNIVERSAL es una interfaz genérica, esta no tiene por defecto atributos de altura y ancho de imágenes. Para que el codec responda a un tamaño de imagen definido por la aplicación cliente, el tamaño es manipulado a través de la interfaz. La Figura 4.2 muestra como un codec anida la estructura de argumentos de entrada de IUNIVERSAL. A partir de esta se adhieren los atributos necesarios para que la interfaz pueda manipular la información de las dimensiones de la imagen y operar este contenido en el filtro. La aplicación se trabaja con una imagen cuya resolución es de 640 pixeles a lo ancho y 480 pixeles a lo alto (640x480).

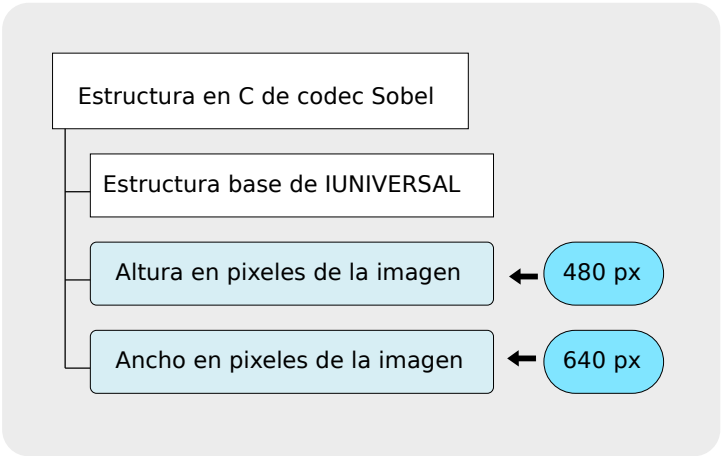


Figura 4.2: Expansión de estructura de IUNIVERSAL del codec Sobel

4.2 IMG_sobel_3x3

La Figura 4.3 muestra un diagrama de como una imagen es procesada por el filtro de imágenes. La función de la biblioteca IMGLIB recibe las dimensiones de la imagen de entrada para operar sobre cada pixel de esta. Por cada pixel, se calcula el gradiente de

forma horizontal y vertical con base en la información de los pixeles circundantes. Después se calula la magnitud del gradiente con la suma de los resultados de los Kernels.

La umbralización se hace a partir de funciones condicionales. Para un pixel, el cálculo de la magnitud del gradiente es llevada a cero si es inferior ante un parámetro de umbral definido o llevado a su máximo si el cálculo de la magnitud del gradiente es mayor ante el parámetro de umbral. Por cada pixel procesado se realiza una escritura en el búfer de salida. Si el búfer de entrada no ha sido recorrido por completo se itera las veces necesarias para filtrar la imagen por completo.

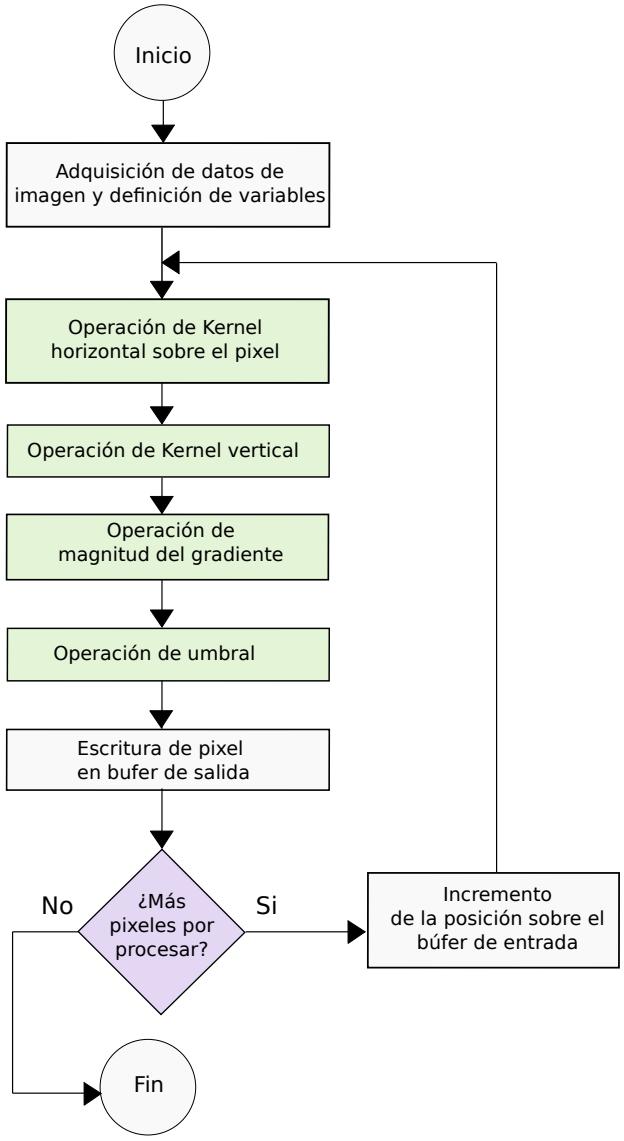


Figura 4.3: Diagrama de flujo de la operación del filtro Sobel

4.2.1 Formato de la imagen

El filtro de imágenes no puede reconocer el tipo de archivo que recibe a la entrada ni la capacidad de preprocesarlo, debido a esto los datos de la imagen deben de tener formato

crudo. El formato crudo presenta únicamente datos propios de la imagen sin metainformación que la caracterize, así es posible que el operador recorra de manera iterativa el búfer de entrada pixel por pixel, sin la necesidad de reconocer algún formato y tener que decodificarlo para aplicar el filtro.

Además, como el operador Sobel calcula el gradiente del contenido, la información de cromacidad de la imagen (como el color de esta) es irrelevante. Más bien, el filtro opera sobre el contraste que existe entre diferentes regiones de la imágen, precisamente los contrastes lúminicos. Debido a lo anterior la imágen de entrada se presenta en escala de grises y formato crudo. Se considera la relación de tamaño en pixeles de la imagen con el tamaño en memoria de la misma. Tomando un caso práctico se utiliza una tuberia de GStreamer que genera un archivo crudo, en escala de grises y a 8 bits por pixel,

$$8 \text{ bits/pixel} = 1 \text{ byte/pixel}$$

$$640 \text{ x } 480 \text{ pixeles} = 307200 \text{ bytes}$$

Estos datos se usan para reservar memoria, y así asegurar el espacio suficiente en el mapa de memoria cuando la aplicación se ejecute.

4.3 Aplicacion cliente

La aplicación que se ejecuta en el procesador ARM llama al proceso remoto utilizando Codec Engine. Además de eso, realiza los preparativos necesarios para ser una aplicación funcional en un entorno de ejecucción GNU/Linux embebido. La Figura 4.4 ilustra como funciona la aplicación desde el comienzo hasta el final.

