DESARROLLO DE UN LENGUAJE DE PROGRAMACIÓN GRÁFICO PARA MICROCONTROLADORES

Desarollo de un Lenguaje de Programación Gráfico para Microcontroladores

Klever David Cajamarca Sacta
Egresado de la Carrera de Ingeniería Electrónica
Facultad de Ingenerías
Universidad Politécnica Salesiana

Dirigido Por:

Ing. Eduardo Calle Ortiz Msc.
Ingeniero Electrónico
Docente de la Universidad Politécnica Salesiana
Facultad de Ingenierías

Asesorado Por:

Ing. Mauricio Ortiz
Ingeniero de Sistemas
Docente de la Universidad Politécnica Salesiana
Facultad de Ingenierías

Cuenca - Ecuador

Datos de catalogación bibliográfica

CAJAMARCA SACTA KLEVER DAVID

Desarrollo de un Lenguaje de Programación Gráfico para Microcontroladores

Universidad Politécnica Salesiana Cuenca - Ecuador, 2011

INGENIERÍA ELECTRÓNICA

Formato: 170x240mm Páginas: 124

Kléver David Cajamarca Sacta.
Egresado de la Carrera de Ingeniería Electrónica.
Facultad de Ingenierías.
Universidad Politécnica Salesiana.
davidcajamarca@hotmail.com

Eduardo Calle Ortiz.
Ingeniero Electrónico.
Master en Tecnologías de la Información.
Especialista en Robótica.
Docente de la Universidad Politécnica Salesiana.
Facultad de Ingenierías.
ecalle@ups.edu.ec

Edición y Producción
Kléver David Cajamarca Sacta

Diseño de la Portada
Eduardo Calle Ortiz

IMPRESO EN ECUADOR - PRINTED IN ECUADOR

Certifico

que la tesis intitulada "Desarrollo de un Lenguaje de Programación Gráfico para Microcontroladores" realizada por el señor Klever David Cajamarca Sacta ha sido realizada bajo mi dirección.

Ing. Eduardo Calle Ortiz MSc.

Certifico

que la tesis intitulada "Desarrollo de un Lenguaje de Programación Gráfico para Microcontroladores" realizada por el señor Klever David Cajamarca Sacta ha sido realizada bajo mi asesoría.

Ing. Mauricio Ortiz

Certifico

que la tesis intitulada "Desarrollo de un Lenguaje de Programación Gráfico para Microcontroladores" previa a la obtención del Título de Ingeniero Electrónico es de mi autoría.

Klever David Cajamarca Sacta

Agradecimientos

Este proyecto ha sido dirigido por el Ing. Eduardo Calle Ortiz, a quien expreso mi más profundo agradecimiento por el apoyo brindado, por sus ideas y conocimiento transmitidos, por la confianza y la paciencia en el desarrollo del sistema.

También agradezco a las personas que han contribuido a la finalización del proyecto, al Ing. Mauricio Ortiz por su asesoría, y a mis padres por la libertad y confianza brindadas.

Índice general

Índice de figuras

Índice de cuadros

Índice de algoritmos

Capítulo 1

CONCEPTOS Y PLANIFICACIÓN DE SOFTWARE EN INGENIERÍA

En el presente capítulo se exponen conceptos fundamentales de algoritmos y programación en torno a los cuales se desarrolla el sistema de programación gráfico implementado. Esta fundamentación permitirá que la herramienta de software desarrollada cumpla con principios fundamentales dentro de la ingeniería y las matemáticas, para que de este modo pueda ser continuada y expandida en su funcionalidad y pueda ser portada a nuevos sistemas electrónicos (microcontroladores, sistemas microprocesados, etc) o computacionales (nuevos sistemas operativos, arquitecturas de hardware, etc).

1.1. Conceptos Fundamentales de Programación

En esta sección se analizarán los conceptos fundamentales de programación, tomando como referencia la programación estructurada, ya que este es el paradigma que se aplica en la programación de microcontroladores. Además se hace una exposición de los paradigmas de programación mas importantes, cuyos conceptos han contribuido al desarrollo del sistema de programación gráfica planteado.

1.1.1. Maquinas de Estado Finito

En primer lugar analicemos un sistema de control tradicional, un sistema de control recibe un número de estímulos (Entradas) las cuales se procesan en el sistema de control produciendo acciones (Salidas), las cuales afectan a la aplicación. Este modelo es suficiente para sistemas simples, aplicaciones mas realistas requieren un modelo de control mucho más sofisticado.

Uno de los modelos mas potentes es el de Máquina de Estado Finito, el cual se utiliza para describir el comportamiento del sistema en todas las situaciones posibles. En resumen el concepto de máquina de estado finito es un modelo de sistema sofisticado, que amplia el modelo de sistema de control tradicional.

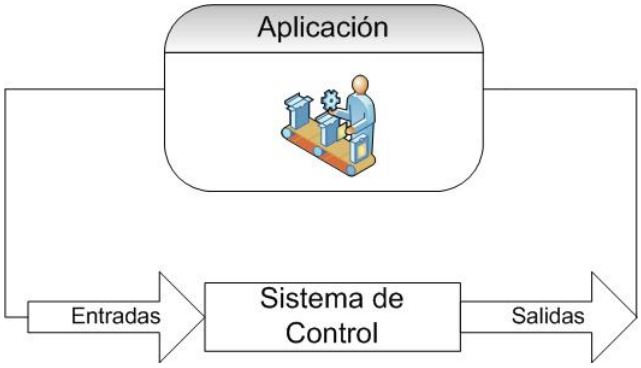

Figura 1.1.1: Sistema de Control Tradicional

La máquina de estado finito introduce el concepto de estado como información sobre su estado pasado. Todos los estados representan todas las posibles situaciones en los que la máquina de estado puede estar. Entonces, contiene un tipo de memoria: la secuencia de procesos para que la máquina de estado pueda haber alcanzado la situación presente. A medida que la aplicación se ejecuta, el estado cambia de tiempo en tiempo, y las salidas pueden depender del estado actual, así como de las entradas. Debido a que el número de situaciones para una máquina de estado dada es finito, el número de estados es finito, debido a esto se le conoce como máquina de estado finito[1].

A modo de resumen podría decirse que una máquina de estado finito es un concepto matemático que permite representar un sistema (consistente de un proceso, entradas y salidas) de una forma generalizada y abstracta. Al concepto de máquina de estado finito también se le conoce con el nombre de automata finito.

Un sistema de control tradicional determina sus salidas dependiendo de las entradas, si el valor presente de las entradas es suficiente para determinar las salidas el sistema es llamado “sistema combinacional"[2], y no necesita el concepto de estado. Si el sistema de control requiere información adicional sobre la secuencia de cambios en la entrada para determinar la salida, el sistema es llamado “sistema secuencial". Los sistemas de control tradicional pueden considerarse como un caso especial de una máquina de estado finito, con un solo estado posible, este estado único se aplica en todos los casos.

[1] Dado que para el presente estudio solo se requieren de máquinas de estado finito, se les llamará simplemente maquinas de estado.

[2] Este nombre hace referencia al hecho de que se basa en la lógica combinacional.

Una máquina de estado puede representarse mediante el diagrama de la figura 1.1.2, el conjunto de cambios en la entrada requerido para determinar el comportamiento de la máquina de estado se almacena en la variable Estado. Tanto las Condiciones de Transición de Estado como las Condiciones de Salida son funciones de las Entradas y de un Estado.

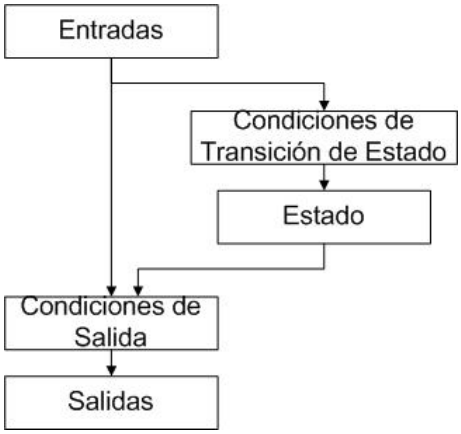

Figura 1.1.2: Concepto de Máquina de Estado Finito

El concepto de máquina de estado finito tiene una definición matemática explícita, que puede aplicarse de manera genérica en todos los ámbitos en los cuales se puede aplicar las máquinas de estado finito, esta definición se presenta a continuación:

Definición 1. Una máquina de estado finito determinista M esta compuesto de los siguientes elementos[3]:

- Un conjunto de símbolos de entrada, conocido como alfabeto (X).

- Un conjunto finito de estados (Q).

- Un estado o condición inicial (q_0).

- Una función de transición (δ).

- Un conjunto de estados finales (F).

El comportamiento de la máquina de estado finito se describe como una secuencia de eventos que ocurren en instantes discretos de tiempo de la forma: $t = 1, 2, 3, \ldots$.

[3]En el capítulo 3 se expondrá la aplicación de las maquinas de estado finito a la programación gráfica, analizando las herramientas para describirlas, tales como el Diagrama de Transición de Estados y la Matriz de Transición.

1.1.2. Condicionales y Bucles

Instrucciones Condicionales

En el modelo de ejecución de los programas, las instrucciones se ejecutan en secuencia, una después de otra, en el mismo orden en que aparecen en el código del programa. Esta ejecución puramente secuencial no permitiría realizar programas muy complejos, porque siempre se realizarían las mismas operaciones.

Por este motivo surgieron las instrucciones condicionales, cuyo objetivo es permitir controlar el flujo de ejecución del programa. Las instrucciones condicionales se resumen en la siguiente tabla:

Instrucción	*Significado*

`if(condición)`
`  instrucción_si;`
`else`
`  instrucción_no;`
	La condición tiene que ser una expresión cuya evaluación dé como resultado un dato de tipo compatible con entero. Si el resultado es distinto de cero, se considera que la condición se cumple y se ejecuta instrucción_si. En caso contrario, se ejecuta instrucción_no. La utilización de la instrucción else es opcional.

`switch(expresión)`
`{`
`  case valor_1 :`
`  instrucciones;`
`  break;`
`  case valor_2 :`
`  instrucciones;`
`  break;`
`  default :`
`  instrucciones;`
`  break;`
`}`
	La evaluación de la expresión debe resultar en un dato compatible con entero. Este resultado se compara con los valores indicados en cada case y, de ser igual a alguno de ellos, se ejecutan todas las instrucciones a partir de la primera indicada en ese caso y hasta el final del bloque del switch. Es posible "romper" esta secuencia introduciendo una instrucción break; que finaliza la ejecución de la secuencia de instrucciones. Opcionalmente, es posible indicar un caso por omisión (default) que permite especificar qué instrucciones se ejecutaran si el resultado de la expresión no ha producido ningún dato coincidente con los casos previstos.

Cuadro 1.1: Instrucciones Condicionales

En el caso de la instrucción **if** es posible ejecutar más de una instrucción, tanto si la condición se cumple como si no, agrupando las instrucciones en un bloque. Los bloques de instrucciones son instrucciones agrupadas entre llaves:

```
if(condición)
{
  instrucción_1;
  instrucción_2;
  instrucción_3;
    .
    .
    .
  instrucción_n;
}
```

Instrucciones Iterativas (bucles):

Las instrucciones iterativas son instrucciones de control de flujo que permiten repetir la ejecución de un bloque de instrucciones. En la siguiente tabla se muestran las instrucciones iterativas definidas:

Instrucción	*Significado*

`while(condición)`
`{`
`  instrucciones;`
`}`
	Se ejecutan todas las instrucciones en el bloque del bucle mientras la expresión de la condición dé como resultado un dato de tipo compatible con entero distinto de cero; es decir, mientras la condición se cumpla. Las instrucciones pueden no ejecutarse nunca.

`do`
`{`
`  instrucciones`
`}`
`while(condición);`
	De forma similar al bucle while, se ejecutan todas las instrucciones en el bloque del bucle mientras la expresión de la condición se cumpla. La diferencia estriba en que las instrucciones se ejecutarán, al menos, una vez. (La comprobación de la condición y, por tanto, de la posible repetición de las instrucciones, se realiza al final del bloque.)

`for(inicialización`
`;`
`condición;`
`continuación)`
`{`
`  instrucciones;`
`}`
	El comportamiento es parecido a un bucle while; es decir, mientras se cumpla la condición se ejecutan las instrucciones de su bloque. En este caso, sin embargo, es posible indicar qué instrucción o instrucciones quieren ejecutarse de forma previa al inicio del bucle (inicialización) y qué instrucción o instrucciones hay que ejecutar cada vez que finaliza la ejecución de las instrucciones (continuación).

Cuadro 1.2: Instrucciones Iterativas

Como se puede apreciar, todos los bucles pueden reducirse a un bucle "mientras". Aun así, hay casos en los que resulta más lógico emplear alguna de sus variaciones.

Hay que tener presente que la estructura del flujo de control de un programa en un lenguaje de alto nivel no refleja lo que realmente hace el procesador (saltos condicionales e incondicionales) en el aspecto del control del flujo de la ejecución de un programa. Aun así, el lenguaje C dispone de instrucciones que nos acercan a la realidad de la máquina, como la de salida forzada de bucle (break;) y la de la continuación forzada de bucle (continue;).

Normalmente, la programación de un bucle implica determinar cuál es el bloque de instrucciones que hay que repetir y, sobre todo, bajo qué condiciones hay que realizar su ejecución. En este sentido, es muy importante tener presente que la condición que gobierna un bucle es la que determina la validez de la repetición y, especialmente, su finalización cuando no se cumple. Nótese que debe existir algún caso en el que la evaluación de la expresión de la condición dé como resultado un valor 'falso'. En caso contrario, el bucle se repetiría indefinidamente (esto es lo que se llamaría un caso de "bucle infinito").

Habiendo determinado el bloque iterativo y la condición que lo gobierna, también cabe programar la posible preparación del entorno antes del bucle y las instrucciones que sean necesarias a su conclusión: su inicialización y su finalización. En resumen, la instrucción iterativa debe escogerse en función de la condición que gobierna el bucle y de su posible inicialización.

Es recomendable evitar el empleo del for para los casos en que no haya contadores. En su lugar, es recomendable emplear un bucle while.

En algunos casos, gran parte de la inicialización coincidiría con el cuerpo del bucle, o bien se hace necesario evidenciar que el bucle se ejecutará al menos una vez. Si esto se da, es conveniente utilizar una estructura do...while.

1.1.3. Funciones

La lectura del código fuente de un programa implica realizar el seguimiento del flujo de ejecución de sus instrucciones (el flujo de control). Evidentemente, una ejecución en el orden secuencial de las instrucciones no precisa de mucha atención. Pero los programas por lo general contienen también instrucciones condicionales e iterativas. Por ello, el seguimiento del flujo de control puede resultar complejo si el código fuente ocupa más de lo que se puede observar.

En consecuencia, resulta conveniente agrupar aquellas partes del código que realizan una función muy concreta en un subprograma identificado de forma individual. Además se tiene la característica de que estas agrupaciones de código se puede utilizan en diversos momentos de la ejecución de un programa, lo que implica ventajas en el desarrollo de programas.

A las agrupaciones de código en las que se divide un determinado programa se las llama funciones. En programación estructurada, en el caso específico del lenguaje C, todo el código debe estar distribuido en funciones y, de hecho, el propio programa principal es una función: la función principal (main). Generalmente, una función incluirá en su código, la programación de unos pocos esquemas algorítmicos de procesamiento de secuencias de datos y algunas ejecuciones condicionales o alternativas. Es decir, lo necesario para realizar una tarea concreta[4].

Declaración y Definición

La declaración de cualquier entidad (variable o función) implica la manifestación de su existencia al compilador, mientras que definirla supone describir su contenido. La declaración consiste exactamente en lo mismo que para las variables: manifestar su existencia. En este caso, de todas maneras hay que describir los argumentos que toma y el resultado que devuelve para que el compilador pueda generar el código, y poderlas emplear.

La declaración consiste exactamente en lo mismo que para las variables: manifestar su existencia. En este caso, de todas maneras hay que describir los argumentos que toma y el resultado que devuelve para que el compilador pueda generar el código, y poderlas emplear. En cambio, la definición de una función se corresponde con su programa, que es su contenido. De hecho, de forma similar a las variables, el contenido se puede identificar por la posición del primero de sus bytes en la memoria principal. Este primer byte es el primero de la primera instrucción que se ejecuta para llevar a cabo la tarea que tenga programada.

Sintaxis de la Declaración de Funciones:

La declaración de una función consiste en especificar el tipo de dato que devuelve, el nombre de la función, la lista de parámetros que recibe entre paréntesis y un punto y coma que finaliza la declaración:

```
tipo_de_dato nombre_función(parámetros);
```

Hay que tener presente que no se puede hacer referencia a funciones que no estén declaradas previamente. Por este motivo, es necesario incluir los ficheros de cabeceras de las funciones estándar de la biblioteca de C como stdio.h, por ejemplo. En

[4]Cabe indicar que la definición de funcion en programación tiene similitudes con la nocion de función matemática, ya que para cierta entrada se espera un salida generada a partir de un proceso. Definiendo y poniendo limites al comportamiento de una funcion en programación se puede tratar como una entidad matemática, lo que permite que se pueda manipular como tal, y se pueda aplicar y utilizar todas las herramientas que componen el análisis de funciones en matemáticas. Claro ejemplo de esto es la noción de ortogonalidad que se analizará en la siguiente seccion.

el lenguaje de programación C, si una función no ha sido previamente declarada, el compilador supondrá que devuelve un entero. De la misma manera, si se omite el tipo de dato que retorna, supondrá que es un entero.

Definición de una Función

La definición de una función está encabezada siempre por su declaración, la cual debe incluir forzosamente la lista de parámetros si los tiene. Esta cabecera no debe finalizar con punto y coma, sino que irá seguida del cuerpo de la función, delimitada entre llaves de apertura y cierre:

Algoritmo 1.1 Definición de una Función en un Lenguaje Estructurado

```
tipo_de_dato nombre_función(parámetros)
{ /* Cuerpo de la Función: */
    - Declaración de Variables Locales
    - Instrucciones de la Función
    if(tipo_de_dato!=void)
    {
        - Retorno de Valores Resultado de la Función (si se
            espera un resultado de la función)
    }
    else
    {
        - Retorno
    }
}
```

Ortogonalidad

La ortogonalidad de funciones significa que las funciones se pueden combinar, y que esta combinación tendrá sentido, es decir:

$$función1(función2(variable)) = resultado$$

Para que se pueda aplicar la ortogonalidad se debe cumplira la condición de que el tipo de dato que produce la función2 sea igual al tipo de dato de entrada de la función 1. El tipo de dato del resultado será el tipo de dato de salida de la función1.

El siguiente ejemplo muestra en la práctica el uso de la propiedad de ortogonalidad de las funciones:

Algoritmo 1.2 Aplicación de la ortogonalidad de funciones

```
/* Declaracion de Funciones */
int función_suma(int num1, int num2);
float función_división(int numerador, int denominador);


/* Programa Principal */
float resultado=función_división(función_suma(7,23),4),


/* Resultado de la Ejecución */
resultado="7.5";
```

1.1.4. Tipos y Estructuras Básicas de Datos

Los tipos de datos basicos de un lenguaje son aquéllos cuyo tratamiento se puede realizar con las instrucciones del mismo lenguaje; es decir, que están soportados por el lenguaje de programación correspondiente.

Datos de tipo Entero

Los tipos de datos básicos más comunes son los compatibles con enteros. La representación binaria de éstos no es codificada, sino que se corresponde con el valor numérico representado en base 2. Por tanto, se puede calcular su valor numérico en base 10 sumando los productos de los valores intrínsecos (0 o 1) de sus dígitos (bits) por sus valores posicionales ($2^{posicion} - 1$) correspondientes.