La memoria que se utiliza para transmitir los datos del procesador ARM al DSP debe de ser configurada. En esta etapa se define que la memoria es física contigua, además de sus banderas y alineación. Con los parámetros de memoria listos, la función *Memory_Alloc* de la interfaz OSAL reserva espacio para los búfer de entrada y salida, el controlador CMEM se encarga de reservar memoria física contigua a partir de la solicitud de la aplicación cliente.

Codec Engine debe cargar componentes para manejar un proceso remoto. Con la apertura de un proceso de Codec Engine a través de la función *Engine_open* es posible comunicarse con otros procesadores bajo un esquema cliente - servidor, esto abre el paso a la inicialización del algoritmo con la función *UNIVERSAL_create* y hacer uso de él.

La aplicación hasta el momento ha hecho los preparativos para ejecutar el algoritmo en el DSP, si se da el caso en que: no se pudo reservar memoria debido a espacio insuficiente o parámetros inválidos, no se pudo cargar los componentes de Codec Engine o no se inicializó el algoritmo, se libera la memoria que ha utilizado la aplicación hasta el momento.

Por otro lado, si los preparativos están listos se puede empezar el llamado al proceso del DSP. Como el operador Sobel necesita tener las dimensiones de la imagen, se definen esos valores en los argumentos de entrada de IUNIVERSAL (que se encuentran expandidos) para manejar estos datos. La aplicación lee un archivo (de formato crudo) que es colocado en el búfer de entrada y se invoca a la función *process* para ejecutar el filtro de imagenes. Cuando el cómputo de los datos termina, la información recibida en el búfer de salida es escrita en un archivo, este archivo contiene la imagen con bordes detectados por el operador Sobel. Finalmente se libera la memoria ocupada por la aplicación.

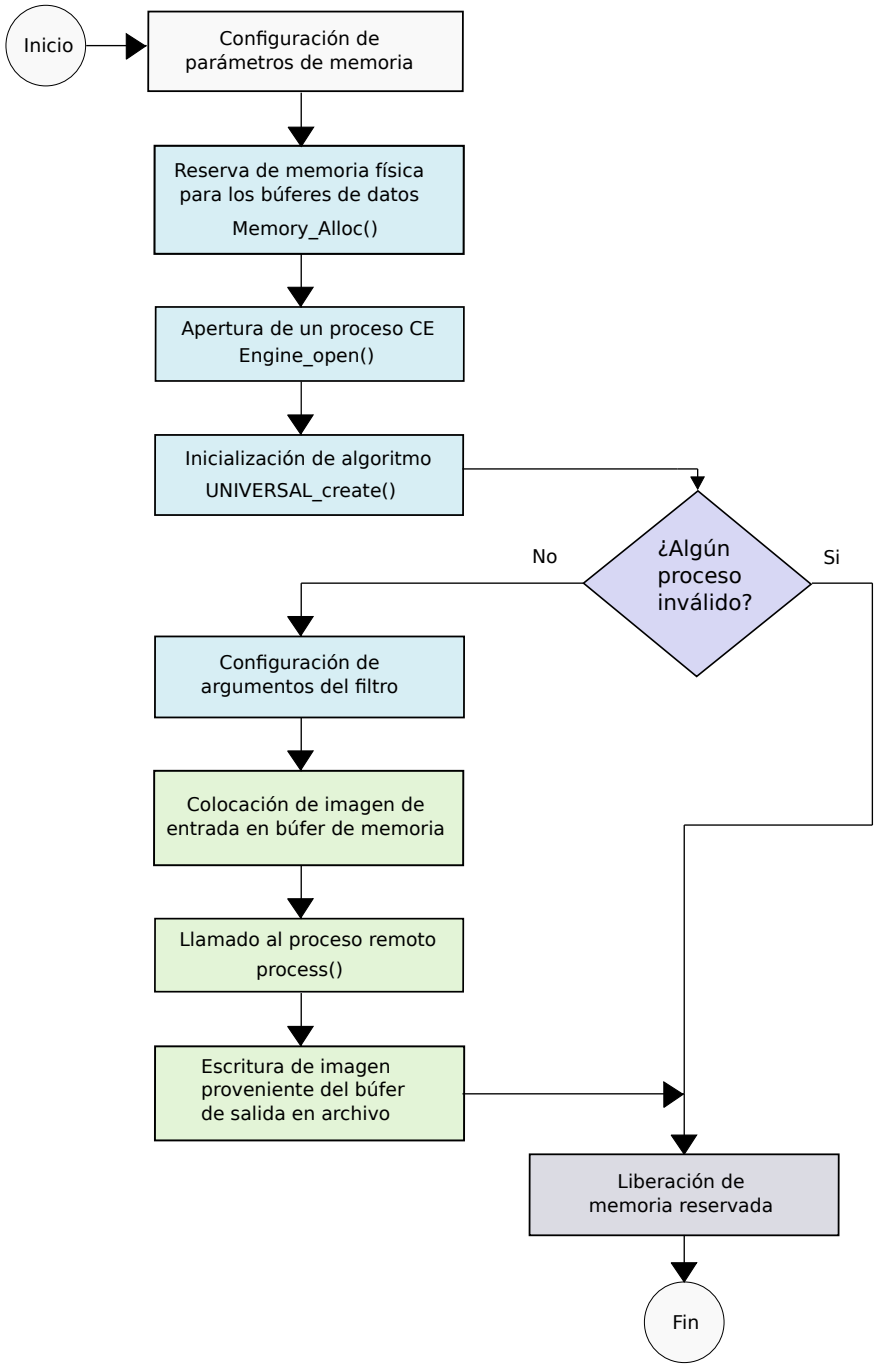


Figura 4.4: Diagrama de operación de aplicación cliente

Capítulo 5

Configuración del ambiente de ejecución

Para llevar la implementación del filtro de imágenes a la arquitectura, el marco de trabajo da especificaciones al sistema operativo GNU/Linux embebido. Estas indican bajo cuales condiciones de operación la aplicación puede ejecutarse, esto se hace por medio de un programa del Shell que es entregado junto a la aplicación del filtro al momento de la instalación. Este lleva a cabo inicializaciones y configuraciones del sistema antes de que el detector de bordes sea ejecutado.

Por medio de la Figura 5.2 se representan las tareas que hace el programa del Shell, que consisten en cargar módulos al sistema GNU/Linux embebido y configurar el uso de memoria de la partición.

5.1 Partición de Linux

En el mapa de memoria del EVM DM8168 se toma una porción de la partición de Linux para alojar el ejecutable del DSP. Esto lleva a que el programa reduzca con un comando el tamaño de la partición:

```
MEM=168M
```

Esto reduce la partición del sistema Linux de 364MB a 168MB.

5.2 Carga de módulos

La estructura de CMEM para la aplicación de prueba se muestra en la figura 5.1. La aplicación cliente necesita reservar búferes de entrada y salida. Al momento en el que la aplicación cliente llama a la función para reservar memoria, CMEM atiende la solicitud

y le asigna uno de los búfer disponibles. Esto se realize por medio de un parámetro que define las características de memoria y un comando que carga el módulo:

```
CMEM_PARAMS= phys_start=0x96C00000 phys_end=0x98000000 pools=2x307200
```

```
modprobe cmemk CMEM_PARAMS
```

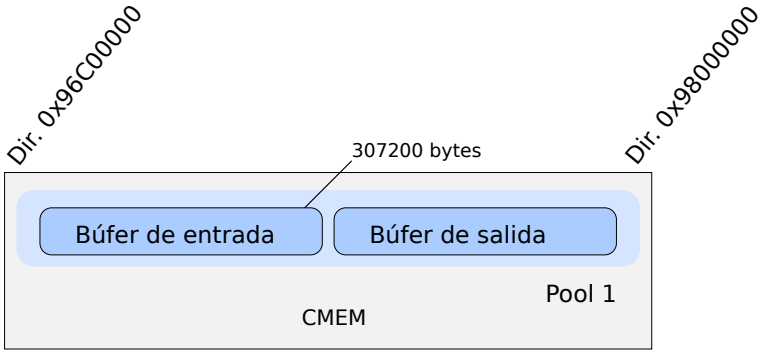


Figura 5.1: Configuración de CMEM para alojar los búferes del DSP

Esto define una región de CMEM para esas direcciones de memoria, con un *pool* que tiene dos búferes de datos cada uno de 307200 bytes de espacio en memoria. Todos estos comandos son ejecutados por el programa de configuración del ambiente de ejecución que muestra la Figura 5.2. Otro de los módulos que se carga es *syslink*. syslink provee software de conectividad entre múltiples procesadores [22], el cual es insertado al sistema GNU/Linux embebido con el siguiente comando,

```
modprobe syslink
```

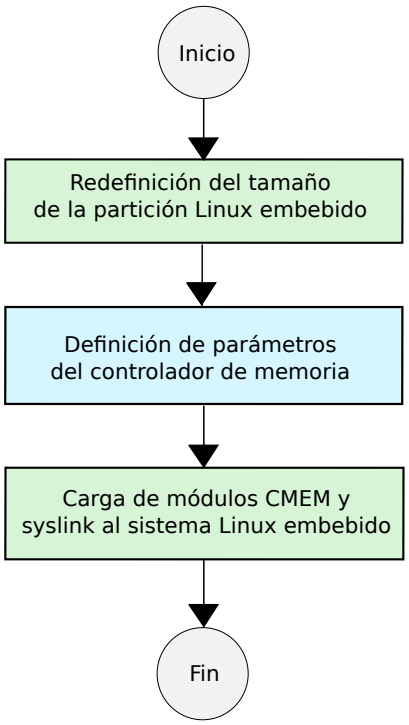


Figura 5.2: Configuración del ambiente de ejecución para la aplicación del filtro de imágenes

Capítulo 6

Resultados y Análisis

6.1 Factorización de secuencia de construcción

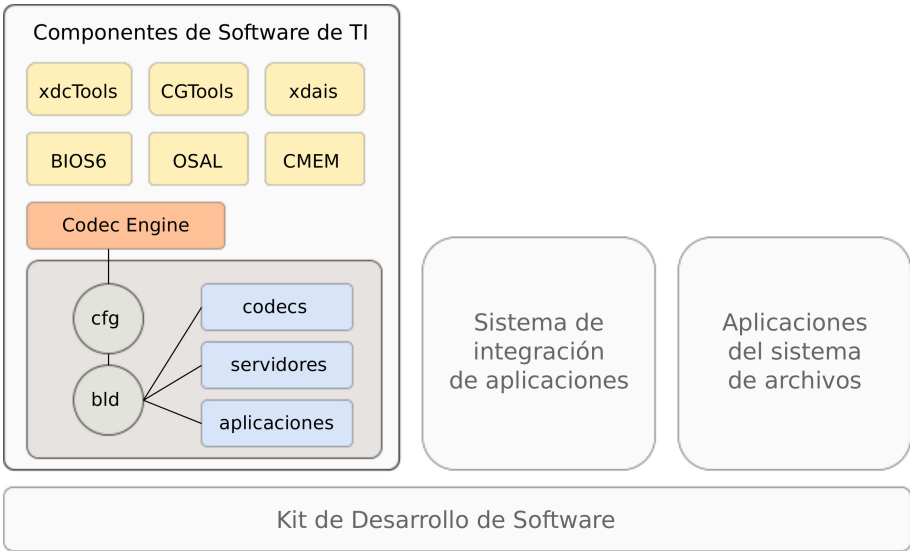
El proyecto se encarga de reordenar y agrupar la sencuencia de construcción de TI y pasarla a un formato aprovechable por un desarrollador del SDK. La Figura 6.1 expone como el sistema de construcción se transforma, en la subfigura *a* la construcción de paquetes para un ambiente multiprocesador en la arquitectura DM8168 se realiza desde Codec Engine.

Este contiene software de configuración denotado *cfg* y un sistema de construcción *bld* que permiten generar aplicaciones cliente, servidor, y codecs. Este esquema de construcción no utiliza el sistema de integración de aplicaciones del SDK. Por lo cual, no se aprovechan los sistemas de configuración y construcción existentes, esto hace que se subutilicen recursos del SDK para el sistema en chip DM8168 y resta dinámica al entorno de desarrollo para manipular software.

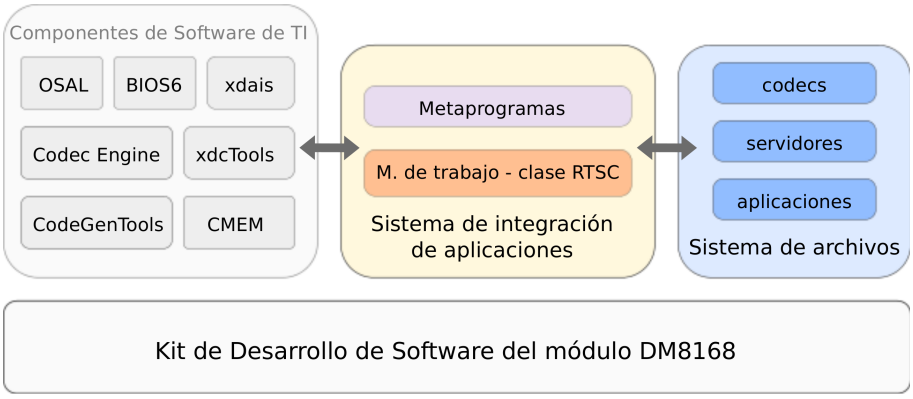
Por otro lado la subfigura *b* expone el rediseño de la secuencia de construcción, el marco de trabajo toma las definiciones y componentes necesarios que operan bajo el preámbulo de TI y los traduce al formato del SDK. Los metaprogramas también forman parte del marco de trabajo. Con esto es posible generar de una manera simplificada código ejecutable y que además forme parte del repositorio interno de aplicaciones del SDK y del sistema de archivos de GNU/Linux embebido.