Se tratan bien como números naturales, o bien como representaciones de enteros en base 2, si pueden ser negativos. En este último caso, el bit más significativo (el de más a la izquierda) es siempre un 1 y el valor absoluto se obtiene restando el número natural representado del valor máximo representable con el mismo número de bits más 1.

Es importante tener presente que el rango de valores de estos datos depende del número de bits que se emplee para su representación, los datos de tipo entero se presentan en la siguiente tabla:

Designación	***Número de bits***	***Rango de valores***
signed char	8 (1 byte)	-128 a +127
unsigned char	8 (1byte)	0 a 255
signed int	16 (2 byte)	- 32768 a +32767
unsigned int	16 (2 byte)	0 a 65535
signed long	32 (4 byte)	-2147483648 a +2147482647
unsigned long	32 (4 byte)	0 a 4294967295

Cuadro 1.3: Datos de Tipo Entero

Hay que recordar especialmente los distintos rangos de valores que pueden tomar las variables de cada tipo para su correcto uso en los programas. De esta manera, es posible ajustar su tamaño al que realmente sea útil.

El tipo carácter (char) es un entero que identifica una posición de la tabla de caracteres ASCII. Para evitar tener que traducir los caracteres a números, éstos se pueden introducir entre comillas simples (por ejemplo: 'A'). También es posible representar códigos no visibles como el salto de línea ('\n') o la tabulación ('\t').

Datos de Punto Flotante

Este tipo de datos es más complejo que el anterior, pues su representación binaria se encuentra codificada en distintos campos. Así pues, no se corresponde con el valor del número que se podría extraer de los bits que los forman.

Los datos de punto flotante se representan mediante signo, mantisa y exponente; esto se corresponde con el hecho de que un dato de punto flotante representa un numero real[5]. La mantisa expresa la parte fraccionaria del número y el exponente es el número al que se eleva la base correspondiente:

$$numero = [signo]mantisa \cdot base^{exponente}$$

En función del número de bits que se utilicen para representarlos, los valores de la mantisa y del exponente serán mayores o menores. Los distintos tipos de datos de punto flotante y sus rangos aproximados se muestran en la siguiente tabla:

[5]Un numero real es cualquier numero que tenga representación decimal, el conjunto de los numeros reales abarca a los numeros racionales (que tienen representacion fraccionaria) como a los irracionales (que no tienen representación fraccionaria y tienen infinitas cifras decimales no periódicas).

Designación	*Numero de bits*	*Rango de valores*
float	32 (4 bytes)	$\pm 3,4 \cdot 10^{\pm 38}$
double	64 (8 bytes)	$\pm 1,4 \cdot 10^{\pm 308}$
long double	96 (12 bytes)	$\pm 1,1 \cdot 10^{\pm 4932}$

Cuadro 1.4: Datos de punto flotante

Como se puede deducir de la tabla anterior, es importante ajustar el tipo de datos real al rango de valores que podrá adquirir una determinada variable para no ocupar memoria innecesariamente. También cabe prever lo contrario: el uso de un tipo de datos que no pueda alcanzar la representación de los valores extremos del rango empleado provocará que éstos no se representen adecuadamente y, como consecuencia, el programa correspondiente puede comportarse de forma errática.

1.2. Paradigmas de Programación

1.2.1. Programación Imperativa o de Procedimientos

En el paradigma de la programación imperativa, las instrucciones son órdenes que se llevan a cabo de forma inmediata para conseguir algún cambio en el estado del procesador y, en particular, para el almacenamiento de los resultados de los cálculos realizados en la ejecución de las instrucciones. Ejemplos de lenguajes de programación imperativa son: Basic, Fortran, Cobol[6].

La mayoría de algoritmos en programacion imperativa consisten en una secuencia de pasos que indican lo que hay que hacer. Estas instrucciones suelen ser de carácter imperativo, es decir, indican lo que hay que hacer de forma incondicional. En este tipo de programas, cada instrucción implica realizar una determinada acción sobre su entorno, en este caso, en el computador en el que se ejecuta.

Para entender cómo se ejecuta una instrucción, en el paradigma de programacion imperativa es necesario ver cómo es el entorno en el que se lleva a cabo. La mayoría de los procesadores se organizan de manera que los datos y las instrucciones se encuentran en la memoria principal y la unidad central de procesamiento (CPU) es la que realiza el siguiente algoritmo para poder ejecutar el programa en memoria:

1. Leer de la memoria la instrucción que hay que ejecutar.
2. Leer de la memoria los datos necesarios para su ejecución.

[6] Estos programas en sus versiones originales constituyen claros ejemplos de programacion imperativa, versiones actualizadas de estos lenguajes de programacion incluyen funcionalidades que permiten enmarcarlas en otros paradigamas como programacion estructurada, funcional u orientada a objetos.

3. Realizar el cálculo u operación indicada en la instrucción y, según la operación que se realice, grabar el resultado en la memoria.

4. Determinar cuál es la siguiente instrucción que hay que ejecutar.

5. Volver al primer paso.

La CPU hace referencia a las instrucciones y a los datos que pide a la memoria o a los resultados que quiere escribir mediante el número de posición que ocupan en la misma. Esta posición que ocupan los datos y las instrucciones se conoce como dirección de memoria. En el nivel más bajo, cada dirección distinta de memoria es un único byte y los datos y las instrucciones se identifican por la dirección del primero de sus bytes. En este nivel, la CPU coincide con la CPU física de que dispone el computador.

En los lenguajes de programacion imperativa se mantiene el hecho de que las referencias a los datos y a las instrucciones sea la dirección de la memoria física del ordenador. Independientemente del nivel de abstracción en que se trabaje, la memoria es, de hecho, el entorno de la CPU. Cada instrucción realiza, en este modelo de ejecución, un cambio en el entorno: puede modificar algún dato en memoria y siempre implica determinar cuál es la dirección de la siguiente instrucción a ejecutar. Dicho de otra manera: la ejecución de una instrucción supone un cambio en el estado del programa. Éste se compone de la dirección de la instrucción que se está ejecutando y del valor de los datos en memoria. Entonces llevar a cabo una instrucción implica cambiar de estado el programa.

Desventajas:

- Los inconvenientes en este paradigma de programación son causados por la existencia y la utilización de la instrucción GOTO, debido a que el uso de la instruccion GOTO permite saltos en la ejecución del programa, lo que hace que el flujo del programa se desvie deliberadamente. Esta característica hace que sea dificil determinar el estado del programa, ademas de hacer intrincado el control de flujo, lo que puede provocar errores y excepciones en el programa. Esta instrucción tambien causa que los programas carezcan de orden y estructura, características que son fundamentales en la programación moderna.

- Uno de los mayores inconvenientes de este tipo de programación es la dificil interpretación del codigo escrito, debido a que los programas carecen de una estructura que pueda ser interpretada coherentemente por un ser humano.

- Este tipo de programación hace que sea imposible la reutilización de código, por las razones antes expuestas.

1.2.2. Programación Estructurada

Este paradigma de la programación está basado en la programación imperativa, a la que impone restricciones respecto de los saltos que pueden efectuarse durante la ejecución de un programa. Con estas restricciones se consigue aumentar la legibilidad del código fuente, permitiendo a sus lectores determinar con exactitud el flujo de ejecución de las instrucciones. Ejemplos de lenguajes de programación estructurada son: C, Pascal.

La programación estructurada resultó del análisis de las estructuras de control de flujo subyacentes a todo programa de computador. El producto de este estudio reveló que es posible construir cualquier estructura de control de flujo mediante tres estructuras básicas: la secuencial, la condicional y la iterativa.

La conjunción de estas propuestas proporciona las bases para la construcción de programas estructurados en los que las estructuras de control de flujo se pueden realizar mediante un conjunto de instrucciones muy reducido. De hecho, la estructura secuencial no necesita ninguna instrucción adicional, pues los programas se ejecutan normalmente llevando a cabo las instrucciones en el orden en que aparecen en el código fuente. Vale la pena indicar que, en cuanto a la programación estructurada se refiere, sólo es necesaria una única estructura de control de flujo iterativa. A partir de ésta se pueden construir todas las demás [7].

1.2.3. Programación Orientada a Objetos

El paradigma de la orientación a objetos nos propone una forma diferente de enfocar la programación sobre la base de la definición de objetos y de las relaciones entre ellos. Ejemplos característicos de lenguajes de programación orientados a objetos son: C++, C#, Java.

Cada objeto se representa mediante una abstracción que contiene su información esencial, sin preocuparse de sus demás características.

Esta información está compuesta de datos (variables) y acciones (funciones) y, a menos que se indique específicamente lo contrario, su ámbito de actuación está limitado a dicho objeto (ocultamiento de la información). De esta forma, se limita el alcance de su código de programación, y por tanto su repercusión, sobre el entorno que le rodea. A esta característica se le llama encapsulamiento.

Las relaciones entre los diferentes objetos pueden ser diversas, y normalmente suponen acciones de un objeto sobre otro que se implementan mediante mensajes entre los objetos.

[7] Esta condicion fue demostrada en 1966 por Bohm y Jacopini

Una de las relaciones más importante entre los objetos es la pertenencia a un tipo más general. En este caso, el objeto más específico comparte una serie de rasgos (información y acciones) con el más genérico que le vienen dados por esta relación de inclusión. El nuevo paradigma proporciona una herramienta para poder reutilizar todos estos rasgos de forma simple: la herencia.

Finalmente, una característica adicional es el hecho de poder comportarse de forma diferente según el contexto que lo envuelve. Es conocida como polimorfismo (un objeto, muchas formas). Además, esta propiedad adquiere toda su potencia al ser capaz de adaptar este comportamiento en el momento de la ejecución y no en tiempo de compilación.

En resumen estos son los elementos fundamentales que deben de poseer un lenguaje de programacion orientado a objetos:

- Abstracción: Determinación de las características de los objetos, que sirven para identificarlos y hacerlos diferentes a los demás.
- Encapsulamiento: Es el proceso que agrupa y almacena los elementos que definen la estructura y el comportamiento de una abstracción, en un mismo lugar.
- Modularidad: Es la propiedad de agrupar las abstracciones que guardan cierta relación lógica, y a la vez minimizar la interdependencia entre las diversas agrupaciones.
- Jerarquía: Consiste en establecer un orden o una clasificación de las abstracciones.