6.2 Reducción del tiempo de construcción

Con la factorización, se da una reducción en el tiempo de construcción de software para la arquitectura. En la Tabla 6.1 se muestra un experimento en el cual se reducen los tiempos que emplea un desarrollador con conocimientos en el manejo de paquetes RTSC y el SDK. Se divide por etapas, la etapa de configuración del ambiente de desarrollo requiere definir todas las rutas de paquetes y la versión exacta de los mismos, además deben definirse



(a) Sistema de construcción original



(b) Sistema de construcción factorizado

Figura 6.1: Comparación de sistemas de construcción original y factorizado

Tabla 6.1: Reducción de tiempo de construcción de software

Etapa de construcción	Tiempo con sistema original (min)	Tiempo con el marco de trabajo (min)
Configuración de ambiente de desarrollo	12.4	0
Construcción de contenido	6.8	5.2
Configuración de ambiente de ejecución	1.8	0.6
Totalidad del proceso	21.0	5.8

compiladores junto a sus opciones, la arquitectura para cual construir y bajo que modelo de operación. El marco de trabajo contempla todas estas definiciones con la finalidad de que el desarrollador dedique su tiempo a construir e implementar código.

La construcción de software con el sistema original utiliza definiciones y componentes que se encuentran dispersos. Esto ocasiona que el flujo de construcción deba de hacer búsqueda de utilidades de programación en diferentes rutas. Con el marco de trabajo las definiciones para generar código se encuentran centralizadas.

Por medio del programa que configura el ambiente de ejecución, el desarrollador define el entorno donde la aplicación del filtro de imágenes se ejecuta. Ante la ausencia del programa de configuración, el desarrollador tendría que identificar las necesidades de la arquitectura para ejecutar la aplicación de prueba, además de ejecutar los comandos pertinentes. El cambio del sistema de construcción hace una mejora en el tiempo de creación de aplicaciones de un 73.4%.

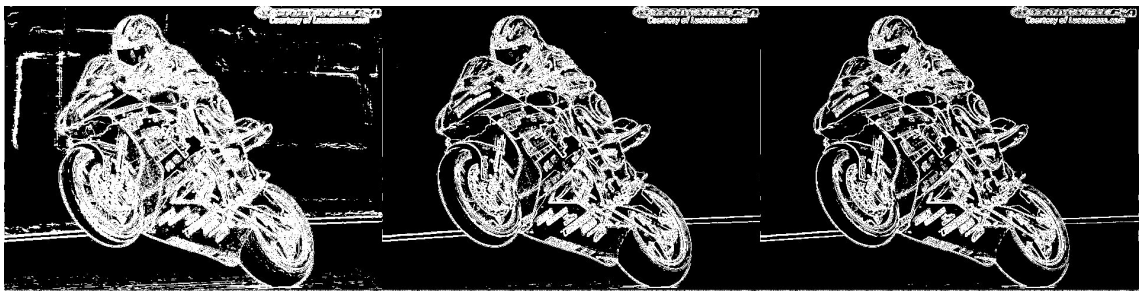
6.3 Detección de bordes en imagen de prueba

Se toma como punto de partida una imagen de prueba con sus componentes; lúminica y cromáticas. La Figura 6.2 ilustra como la imagen del motociclista [29] es convertida a escala de grises con ayuda de una tuberia de GStreamer. La tuberia toma una imagen que proviene de un archivo en formato JPEG y le sustrae las componentes cromáticas. La imagen descompuesta es procesada por la aplicación, y el operador Sobel actúa sobre la imagen para resaltar sus bordes. Cabe notar que este es un resultado parcial, las zonas de color gris no están umbralizadas y en este contexto se interpretarían como ruido ya que para objeto de detección de bordes no se pueden interpretar. Dado a esto se le aplica una variación de umbral para resaltar bordes y oscurecer las zonas que no lo sean.

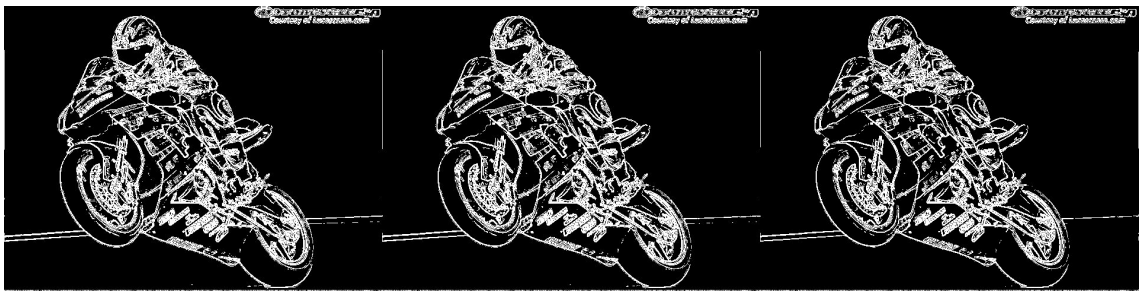
Para el formato de la imagen de prueba, la luminancia tiene una excursión de 0 a 255 para valores enteros. Cero corresponde al color negro y 255 al máximo de esta componente

(a) **(b)** **(c)**

Figura 6.2: Imagen de ejemplo con resolución de 640 x 480 convertida a escala de grises y procesada por el operador Sobel



(a) Umbral=28% **(b)** Umbral=56% **(c)** Umbral=69%



(d) Umbral=82% **(e)** Umbral=86% **(f)** Umbral=96%

Figura 6.3: Juego de imagenes filtradas por operador Sobel con diferentes niveles de umbral

que sería blanco. La Figura 6.3 expone diferentes valores de umbral que se le asignan al algoritmo para cambiar su comportamiento ante el cálculo de la magnitud del gradiente.

La figura muestra que con parámetros de umbral bajos como el de la subfigura *a*, el operador Sobel tiene un comportamiento más inclusivo ante contrastes lúminicos que presente la imagen. Esto ocasiona también, que ante contrastes lúminicos poco intensos en regiones compactas, el filtro pierda claridad en la detección.

Por otro lado, con parámetros de umbral altos como el de la subfigura *f*, el operador funciona con más nitidez ante contrastes lúminicos más intensos en regiones compactas. Esto conlleva a que tenga un comportamiento más exclusivo en detección y que contrastes lumínicos tenues no sean detectados y por lo tanto se pierdan.