Además de estos elementos fundamentales, también existen otros 3 elementos secundarios , los cualesa aunque son recomendables, no son indispensables para clasificar un lenguaje dentro de este paradigma:

- Tipificación: Mecanismo que intenta restringir el intercambio entre abstracciones que poseen diversas características.
- Persistencia: Es la propiedad de un objeto a continuar existiendo a través del tiempo y/o del espacio.
- Concurrencia: Es la propiedad que distingue a los objetos activos, de los que no lo están.

1.2.4. Programación Funcional

El paradigma de programación funcional está basado en el concepto matemático de función. Una función es una correspondencia uno a uno del conjunto de argumentos(dominio) al conjunto de resultados(rango). Los programas en los lenguajes de programación funcional se construyen a partir de aplicaciónes de funciones, del mismo modo que en matemáticas el resultado de la función depende solamente de los argumentos de la función. Ejemplos de lenguajes de programación funcional son Haskell y Scheme.

Una de las propiedad mas importantes de un lenguaje de programación funcional es la transparencia referencial, la cual se define como la propiedad de no requerir el estado anterior del sistema o el valor actual de las variables locales o globales. En contraste, en los lenguajes de programación imperativa el resultado de una función no depende solamente de sus argumentos, sino que también depende de algún estado, tal como las variables globales.

La transparencia referencial facilita el análisis de los programas. Los métodos ya desarrollados en matemáticas pueden ser aplicados directamente para demostrar propiedades de los programas funcionales. Esto significa que se pueden realizar transformaciones en los programas que mantendrán ciertas propiedades[8]. Es fácil para un programador entender un fragmento del programa, ya que el programa no depende de variables globales dispersas a través de todo el programa. El significado de una linea de programa es determinado solamante a partir de esa línea de programa.

1.2.5. Programación Lógica y Basada en Reglas

El paradigma de programación lógica expresa la lógica de una aplicación sin tener que describir su control de flujo. Este paradigma de programación es también conocido como programación declarativa, ejemplos de lenguajes de programación lógica son SQL y Prolog.

El principal objetivo de este paradigma es describir que es lo que el programa debe conseguir mas que especificar como debe conseguirlo[9].

Las ventajas de este paradigma de programación lógica son:

- Permite una abstracción de los procedimientos requeridos para llevar a cabo una tarea, permitiendo que los programadores desarrollen sus aplicaciones eficientemente.

[8] Propiedades matemáticas como la invarianza, ortogonalidad, simetría, etc.

[9] A diferencia de la programación imperativa, en la cual se debe especificar explícitamente los procedimientos para la ejecución de un programa

- Elimina o minimiza los llamados "efectos secundarios" que consisten en inconsistencias o excepciones que se producen en programación imperativa y en menor medida en programación estructurada.

La implementación de lenguajes de programación utilizando este paradigma esta todavía en fase experimental, en la actualidad se encuentra en desarrollo para poder implementarlo en sistemas que utilizan esquemas de computación en paralelo.

1.3. Lenguajes de Programación en Función del Nivel de Abstracción

Existe una clasificación conceptual importante en el ámbito de los lenguajes de programación, la cual depende es

1.3.1. Lenguajes de Bajo Nivel

El lenguaje de programación de bajo nivel que se utiliza en los microcontroladores es el lenguaje ensamblador (assembly language), el cual es una versión simbólica de las operaciones y componentes físicos del microcontrolador. Los programas en lenguaje ensamblador se traducen en código máquina a través de un programa llamado assembler.

Las instrucciones en un lenguaje de bajo nivel tienen que tomar en cuenta los detalles de las características físicas del dispositivo, por ello comúnmente en el desarrollo de aplicaciones se tiene que programar aspectos que son completamente irrelevantes al algoritmo que se quiere implementar.

Ejemplos de lenguaje de bajo nivel son el conjunto de instrucciones RISC[10] y CISC.

1.3.2. Lenguajes de Alto Nivel

Los lenguajes de programación de alto nivel utilizan mecanismos de abstracción apropiados para asegurar que la programación sea independiente de los detalles de las características físicas del dispositivo. De esta forma los lenguajes de alto nivel están hechos para expresar algoritmos de forma que su lógica y estructura sea entendible para las personas, teniendo como objetivo que sus instrucciones emulen la expresión y el lenguaje natural de las personas.

Ejemplos de lenguajes de alto nivel son SQL, Prolog, entre otros.

[10] Los cuales son lenguajes ensamblador de diversas arquitecturas de microprocesadores y microcontroladores.

1.4. Consideraciones de Arquitectura de Software

1.4.1. Ciclo de desarrollo de Software

Para el desarrollo de software se tienen varios modelos idealizados que especifican el ciclo de desarrollo de las aplicaciones, los principales modelos (los cuales influyen en el desarrollo de la presente aplicación) se exponen a continuación:

Modelo en Cascada

El modelo en cascado deriva su nombre de la similidad de su estructura de proceso de desarrollo con una cascada. Este modelo se basa en los siguientes supuestos:

- El proceso de desarrollo de software consiste de un número de fases en secuencia, de forma que solamente cuando una fase se completa, el trabajo en la siguiente fase puede comenzar. Esto presupone un control de flujo unidireccional entre las fases.
- Desde la primera fase hasta la última, hay un flujo de información primaria hacia abajo y esfuerzos de desarrollo.
- El trabajo puede ser dividido de acuerdo a fases, involucrando diferentes clases de especialistas.
- Es posible asociar un objetivo para cada fase y planificar los resultados de cada fase.
- La salida de una fase corresponde a la entrada de la siguiente fase.
- Es posible desarrollar diferentes herramientas para satisfacer los requerimientos de cada fase.
- Las fases proveen una base para la administración y el control debido a que definen segmentos del flujo de trabajo, los cuales pueden ser identificados para propósitos administrativos, y especifica los documentos y resultados que serán producidos en cada fase.

Este modelo provee una aproximación práctica y disciplanada al desarrollo de software. Las fases consideradas en este modelo son las siguientes[11]:

[11] Tomando como referencia el modelo de Royce (1970), tomando como referencia [2].

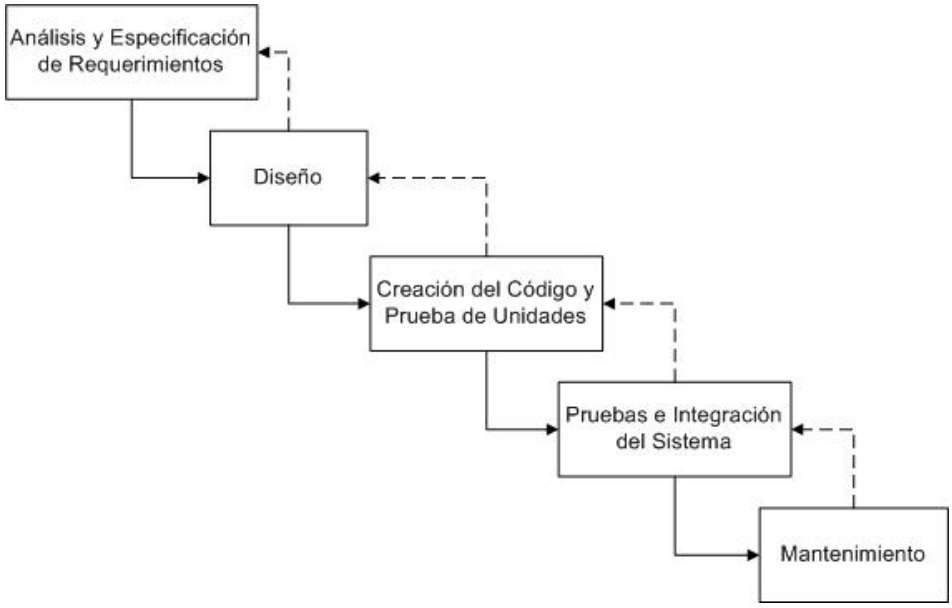

Figura 1.4.1: Modelo en Cascada de Desarrollo de Software

Las principales fases de este modelo de desarrollo de software se describen a continuación:

- Análisis y Especificación de Requerimientos: Los servicios, restricciones y objetivos se establecen consultando con los posibles usuarios del sistema. Entonces se definen en detalle y sirven como especifiación del sistema.
- Diseño: El proceso de diseño del sistema divide los requerimientos en sistemas de hardware y de software. De este modo se establece una arquitectura global del sistema. El diseño de software involucra identificar y describir las abstracciones fundamentales del sistema y sus relaciones.
- Creación de Código y Prueba de Unidades: En esta fase el diseño del software es implementado como un conjunto de unidades de programa (codificación). La prueba de unidades involucra la verificación de que cada unidad cumpla con su especificación.
- Pruebas e Integración del Sistema: Las unidades individuales del programa o programas se integran y prueba como un sistema completo, para asegurar que los requerimientos del software se han cumplido.
- Mantenimiento: El sistema es instalado y puesto en operación, entonces el mantenimiento involucra corregir errores que no se han encontrado en etapas anteriores del ciclo de desarrollo, mejorando la implementación de las unidades del sistema y actualizando los servicios del sistema a medida que se encuentran nuevos requerimientos.