Tabla 6.2: Carga de procesamiento de operador Sobel ejecutado en el procesador ARM y en los procesadores ARM y DSP

	Proceso local (%)	Proceso remoto (%)
Arm Cortex A8	96	3
DSP C6748	0	44

6.4 Carga de procesamiento

El operador Sobel se implementa de dos formas en el SoC con el motivo de comparar la carga entre los procesadores. La carga de procesamiento cuando el operador Sobel se ejecuta únicamente en el procesador ARM y cuando se ejecuta en el DSP es mostrado en la Tabla 6.2. Para el caso cuando el filtro se implementa en un único procesador, la cantidad de carga de trabajo que tiene el procesador ARM se debe a que este lleva a cabo operaciones aritméticas del filtro de imágenes de forma recursiva por cada pixel. El procesador ARM tiene una arquitectura que es más aprovechable para tareas de administración y supervisión de procesos y no para operaciones de procesamiento digital de imágenes.

Por otra parte, se ejecuta el filtro en el DSP y se usa al procesador ARM como administrador. Por medio de la función de Codec Engine *Server_getCpuLoad* se hace una solicitud información de carga al sistema operativo de tiempo real BIOS6. Este atiende la solicitud y transmite la información de carga a través de la interfaz. En este caso todo el algoritmo es llevado a cabo por el DSP. La aplicación cliente solo maneja la lectura y escritura de archivos y el control del flujo de ejecución.

6.5 Mapa de memoria resultante

A partir de la memoria del sistema y de la manipulación de memoria para ejecutar un proceso en el DSP C6748, se ilusta con la Figura 6.4 la actualización del mapa. La reducción del bloque asignado al sistema GNU/Linux embebido permite asignar una porción de esta partición al ejecutable del DSP. La ejecución de filtro de imágenes en el ambiente multiprocesador del SoC DM8168, se puede porque la aplicación de prueba opera bajo un esquema de memoria compartida, donde se tienen bloques definidos que no traslapan o invaden otros sectores de memoria del sistema.

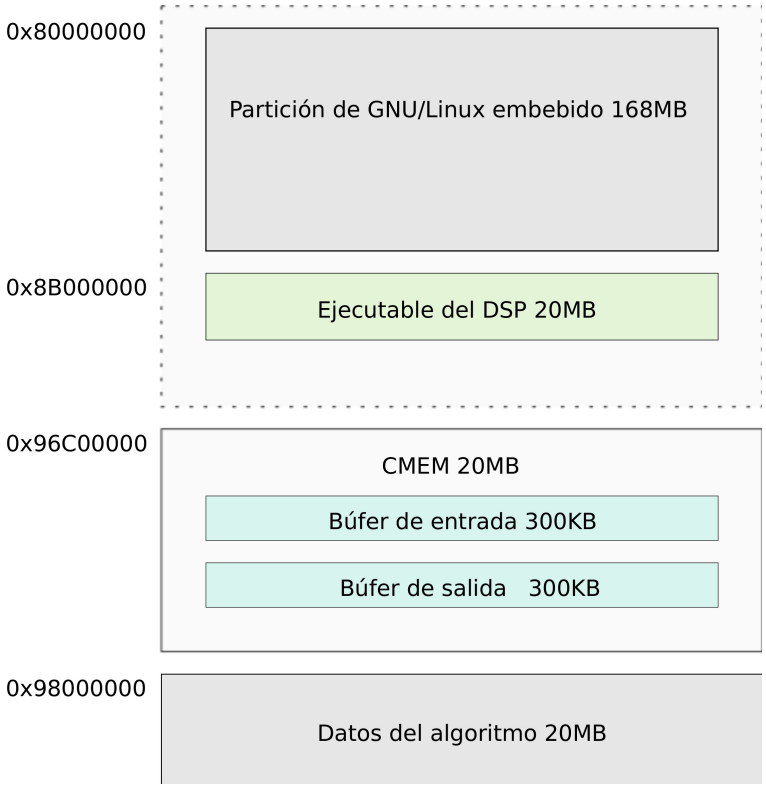


Figura 6.4: Mapa de memoria para aplicación demo ARM + DSP

Capítulo 7

Conclusiones

Los sistemas empotrados y el desarrollo de soluciones basadas en sistemas Linux, abarcan distintas consideraciones técnicas, desde la manipulación de un sistema de construcción para arquitecturas específicas, hasta técnicas de programación especiales para sistemas con limitaciones de memoria y potencia computacional.

El Kit de Desarrollo ofrece amplios recursos de software y utilidades de programación. Sin embargo las formas en las que se presentan estos componentes de software son tan amplias y a veces complejas que imposibilita el aprovechamiento de todos estos medios por parte de un desarrollador en un escenario práctico.

Las aplicaciones de procesamiento digital de señales tienden a ser de consumo alto en cuanto a carga de procesamiento, la implementación de dichas técnicas en un sistema empotrado aumentan el reto de su integración y su desempeño en un contexto de aplicaciones embebidas.

La elaboración del proyecto, involucra la investigación de tecnologiás de software que forman una base teórica que es necesaria para el diseño del marco de trabajo. Esto permite la incorporación de nuevas técnicas y recursos de programación al SDK del DM8168.

El uso de las interfaces de programación que manipulan a los RTSC en tiempo de ejecución, son métodos estándar que utilizan un lenguaje para abstraer tareas comunes. Las interfaces facilitan el uso de diferentes recursos del SoC bajo un mismo formato. Los Componentes de Software de Tiempo Real ofrecen contenido de software que permiten a los desarrolladores generar soluciones embebidas implementables en un ambiente multiprocesador como el que ofrece el SoC DM8168.

El marco de trabajo para la ejecución de procesos remotos utiliza medios y recursos del SDK que ofrecen una interfaz que habilita el desarrollo e implementación de Componentes de Tiempo Real, siempre apegado al sistema de integración de aplicaciones. El marco es un medio facilitador para formar una base de desarrollo de aplicaciones embebidas, este automatiza y acelera el proceso de construcción de paquetes de software para el DSP en un 73%, permitiendo a un desarrollador concentrarse en el diseño de algoritmos para

esta arquitectura sin la necesidad de invertir tiempo en el sistema de construcción de los RTSC.

Para un desarrollador que no tenga experiencia en el uso de los RTSC y ante la ausencia del marco de trabajo, a este le puede tomar aproximadamente 4 meses (que corresponde al tiempo disponible para realizar este proyecto) en construir e implementar un algoritmo para el DSP. Tomando de referencia una hora de trabajo ingeniero que corresponde a 85 dolares, para un plazo de 4 meses involucraría un costo aproximado de 54 mil dolares, este capital junto a las horas invertidas deben de ser cobradas a los clientes. Esto aumenta el precio de aplicaciones embebidas y dificulta la competividad de la empresa en el mercado de aplicaciones que utilicen técnicas de PDS.

Por otro lado, la utilización del marco de trabajo y la clase del sistema de integración del SDK puede reducir el tiempo de este proceso a 1 mes aproximadamente, considerando que el sistema de construcción es automatizado y el desarrollador dirige su tiempo de trabajo únicamente a la creación e implementación de algoritmos de PDS.