1.4.2. Factores de Calidad del Software

Los factores de calidad del software son requerimientos no-funcionales, los cuales no se han específicado explícitamente, pero son requerimientos que mejoran la calidad del software. Los principales factores de calidad del software son:

- *Claridad:* Representa la claridad de objetivos del software. Además todo el diseño y la documentación de usuario debe estar claramente escrita para que sea fácilmente entendible.
- *Sencillez:* Minimización de toda la información o procesos redundantes. Esto es importante cuando la capacidad de memoria es limitada, y es en general considerada una buena práctica para mantener las líneas de código al mínimo. Se puede mejorar reemplazando las funcionalidades repetidas a través de una subrutina o función que implemente esa funcionalidad. También se aplica a la documentación del software.
- *Unidad:* Presencia de todas las partes constitutivas, con cada parte completamente desarrollada. Esto significa que si el código llama a una subrutina desde una librería externa, el paquete de software debe proveer la referencia a esa librería y todos los parámetros requeridos deben ser enviados.
- *Portabilidad:* Es la capacidad del software para ejecutarse adecuada y fácilmente en multiples configuraciones de sistemas. La portabilidad significa que el programa puede ejecutarse tanto en diferente hardware (por ejemplo PCs o smartphones) y entre diferentes sistemas operativos (por ejemplo Windows y Linux).
- *Consistencia:* Es la uniformidad en notación, simbología, apariencia, y terminología en todo el paquete de software.
- *Mantenibilidad:* Disponibilidad para facilitar actualizaciones para satisfacer nuevos requerimientos. En consecuencia el software que tiene la característica de mantenibilidad debe estar bien documentado, y tener capacidad libre de memoria, almacenamiento, utilización de procesador y otros recursos.
- *Verificabilidad:* Disposición para soportar evaluaciones de rendimiento. Esta característica debe ser implementada durante la fase de diseño para que el producto sea fácilmente probado, un diseño complejo hace que sean difíciles las pruebas.
- *Usabilidad:* Conveniencia y practicidad de uso. Esto es afectado por elementos como la interfaz humano-computador. El componente del software que tiene mayor impacto en este factor es la interfaz de usuario (UI[12]), la cual se recomienda que sea gráfica (GUI[13]).

[12]Siglas en inglés de User Interface: Interfaz de usuario.
[13]Siglas en inglés de Graphical User Interface: Interfaz gráfica de usuario.

- *Confiabilidad:* La confiabilidad significa que se puede esperar que el software realize satisfactoriamente las funciones a las que esta destinado. Esto implica un factor de tiempo en el cual se espera que un software confiable realize una tarea correctamente. También estan consideradas situaciones del entorno en las cuales se requiere que el software se desempeñe correctamente independientemente de las condiciones que se puedan presentar[14].

- *Eficiencia:* La eficiencia implica que se cumplan todos los objetivos sin deperdiciar recursos tales como memoria, espacio, utilización de procesador, ancho de banda de la red, tiempo, etc.

- *Seguridad:* Habilidad para proteger los datos contra acceso no autorizado y para no ser afectado por interferencia inadvertida o maliciosa en sus operaciones. Para esto se pueden implementar mecanismos tales como la autenticación, control de acceso y encriptación.

1.5. Microcontroladores dsPic

Los dispositivos para los cuales se ha desarrollado el lenguaje de lenguaje de programación gráfico son los Microcontroladores, y en particular la familia dsPic30F del fabricante Microchip. En esta sección se desarrollará una introducción a los microcontroladores y su arquitectura.

1.5.1. Arquitectura de los Microcontroladores dsPic

Para análizar la arquitectura de los microcontroladores dsPic se hará un análisis de los bloques que lo constituyen, sin análizar sus componentes internos; ya que el uno de los propósitos de un lenguaje de programáción gráfico es la abstracción del hardware del que esta constituido el microcontrolador. El siguiente diagrama de bloques muestra los componentes estructurales de un microcontrolador de la familia dsPic30F:

[14] Esta condición también es conocida como robustez.

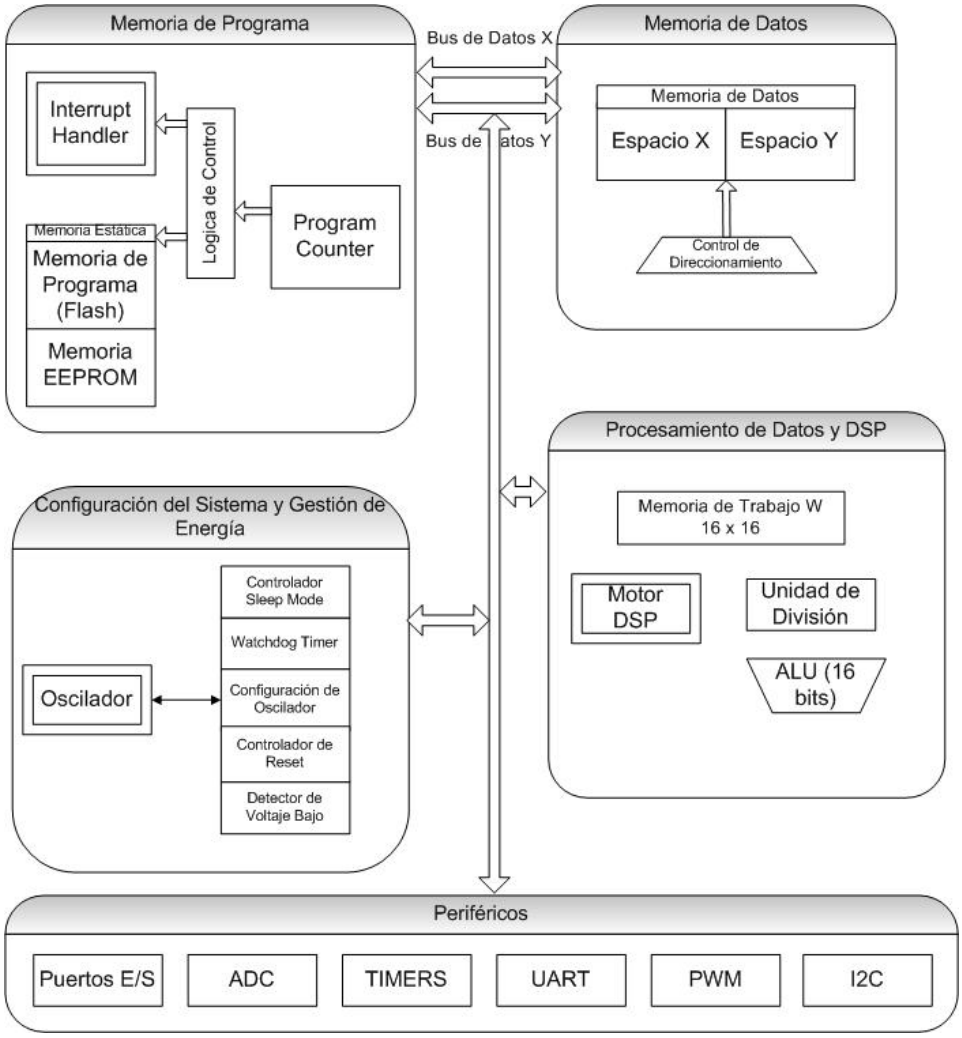

Figura 1.5.1: Arquitectura Microcontrolador dsPic30F

Los elementos de mayor reelevancia para el usuario de un microcontrolador programado a través de un sistema de programación gráfica serían los periféricos, debido a que son los elementos que se utilizan en las aplicaciones prácticas y a que la implementación del paradigma implica obtener la mayor abstracción posible de los demás elementos.

1.5.2. Aplicaciones

Entre los campos de aplicacion más reelevantes de los microcontroladores dsPic están las siguientes:

- Sistemas de Automatización Industrial.
- Domótica.
- Sistemas de Control.
- Instrumentación.
- Procesamiento Digital de Señales.
- Telecomunicaciones.

Cabe señalar que los microcontroladores son una alternativa de solución para un gran ámbito de problemas que pueden surgir en el campo de la Ingeniería Electrónica, Eléctrica, Mecánica, Industrial, Automotriz y de Control.

Capítulo 2

DESARROLLO DEL ENTORNO DE PROGRAMACIÓN GRÁFICO

En este capítulo se exponen los fundamentos del lenguaje de programación utilizado para implementar el sistema de programación gráfica, así como los correspondientes resultados de esta implementación.

2.1. El lenguaje de programación C# aplicado al desarrollo de interfaz gráfica

En esta sección se exponen las características y ventajas del lenguaje de programación C#, las cuales han llevado a que este sea el principal lenguaje de programación utilizado en este proyecto

2.1.1. El .Net Framework

Introducción

El .NET es el conjunto de nuevas tecnologías en las que Microsoft ha estado trabajando durante los últimos años con el objetivo de obtener una plataforma sencilla y potente para distribuir el software en forma de servicios que puedan ser suministrados remotamente y que puedan comunicarse y combinarse unos con otros de manera totalmente independiente de la plataforma, lenguaje de programación y modelo de componentes con los que hayan sido desarrollados, ésta es la llamada plataforma .NET.

Para crear aplicaciones para la plataforma .NET, tanto servicios Web como aplicaciones tradicionales (aplicaciones de consola, aplicaciones de ventanas, servicios de Windows NT, etc.), Microsoft ha publicado el denominado kit de desarrollo de software conocido como .NET Framework SDK, que incluye las herramientas necesarias tanto para su desarrollo como para su distribución y ejecución y Visual Studio.NET, que permite hacer todo la anterior desde una interfaz visual basada en ventanas.

El concepto de Microsoft.NET también incluye al conjunto de nuevas aplicaciones que Microsoft y terceros han (o están) desarrollando para ser utilizadas en la plataforma .NET. Entre ellas podemos destacar aplicaciones desarrolladas por Microsoft tales como Windows.NET, Hailstorm, Visual Studio.NET, MSN.NET, Office.NET, y los nuevos servidores para empresas de Microsoft (SQL Server.NET, Exchange.NET, etc.).

De igual forma, para plataformas basadas en GNU/Linux se ha desarrollado una implementación en Software Libre completa del .Net Framework, la cual se llama Mono y es perfectamente compatible con .NET.

El CLR (Common Language Runtime)

El Common Language Runtime (CLR) es el núcleo de la plataforma .NET. Es el motor encargado de gestionar la ejecución de las aplicaciones para ella desarrolladas y a las que ofrece numerosos servicios que simplifican su desarrollo y favorecen su fiabilidad y seguridad. Las principales características y servicios que ofrece el CLR son:

Modelo de programación consistente: A todos los servicios y facilidades ofrecidos por el CLR se accede de la misma forma: a través de un modelo de programación orientado a objetos. Esto es una diferencia importante respecto al modo de acceso a los servicios ofrecidos por los algunos sistemas operativos actuales (por ejemplo, los de la familia Windows), en los que a algunos servicios se les accede a través de llamadas a funciones globales definidas en DLLs y a otros a través de objetos (objetos COM en el caso de la familia Windows).