La implementación del filtro de imágenes utilizando la interfaz IUNIVERSAL es un punto de comienzo para que la empresa incursione en técnicas de procesamiento digital de imágenes más sofisticadas y complejas en el SoC DM8168. El operador Sobel necesita de una umbralización para detectar bordes de una manera más inclusiva o exclusiva. Este parámetro tiene que ser fijado de acuerdo a las particularidades de la aplicación que lo implemente, de manera que el filtro sea lo suficientemente sensible a los bordes que se buscan detectar.

Utilizar al DSP como procesador esclavo libera carga de trabajo del procesador ARM, esto permite a que este se dedique a otras tareas, incluyendo la delegación de más procesos a otras unidades esclavas. La reducción de la carga de procesamiento de 96% a 3% en el procesador ARM se debe a que la arquitectura se encarga únicamente de controlar el proceso remoto, mientras que el procesamiento de la imagen es realizado en su totalidad por el DSP.

Para que dos procesadores se transfieran datos, ambos deben de implementar correctamente las interfaces de programación para que exista una comunicación coherente entre ellos.

La arquitectura DM8168 tiene un mapa de memoria cuyas particiones cubren diferentes propósitos, las aplicaciones ARM + DSP deben operar bajo un esquema de memoria compartida del SoC, esto evita el traslape de sectores de memoria unos con otros que ocasionen fallas en tiempo de ejecución. Los controladores del sistema propician los medios necesarios para la conectividad entre procesadores y acceso a memoria, estos son utilizados por el programa de prueba para ejecutarse en un ambiente multiprocesador.

7.1 Recomendaciones

Los Componentes de Software de Tiempo Real además de ser utilizados para generar contenido ejecutable para el DSP C6748 del SoC DM8168, permiten generar código ejecutable para el Subsistema Ducati. El Subsistema Ducati usado por la empresa tiene el limitante de ser un firmware sin código fuente distribuido. Esto restringe la utilización de técnicas de codificación, decodificación, captura y despliegue de datos de imágenes y video. Por ejemplo, para implementar un codificador de imágenes JPEG en el Subsistema Ducati no se tiene un sistema de construccón integrado al SDK que genere código ejecutable para el módulo HDVICP2. Además se necesita de un codec JPEG, que implemente una interfaz xDM para que el procesador ARM (anfitrión) lo pueda invocar por medio del Controlador de Media. Esto involucra obstáculos técnicos en cuanto a manipulación y desarrollo de Componentes de Software de Tiempo Real necesarios para operar en un escenario de multiprocesamiento del SoC.

El marco de trabajo para la ejecución de procesos remotos fue diseñado en miras a cubrir esta carencia. El marco provee una base de software para generar contenido ejecutable para el Controlador de Media (procesadores ARM Cortex M3) y los módulos HDVICP2 y HDVPSS. A través del marco se pueden integrar nuevas técnicas de procesamiento de datos multimedia en estos módulos.

Con base en esto, el proyecto abre el espacio a que los desarrolladores expandan las capacidades del marco de trabajo. De esta manera, puede formar parte un entorno de desarrollo completo para sistemas embebidos que tengan unidades de procesamiento dedicadas. Existen otros sistemas (como el DM8148, OMAP3530, entre otros) que tienen un sistema de desarrollo basado en RTSC, de esta forma se abre la posibilidad de usar el marco de trabajo para otros productos de Texas Instruments que la empresa utiliza.

Otro punto importante, es que en el transcurso del proyecto se revelaron diferentes codecs y algoritmos que pueden ser usados por el marco de trabajo. Con esto se pueden incursionar en más técnicas de procesamiento de datos multimedia en el DSP C6748. Entre los cuales resaltan;

- Codecs que ofrece Texas Instruments

TI ofrece un decodificador AAC-LC de audio, este paquete está construido y listo para ser integrado al DSP C6748.

- IMGLIB

La biblioteca IMGLIB contempla funciones de análisis, de filtrado, y de compresión/decompresión de imagenes.

IMG_ycbcr422p_rgb565: Para conversión de contenido visual de formato Y‘CbCr hacia formato RGB.

IMG_fdct_8x8: Transformada discreta cosenoidal, también se encuentra disponible la transformada inversa.

IMG_wave_horz: Este es un wavelet que genera una descomposición periódica ortogonal en una dimensión. Esta puede ser utilizada para compresión de imagenes para el formato JPEG 2000, el cual es superior al formato JPEG convencional en cuanto a desempeño de compresión y otras métricas. Su complemento vertical tambien se encuentra en la biblioteca [43].

Estos son ejemplos de diferentes algoritmos que se podrían implementar en el DSP C6748 del SoC DM8168, muchos de hechos están hechos para el procesador de señales C64P. El C64P y C6748 pertenecen a la familia C6000 de procesadores digitales de señales, esto hace que algoritmos hechos para un procesador sean compatibles con otro de distinta arquitectura.

Bibliografía

[1] Sobel edge detector [online]. 2003 [visitado el 2 de mayo de 2013]. URL http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm.

[2] Command - xdc expanded c package build command [online]. 2008 [visitado el 3 de mayo de 2013]. URL http://rtsc.eclipse.org/docs-tip/Command_-_xdc.

[3] Command - xdc.tools.configuro [online]. 2008 [visitado el 12 de mayo de 2013]. URL http://rtsc.eclipse.org/docs-tip/Command_-_xdc.tools.configuro.

[4] Command - xs xdcscript interpreter [online]. 2008 [visitado el 16 de mayo de 2013]. URL http://rtsc.eclipse.org/docs-tip/Command_-_xs.

[5] Overview of rtsc - rtsc-pedia [online]. 2008 [visitado el 10 de abril de 2013]. URL http://rtsc.eclipse.org/docs-tip/Overview_of_RTSC.

[6] Sobel edge detection [online]. 2009 [visitado el 2 de mayo de 2013]. URL http://www.dewtell.com/code/cpp/sobel.htm.

[7] Xdais download page [online]. 2009 [visitado el 5 de febrero de 2013]. URL http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/xdais/index.html.

[8] Codec engine application programming interface (api): Universal algorithm interface [online]. 2010 [visitado el 23 de mayo de 2013]. URL http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/ce/2_26_00_08/exports/codec_engine_2_26_00_08/docs/html/group__ti_sdo__ce__universal___u_n_i_v_e_r_s_a_l.html.

[9] Gnu make [online]. 2010 [visitado el 4 de marzo de 2013]. URL http://www.gnu.org/software/make/manual/make.html.

[10] Hdfpga: Ti netra is out - dm8168 [online]. 2010 [visitado el 20 de enero de 2013]. URL http://hdfpga.blogspot.com/2010/04/ti-netra-dm8168.html.

[11] Overview of an embedded system [online]. 2010 [visitado el 26 de enero de 2013]. URL http://embedded-lab.com/blog/?p=949.