Seguridad de tipos: El CLR facilita la detección de errores de programación difíciles de localizar comprobando que toda conversión de tipos que se realice durante la ejecución de una aplicación .NET se haga de modo que los tipos origen y destino sean compatibles.

Aislamiento de procesos: El CLR asegura que desde código perteneciente a un determinado proceso no se pueda acceder a código o datos pertenecientes a otro, lo que evita errores de programación muy frecuentes e impide que unos procesos puedan atacar a otros. Esto se consigue gracias al sistema de seguridad de tipos antes comentado, pues evita que se pueda convertir un objeto a un tipo de mayor tamaño que el suyo propio, ya que al tratarlo como un objeto de mayor tamaño

podría accederse a espacios en memoria ajenos a él que podrían pertenecer a otro proceso. También se consigue gracias a que no se permite acceder a posiciones arbitrarias de memoria.

Tratamiento de excepciones: En el CLR todo los errores que se puedan producir durante la ejecución de una aplicación se propagan de igual manera: mediante excepciones. Esto es muy diferente a como se venía haciendo en los sistemas Windows hasta la aparición de la plataforma .NET, donde ciertos errores se transmitían mediante códigos de error en formato Win32, otros mediante HRESULTs y otros mediante excepciones. El CLR permite que excepciones lanzadas desde código para .NET escrito en un cierto lenguaje se puedan capturar en código escrito usando otro lenguaje, e incluye mecanismos de depuración que pueden saltar desde código escrito para .NET en un determinado lenguaje a código escrito en cualquier otro. Por ejemplo, se puede recorrer la pila de llamadas de una excepción aunque ésta incluya métodos definidos en otros módulos usando otros lenguajes.

Soporte multihilo: El CLR es capaz de trabajar con aplicaciones divididas en múltiples hilos de ejecución que pueden ir evolucionando por separado en paralelo o intercalándose, según el número de procesadores de la máquina sobre la que se ejecuten. Las aplicaciones pueden lanzar nuevos hilos, destruirlos, suspenderlos por un tiempo o hasta que les llegue una notificación, enviarles notificaciones, sincronizarlos, etc.

Distribución transparente: El CLR ofrece la infraestructura necesaria para crear objetos remotos y acceder a ellos de manera completamente transparente a su localización real, tal y como si se encontrasen en la máquina que los utiliza.

Seguridad avanzada: El CLR proporciona mecanismos para restringir la ejecución de ciertos códigos o los permisos asignados a los mismos según su procedendecia o el usuario que los ejecute. Es decir, puede no darse el mismo nivel de confianza a código procedente de Internet que a código instalado localmente o procedente de una red local; puede no darse los mismos permisos a código procedente de un determinado fabricante que a código de otro; y puede no darse los mismos permisos a un mismo códigos según el usuario que lo esté ejecutando o según el rol que éste desempeñe. Esto permite asegurar al administrador de un sistema que el código que se esté ejecutando no pueda poner en peligro la integridad de sus archivos, la del registro de Windows, etc.

Interoperabilidad con código antiguo: El CLR incorpora los mecanismos necesarios para poder acceder desde código escrito para la plataforma .NET a código escrito previamente a la aparición de la misma y, por tanto, no preparado para ser ejecutando dentro de ella. Estos mecanismos permiten tanto el acceso a objetos COM como el acceso a funciones sueltas de DLLs preexistentes (como la API Win32).

Como se puede inferir de las características comentadas, el CLR lo que hace es gestionar la ejecución de las aplicaciones diseñadas para la plataforma .NET. Por esta razón, al código de estas aplicaciones se le suele llamar código gestionado, y

al código no escrito para ser ejecutado directamente en la plataforma .NET se le suele llamar código no gestionado.

Librería de Clase Base (BCL)

La Librería de Clase Base (BCL[1]) es una librería incluida en el .NET Framework formada por cientos de tipos de datos que permiten acceder a los servicios ofrecidos por el CLR y a las funcionalidades más frecuentemente usadas a la hora de escribir programas. Además, a partir de estas clases prefabricadas el programador puede crear nuevas clases que mediante herencia extiendan su funcionalidad y se integren a la perfección con el resto de clases de la BCL. Por ejemplo, implementando ciertos interfaces podemos crear nuevos tipos de colecciones que serán tratadas exactamente igual que cualquiera de las colecciones incluidas en la BCL.

A través de las clases suministradas en ella es posible desarrollar cualquier tipo de aplicación, desde las tradicionales aplicaciones de ventanas, consola o servicio de Windows NT hasta los novedosos servicios Web y páginas ASP.NET. Es tal la riqueza de servicios que ofrece que puede crearse lenguajes que carezcan de librería de clases propia y sólo usen la BCL como C#.

Dada la amplitud de la BCL, ha sido necesario organizar las clases en ella incluida en Espacios de Nombres (namespace) que agrupen clases con funcionalidades similares. Por ejemplo, los espacios de nombres más usados son:

[1] Del Inglés: Base Class Library.

Namespace	***Utilidad/Tipo de Dato que contiene***
System	Tipos muy frecuentemente usados, como los tipos básicos, tablas, excepciones, fechas, números aleatorios, recolector de basura, entrada/salida en consola, etc.
System.Collections	Colecciones de datos de uso común como pilas, colas, listas, diccionarios, etc.
System.Data	Manipulación de bases de datos. Forman la denominada arquitectura ADO.NET.
System.IO	Manipulación de ficheros y otros flujos de datos.
System.Net	Realización de comunicaciones en red.
System.Reflection	Acceso a los metadatos que acompañan a los módulos de código.
System.Runtime.Remoting	Acceso a objetos remotos.
System.Security	Acceso a la política de seguridad en que se basa el CLR.
System.Threading	Manipulación de hilos.
System.Web.UI.WebControls	Creación de interfaces de usuario basadas en ventanas para aplicaciones Web.
System.Winforms	Creación de interfaces de usuario basadas en ventanas para aplicaciones estándar.
System.XML	Acceso a datos en formato XML.

Cuadro 2.1: Namespaces de la BCL mas utilizados

Common Language Specification (CLS)

El Common Language Specification (CLS) o Especificación del Lenguaje Común es un conjunto de reglas que han de seguir las definiciones de tipos que se hagan usando un determinado lenguaje gestionado si se desea que sean accesibles desde cualquier otro lenguaje gestionado. Obviamente, sólo es necesario seguir estas reglas en las definiciones de tipos y miembros que sean accesibles externamente, y no la en las de los privados. Además, si no importa la interoperabilidad entre lenguajes tampoco es necesario seguirlas. A continuación se exponen algunas de las reglas significativas del CLS:

- Los tipos de datos básicos admitidos son bool, char, byte, short, int, long, float, double, string y object. Nótese que no todos los lenguajes tienen porqué admitir los tipos básicos enteros sin signo o el tipo decimal como lo hace C#.
- Las tablas han de tener una o más dimensiones, y el número de dimensiones de cada tabla ha de ser fijo. Además, han de indexarse empezando a contar desde 0.

- Se pueden definir tipos abstractos y tipos sellados. Los tipos sellados no pueden tener miembros abstractos.
- Las excepciones han de derivar de System.Exception, los delegados de System.Delegate, las enumeraciones de System.Enum, y los tipos por valor que no sean enumeraciones de System.ValueType.
- Los métodos de acceso a propiedades en que se traduzcan las definiciones get/set de éstas han de llamarse de la forma get_X y set_X respectivamente, donde X es el nombre de la propiedad; los de acceso a indizadores han de traducirse en métodos get_Item y setItem; y en el caso de los eventos, sus definiciones add/remove han de traducirse en métodos de add_X y remove_X.
- En las definiciones de atributos sólo pueden usarse enumeraciones o datos de los siguientes tipos: System.Type, string, char, bool, byte, short, int, long, float, double y object.
- En un mismo ámbito no se pueden definir varios identificadores cuyos nombres sólo difieran en la capitalización usada. De este modo se evitan problemas al acceder a ellos usando lenguajes no sensibles a mayúsculas.
- Las enumeraciones no pueden implementar interfaces, y todos sus campos han de ser estáticos y del mismo tipo. El tipo de los campos de una enumeración sólo puede ser uno de estos cuatro tipos básicos: byte, short, int o long.

2.1.2. El lenguaje de Programación C#

Introducción

El lenguaje de programación C# [2] es el nuevo lenguaje de propósito general diseñado por Microsoft para su plataforma .NET. Sus principales creadores son Scott Wiltamuth y Anders Hejlsberg, éste último también conocido por haber sido el diseñador del lenguaje Turbo Pascal y la herramienta RAD Delphi.

Aunque es posible escribir código para la plataforma .NET en muchos otros lenguajes, C# es el único que ha sido diseñado específicamente para ser utilizado en ella, por lo que programarla usando C# es mucho más sencillo e intuitivo que hacerlo con cualquiera de los otros lenguajes ya que C# carece de elementos heredados innecesarios en .NET. Por esta razón, se suele decir que C# es el lenguaje nativo de .NET.

La sintaxis y estructuración de C# es muy similar a la C++, ya que la intención de Microsoft con C# es facilitar la migración de códigos escritos en estos lenguajes a

[2] Leído en inglés "C Sharp"

C# y facilitar su aprendizaje a los desarrolladores habituados a ellos. Sin embargo, su sencillez y el alto nivel de productividad son equiparables a los de Visual Basic.

Un lenguaje que hubiese sido ideal utilizar para estas tareas es Java, pero debido a problemas con la empresa creadora del mismo (Sun Microsystems), Microsoft ha tenido que desarrollar un nuevo lenguaje que añadiese a las ya probadas virtudes de Java las modificaciones que Microsoft tenía pensado añadirle para mejorarlo aún más y hacerlo un lenguaje orientado al desarrollo de componentes.