[12] Linux utils application programming interface (api): cmem.h file reference [online]. 2011 [visitado el 14 de mayo de 2013]. URL http://software-dl.ti.com/dsps/dsps_public_sw/sdo_sb/targetcontent/linuxutils/2_26_03_06/exports/linuxutils_2_26_03_06/docs/html/cmem_8h.html.

[13] The scientist and engineer's guide to digital signal processing [online]. 2011 [visitado el 18 de enero de 2013]. URL http://www.dspguide.com/.

[14] Understanding client-server applications – part i - developer zone - national instruments [online]. 2011 [visitado el 22 de marzo de 2013]. URL http://www.ni.com/white-paper/4431/en.

[15] What is digital signal processing [online]. 2011 [visitado el 20 de enero de 2013]. URL http://www.dspguide.com/whatdsp.htm.

[16] Ezsdk memory map [online]. 2012 [visitado el 2 de mayo de 2013]. URL http://processors.wiki.ti.com/index.php/EZSDK-Memory-Map.

[17] Memory layout and memory map [online]. 2012 [visitado el 2 de mayo de 2013]. URL http://staff.ustc.edu.cn/~xyfeng/research/cos/resources/machine/mem.htm.

[18] Ridgerun 2011q2 sdk development and integration guide - ridgerun developer connection [online]. 2012 [visitado el 27 de abril de 2013]. URL https://developer.ridgerun.com/wiki/index.php/RidgeRun_2011Q2_SDK_Development_and_Integration_Guide#Configuration_system.

[19] Ti introduces new davinci digital media processor [online]. 2012 [visitado el 11 de febrero de 2013]. URL http://www.32bitmicro.com/165-manufacturers/ti.

[20] Buffer [online]. 2013 [visitado el 2 de junio de 2013]. URL http://www.techterms.com/definition/buffer.

[21] C6ezaccel software development tool for ti dsp+arm processors - c6accel-dsplibs - ti software folder (obsolete) [online]. 2013 [visitado el 10 de enero de 2013]. URL http://www.ti.com/tool/c6accel-dsplibs.

[22] Category:syslink [online]. 2013 [visitado el 4 de marzo de 2013]. URL http://processors.wiki.ti.com/index.php/Category:SysLink.

[23] Codec engine [online]. 2013 [visitado el 4 de marzo de 2013]. URL http://processors.wiki.ti.com/index.php/Category:Codec_Engine.

[24] Cross-compiling qt for embedded linux applications [online]. 2013 [visitado el 24 de abril de 2013]. URL http://qt-project.org/doc/qt-4.8/qt-embedded-crosscompiling.html.

[25] Dsp + arm cortex-a8 - davinci dm81x soc - tms320dm8168 - ti.com [online]. 2013 [visitado el 5 de enero de 2013]. URL http://www.ti.com/product/tms320dm8168.

[26] The eclipse foundation [online]. 2013 [visitado el 8 de abril de 2013]. URL http://www.eclipse.org/rtsc/.

[27] gstreamer - open source multimedia framework [online]. 2013 [visitado el 14 de febrero de 2013]. URL http://gstreamer.freedesktop.org/.

[28] How autotools help [online]. 2013 [visitado el 2 de mayo de 2013]. URL http://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node/Why-Autotools.html#Why-Autotools.

[29] Luca scassa [online]. 2013 [visitado el 11 de abril de 2013]. URL http://www.motorcycle-usa.com/553/Motorcycles/Luca-Scassa.aspx.

[30] Quickstudy: Application programming interface [online]. 2013 [visitado el 6 de mayo de 2013]. URL http://www.computerworld.com/s/article/43487/Application_Programming_Interface.

[31] Real-time software components (rtsc) [online]. 2013 [visitado el 8 de abril de 2013]. URL http://www.eclipse.org.

[32] Ridgerun 2011q2 sdk user guide - ridgerun developer connection [online]. 2013 [visitado el 15 de marzo de 2013]. URL https://developer.ridgerun.com/wiki/index.php/RidgeRun-2011Q2-SDK-UserGuide.

[33] Ridgerun Embedded Solutions [online]. 2013 [visitado el 9 de enero de 2013]. URL http://www.ridgerun.com.

[34] Sourcery codebench [online]. 2013 [visitado el 11 de marzo de 2013]. URL http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview.

[35] Spectrum digital inc. dm816x/c6a816x/am389x standalone evm baseboard [online]. 2013 [visitado el 18 de febrero de 2013]. URL http://www.spectrumdigital.com/product-info.php?products-id=253.

[36] Sys/bios real-time kernel [online]. 2013 [visitado el 4 de abril de 2013]. URL http://www.ti.com/tool/sysbios.

[37] Tms320c6000 image library (imglib) [online]. 2013 [visitado el 21 de abril de 2013]. URL http://www.ti.com/tool/sprc264.

[38] J. Coombs and R. Prabhu. Opencv on ti's dsp+arm® platforms: Mitigating the challenges of porting opencv to embedded platforms. 2011.

[39] V. Gite. Linux shell scripting tutorial v1.05 chapter 1 what is linux shell ? [online]. 2002 [visitado el 21 de febrero de 2013]. URL http://www.freeos.com/guides/lsst/ch01sec07.html.

[40] Texas Instruments. Codec engine algorithm creator user's guide. 2007.

[41] Texas Instruments. Codec engine server integrators user's guide. 2007.

[42] Texas Instruments. Xdm user guide. 2007.

[43] Texas Instruments. Tms320c64x+ dsp image/video processing library (v2.0.1). 2008.

[44] Texas Instruments. Tms320dm816x davinci video processors (no. revd). 2011.

[45] Texas Instruments. Tms320c6000 optimizing compiler v7.4 user's guide. 2012.