En resumen, C# es un lenguaje de programación que toma las mejores características de lenguajes preexistentes como Visual Basic, Java o C++ y las combina en uno solo. El hecho de ser relativamente reciente no implica que sea inmaduro, pues Microsoft ha escrito la mayor parte de la BCL usándolo, por lo que su compilador es el más depurado y optimizado de los incluidos en el .NET Framework SDK

Características

Las características más importantes del lenguaje de programación C# son las siguientes:

Sencillez: C# elimina muchos elementos que otros lenguajes incluyen y que son innecesarios en .NET. Por ejemplo:

- El código escrito en C# es autocontenido, lo que significa que no necesita de ficheros adicionales al propio fuente tales como ficheros de cabecera o ficheros IDL
- El tamaño de los tipos de datos básicos es fijo e independiente del compilador, sistema operativo o máquina para quienes se compile (no como en C++), lo que facilita la portabilidad del código.
- No se incluyen elementos poco útiles de lenguajes como C++ tales como macros, herencia múltiple o la necesidad de un operador diferente del punto (.) para acceder a miembros de espacios de nombres[3].

Modernidad: C# incorpora en el propio lenguaje elementos que a lo largo de los años ha ido demostrándose son muy útiles para el desarrollo de aplicaciones y que en otros lenguajes como Java o C++ hay que simular, como un tipo básico decimal que permita realizar operaciones de alta precisión con reales de 128 bits (muy útil en el mundo financiero), la inclusión de una instrucción foreach que permita recorrer colecciones con facilidad y es ampliable a tipos definidos por el usuario, la inclusión de un tipo básico string para representar cadenas o la distinción de un tipo bool específico para representar valores lógicos.

[3]Por ejemplo: Nombre.Clase.Metodo

Orientación a objetos: Como todo lenguaje de programación de propósito general actual, C# es un lenguaje orientado a objetos. Una diferencia de este enfoque orientado a objetos respecto al de otros lenguajes como C++ es que el de C# es más puro en tanto que no admiten ni funciones ni variables globales sino que todo el código y datos han de definirse dentro de definiciones de tipos de datos, lo que reduce problemas por conflictos de nombres y facilita la legibilidad del código.

C# soporta todas las características propias del paradigma de programación orientada a objetos: encapsulación, herencia y polimorfismo.

En lo referente a la encapsulación es importante señalar que aparte de los típicos modificadores public, private y protected, C# añade un cuarto modificador llamado internal, que puede combinarse con protected e indica que al elemento a cuya definición precede sólo puede accederse desde su mismo ensamblado.

Respecto a la herencia a diferencia de C++ y al igual que Java, C# sólo admite herencia simple de clases ya que la herencia múltiple provoca más dificultades técnicas que facilidades y en la mayoría de los casos su utilidad puede ser simulada con facilidad mediante herencia múltiple de interfaces.

Por otro lado y a diferencia de Java, en C# se ha optado por hacer que todos los métodos sean por defecto sellados y que los redefinibles hayan de marcarse con el modificador virtual (como en C++), lo que permite evitar errores derivados de redefiniciones accidentales. Además, un efecto secundario de esto es que las llamadas a los métodos serán más eficientes por defecto al no tenerse que buscar en la tabla de funciones virtuales la implementación de los mismos a la que se ha de llamar. Otro efecto secundario es que permite que las llamadas a los métodos virtuales se puedan hacer más eficientemente al contribuir a que el tamaño de dicha tabla se reduzca.

Orientación a componentes: La propia sintaxis de C# incluye elementos propios del diseño de componentes que otros lenguajes tienen que simular mediante construcciones más o menos complejas. Es decir, la sintaxis de C# permite definir cómodamente propiedades (similares a campos de acceso controlado), *eventos* (asociación controlada de funciones de respuesta a notificaciones) o *atributos* (información sobre un tipo o sus miembros)

Gestión automática de memoria: Todo lenguaje de .NET tiene a su disposición el recolector de basura[4] del CLR. Esto tiene el efecto en el lenguaje de que no es necesario incluir instrucciones de destrucción de objetos. Sin embargo, dado que la destrucción de los objetos a través del recolector de basura es indeterminista y sólo se realiza cuando éste se active, ya sea por falta de memoria, finalización de la aplicación o solicitud explícita en el fuente. C# también proporciona un mecanismo de liberación de recursos determinista a través de la instrucción *using*[5].

[4] El recolector de basura (del Inglés: Garbage Collector) es una función que se implemento por primera vez en Java y sirve para liberar memoria del sistema.

[5] La instruccion using sirve para hacer referencia a las librerias de clase o recursos que se utilizan en un programa en particular, por ejemplo: using Sistema.Libreria; .

Seguridad de tipos: C# incluye mecanismos que permiten asegurar que los accesos a tipos de datos siempre se realicen correctamente, lo que permite evita que se produzcan errores difíciles de detectar por acceso a memoria no perteneciente a ningún objeto y es especialmente necesario en un entorno gestionado por un recolector de basura. Para ello se toman las siguientes medidas:

- Sólo se admiten conversiones entre tipos compatibles. Esto es, entre un tipo y antecesores suyos, entre tipos para los que explícitamente se haya definido un operador de conversión, y entre un tipo y un tipo hijo suyo del que un objeto del primero almacenase una referencia del segundo (downcasting). Lo último sólo puede comprobarlo en tiempo de ejecución el CLR y no el compilador, por lo que en realidad el CLR y el compilador colaboran para asegurar la corrección de las conversiones.
- No se pueden usar variables no inicializadas. El compilador da a los campos un valor por defecto consistente en ponerlos a cero y controla mediante análisis del flujo de control del fuente que no se lea ninguna variable local sin que se le haya asignado previamente algún valor.
- Se comprueba que todo acceso a los elementos de una tabla se realice con índices que se encuentren dentro del rango de la misma.
- Se puede controlar la producción de desbordamientos en operaciones aritméticas, informándose de ello con una excepción cuando ocurra. Sin embargo, para conseguirse un mayor rendimiento en la aritmética estas comprobaciones no se hacen por defecto al operar con variables sino sólo con constantes (se pueden detectar en tiempo de compilación).
- A diferencia de Java, C# incluye delegados, que son similares a los punteros a funciones de C++ pero siguen un enfoque orientado a objetos, pueden almacenar referencias a varios métodos simultáneamente, y se comprueba que los métodos a los que apunten tengan parámetros y valor de retorno del tipo indicado al definirlos.
- Pueden definirse métodos que admitan un número indefinido de parámetros de un cierto tipo, y a diferencia lenguajes como C/C++, en C# siempre se comprueba que los valores que se les pasen en cada llamada sean de los tipos apropiados.

Instrucciones seguras: Para evitar errores muy comunes, en C# se han impuesto una serie de restricciones en el uso de las instrucciones de control más comunes. Por ejemplo, la condición de una estructura de control tiene que ser una expresión condicional y no aritmética, con lo que se evitan errores por confusión del operador de igualdad (==) con el de asignación (=); y todo caso de un *switch* tiene que terminar en un *break*, lo que evita la ejecución accidental de casos y facilita su reordenación.

Sistema de tipos unificado: A diferencia de C++, en C# todos los tipos de datos que se definan siempre derivarán, aunque sea de manera implícita, de una

clase base común llamada System.Object, por lo que dispondrán de todos los miembros definidos en ésta clase (es decir, serán "objetos").

A diferencia de Java, en C# esto también es aplicable a los tipos de datos básicos Además, para conseguir que ello no tenga una repercusión negativa en su nivel de rendimiento, se ha incluido un mecanismo transparente de boxing y unboxing con el que se consigue que sólo sean tratados como objetos cuando la situación lo requiera, y mientras tanto puede aplicárseles optimizaciones específicas.

El hecho de que todos los tipos del lenguaje deriven de una clase común facilita enormemente el diseño de colecciones genéricas que puedan almacenar objetos de cualquier tipo.

Versionable: C# incluye una política de versionado que permite crear nuevas versiones de tipos sin temor a que la introducción de nuevos miembros provoquen errores difíciles de detectar en tipos hijos previamente desarrollados y ya extendidos con miembros de igual nombre a los recién introducidos.

Si una clase introduce un nuevo método cuyas redefiniciones deban seguir la regla de llamar a la versión de su padre en algún punto de su código, difícilmente seguirían esta regla miembros de su misma signatura definidos en clases hijas previamente a la definición del mismo en la clase padre; o si introduce un nuevo campo con el mismo nombre que algún método de una clase hija, la clase hija dejará de funcionar. Para evitar que esto ocurra, en C# se toman dos medidas:

- Se obliga a que toda redefinición deba incluir el modificador "override"[6], con lo que la versión de la clase hija nunca sería considerada como una redefinición de la versión de miembro en la clase padre ya que no incluiría override. Para evitar que por accidente un programador incluya este modificador, sólo se permite incluirlo en miembros que tengan la misma signatura que miembros marcados como redefinibles mediante el modificador virtual. Así además se evita el error tan frecuente en Java de creerse haber redefinido un miembro, pues si el miembro con override no existe en la clase padre se producirá un error de compilación.
- Si no se considera redefinición, entonces se considera que lo que se desea es ocultar el método de la clase padre, de modo que para la clase hija sea como si nunca hubiese existido. El compilador avisará de esta decisión a través de un mensaje de aviso que puede suprimirse incluyendo el modificador new en la definición del miembro en la clase hija para así indicarle explícitamente la intención de ocultación.

Eficiente: En principio, en C# todo el código incluye numerosas restricciones para asegurar su seguridad y no permite el uso de punteros. Sin embargo, y a

[6] El modificador override (del Inglés override: sobreescribir) sobreescribe definiciones de métodos y propiedades en el lenguaje C#.

diferencia de Java, en C# es posible saltarse dichas restricciones manipulando objetos a través de punteros. Para ello basta marcar regiones de código como inseguras (modificador unsafe) y podrán usarse en ellas punteros de forma similar a cómo se hace en C++, lo que puede resultar vital para situaciones donde se necesite una eficiencia y velocidad procesamiento muy grandes.