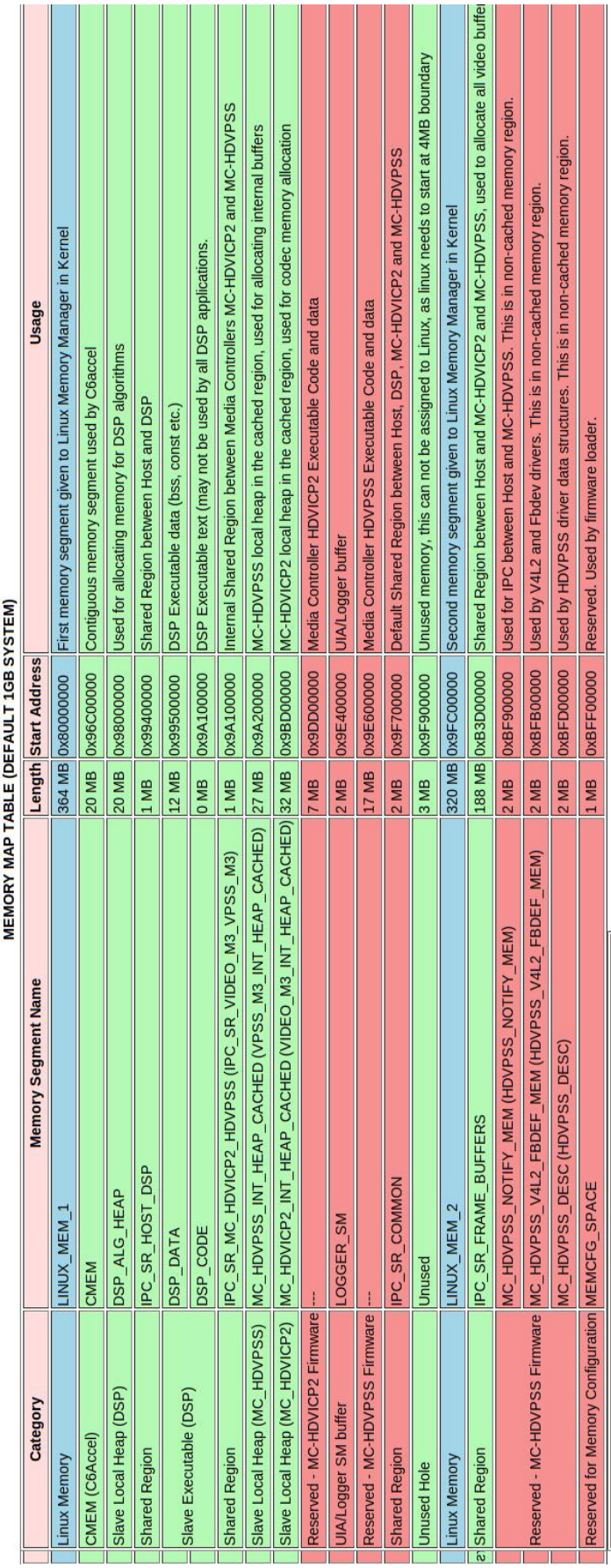
[46] Texas Instruments. Os abstraction layer application programming interface. 2013.

[47] E. Petrakis. Edge detection : intelligence - intelligent system laboratory. 2013.

[48] R. Stallman. Linux and the gnu system [online]. 2007 [visitado el 21 de febrero de 2013]. URL http://www.gnu.org/gnu/linux-and-gnu.html.

Apéndice A

Mapa de memoria del SDK para el DM8168



MEMORY MAP TABLE (DEFAULT 1GB SYSTEM)

Category	Memory Segment Name	Length	Start Address	Usage
Linux Memory	LINUX_MEM_1	364 MB	0x80000000	First memory segment given to Linux Memory Manager in Kernel
CMEM (C6Accel)	CMEM	20 MB	0x96C00000	Contiguous memory segment used by C6accel
Slave Local Heap (DSP)	DSP_ALG_HEAP	20 MB	0x98000000	Used for allocating memory for DSP algorithms
Shared Region	IPC_SR_HOST_DSP	1 MB	0x99400000	Shared Region between Host and DSP
Slave Executable (DSP)	DSP_DATA	12 MB	0x99500000	DSP Executable data (bss, const etc.)
	DSP_CODE	0 MB	0x9A100000	DSP Executable text (may not be used by all DSP applications.
Shared Region	IPC_SR_MC_HDVICP2_HDVPSS (IPC_SR_VIDEO_M3_VPSS_M3)	1 MB	0x9A100000	Internal Shared Region between Media Controllers MC-HDVICP2 and MC-HDVPSS
Slave Local Heap (MC_HDVPSS)	MC_HDVPSS_INT_HEAP_CACHED (VPSS_M3_INT_HEAP_CACHED)	27 MB	0x9A200000	MC-HDVPSS local heap in the cached region, used for allocating internal buffers
Slave Local Heap (MC_HDVICP2)	MC_HDVICP2_INT_HEAP_CACHED (VIDEO_M3_INT_HEAP_CACHED)	32 MB	0x9BD00000	MC-HDVICP2 local heap in the cached region, used for codec memory allocation
Reserved - MC-HDVICP2 Firmware	---	7 MB	0x9DD00000	Media Controller HDVICP2 Executable Code and data
UIA/Logger SM buffer	LOGGER_SM	2 MB	0x9E400000	UIA/Logger buffer
Reserved - MC-HDVPSS Firmware	---	17 MB	0x9E600000	Media Controller HDVPSS Executable Code and data
Shared Region	IPC_SR_COMMON	2 MB	0x9F700000	Default Shared Region between Host, DSP, MC-HDVICP2 and MC-HDVPSS
Unused Hole	Unused	3 MB	0x9F900000	Unused memory, this can not be assigned to Linux, as linux needs to start at 4MB boundary
Linux Memory	LINUX_MEM_2	320 MB	0x9FC00000	Second memory segment given to Linux Memory Manager in Kernel
Shared Region	IPC_SR_FRAME_BUFFERS	188 MB	0xB3D00000	Shared Region between Host and MC-HDVICP2 and MC-HDVPSS, used to allocate all video buffe
Reserved - MC-HDVPSS Firmware	MC_HDVPSS_NOTIFY_MEM (HDVPSS_NOTIFY_MEM)	2 MB	0xBF900000	Used for IPC between Host and MC-HDVPSS. This is in non-cached memory region.
	MC_HDVPSS_V4L2_FBDEF_MEM (HDVPSS_V4L2_FBDEF_MEM)	2 MB	0xBFB00000	Used by V4L2 and Fbdev drivers. This is in non-cached memory region.
	MC_HDVPSS_DESC (HDVPSS_DESC)	2 MB	0xBFD00000	Used by HDVPSS driver data structures. This is in non-cached memory region.
Reserved for Memory Configuration	MEMCFG_SPACE	1 MB	0xBFF00000	Reserved. Used by firmware loader.

Color Code
Linux Memory
Memory used for regions that can be adjusted without rebuilding Media Controller executable.
Memory Reserved for Media Controller Executables

Figura A.1: Mapa de memoria completo del SDK [16]

Apéndice B

Tuberias de GStreamer

Estas tuberias permiten la adquisición y despliegue de imagenes,

- Tuberia que convierte imagen de formato JPEG a formato en crudo en escala de grises

```
gst-launch filesrc location= capture/scassa.jpg ! jpegdec ! ffmpegcolorspace !
'video/x-raw-gray, bpp=8, endianness=4321, width=(int)640, height=(int)480' !
filesink location = capture/scassa\_gray8.raw -v
```

- Tuberia que convierte imagen de formato en crudo a formato JPEG

```
gst-launch filesrc location = capture/scassa\_gray8.raw blocksize= 307200 !
'video/x-raw-gray, bpp=8, width=(int)640, height=(int)480, framerate=
(fraction)0/1, endianness=4321' ! ffmpegcolorspace ! jpegenc ! filesink
location= capture/scassa\_gray8.jpg -v
```