Compatible: Para facilitar la migración de programadores, C# no sólo mantiene una sintaxis muy similar a C, C++ o Java que permite incluir directamente en código escrito en C# fragmentos de código escrito en estos lenguajes, sino que el CLR también ofrece, a través de los llamados Platform Invocation Services (PInvoke), la posibilidad de acceder a código nativo escrito como funciones sueltas no orientadas a objetos tales como las DLLs de la API Win32. Nótese que la capacidad de usar punteros en código inseguro permite que se pueda acceder con facilidad a este tipo de funciones, ya que éstas muchas veces esperan recibir o devuelven punteros.

2.1.3. Visual C# 2010 Express Edition

El Entorno Integrado de Programación (IDE[7]) utilizado en el desarrollo de la aplicación es Visual C# 2010 Express Edition. Este IDE tiene diversas herramientas que potencian el desarrollo de aplicaciones, tanto en línea de comandos, como para aplicaciones de Windows y aplicaciones Web. Entre las características que ofrece este IDE están:

- Automatiza los pasos requeridos para compilar el código fuente, pero al mismo tiempo permite un control total sobre las opciones requeridas para este proceso.
- El editor de texto esta diseñádo específicamente para el lenguaje de programación C#, de este modo el editor detecta errores y sugiere código al escribir.
- El IDE incluye herremientas de diseño para Windows Forms,Web Forms[8], y otras aplicaciones, habilitando un diseño simple de los elementos de la Interfaz de Usuario a través de un paradigma de arrastrar y soltar[9].
- Incluye varias herramientas que automatizan tareas comunes, muchas de las cuales pueden añadir código apropiado a archivos existentes, sin necesidad de que el usuario conozca la síntaxis.
- Contiene muchas herramientas de visualización y navegación a través de los elementos del proyecto, tanto si son archivos de código fuente de C# asi como recursos tales como imágenes o archivos de sonido.

[7] Del inglés: Integrated Development Environment

[8] Windows Forms es el nombre que se le da a una aplicación para Windows desarrollada en Visual C#, de igual modo Web Forms es para aplicaciones Web.

[9] Este paradigma es mejor conocido como “drag and drop".

- Permite utilizar técnicas avanzadas de debugging[10] al desarrollar proyectos, tales como la capacidad de ejecutar instrucciones paso a paso, para verificar el estado y el comportamiento de la aplicación.

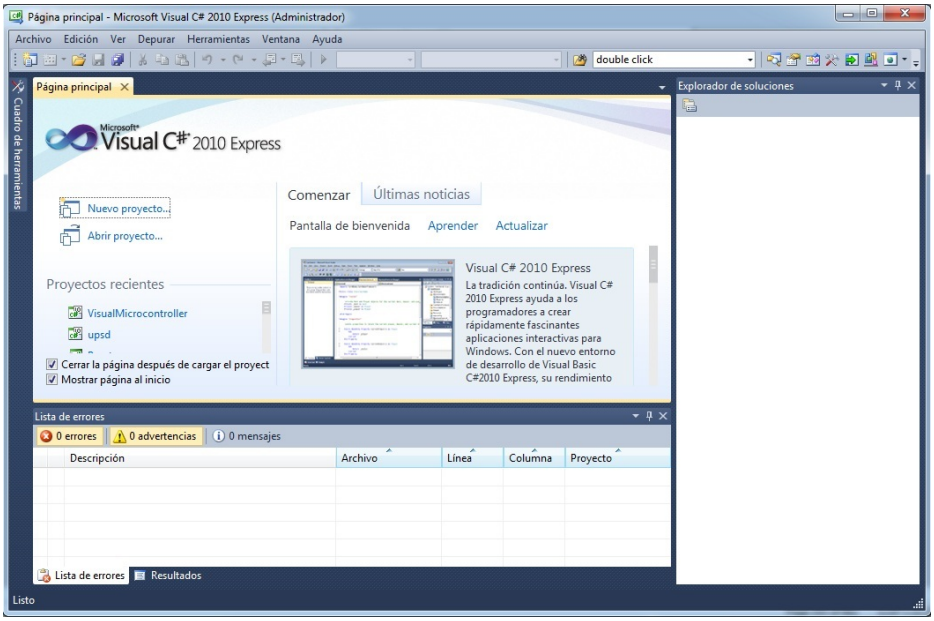

Figura 2.1.1: Interfaz de Usuario de Visual C# 2010 Express Edition

2.2. Diseño e Implementación del Entorno Integrado de Desarrollo (IDE)

2.2.1. Windows Forms

Las herramientas de Windows Forms ofrecen una solución al problema de implementación de Interfaces de Usuario Gráficas (GUI) en el entorno Visual C# 2010[11], estas de basan en un paradigma Drag and Drop, donde por lo general la unidad básica de trabajo (o contenedor) es un Panel, el cual consiste de una ventana con un espacio sobre el cuál se pueden ubicar los Controles.

[10] Debugging hace referencia a la prueba de las aplicaciones y la consiguiente corrección de errores de programación y optimización de los algoritmos.

[11] Siempre se hará referencia a la edición "Express Edition" de esta herramienta, la cual es de utilización libre, a diferencia de la edición "Professional".

Herramientas de Visual C# 2010

La siguiente gráfica muestra las principales Herramientas para el desarrollo de aplicaciones, disponibles en Visual C# 2010:

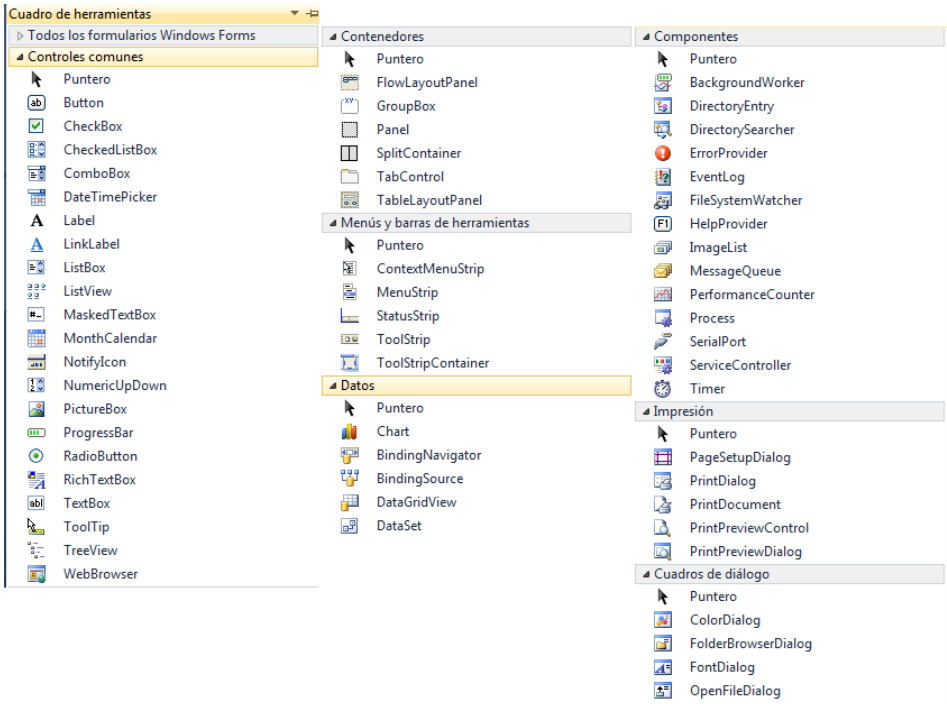

Figura 2.2.1: Herramientas Visual C# 2010

Estas herramientas estan categorizadas de acuerdo a su aplicación, visualmente se puede comprobar su utilidad y funcionalidad en el desarrollo de una aplicación con interfaz gráfica; ya que permite crear menus, botones, contenedores, cuadros de texto, controles y demás herramientas reelevantes al desarrollo de aplicaciones con interfaz gráfica. La utilización de estas herramientas también se puede realizar mediante líneas de código, pero esto es poco eficiente.

2.2.2. Resultados de Implementación

El resultado de la implementación de la interfaz gráfica de la aplicación Visual Microcontroller se muestra en la siguiente gráfica:

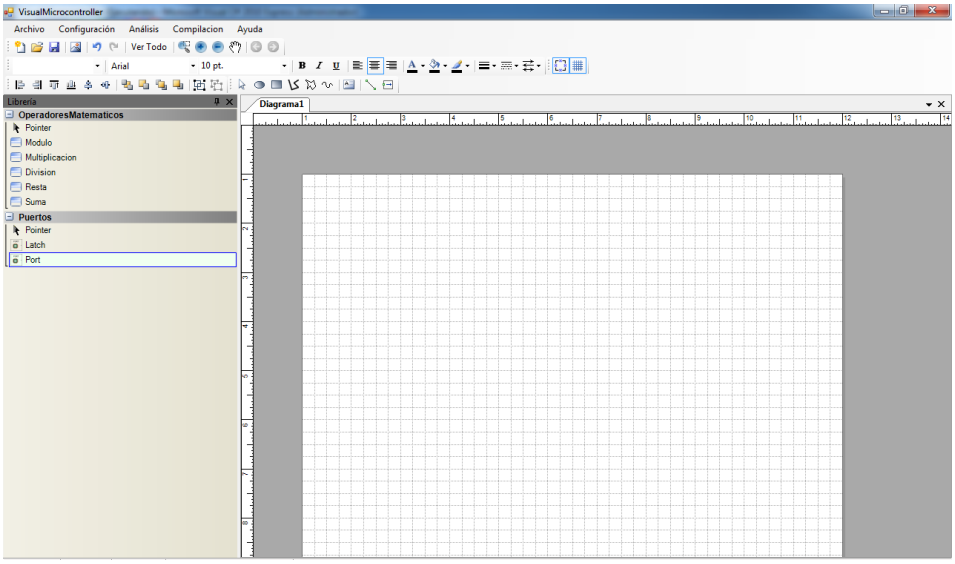

Figura 2.2.2: Interfaz Gráfica Visual Microcontroller

Esta implementación se he realizado utilizando tanto las herramientas de Visual C# 2010, asi como herramientas personalizadas contenidas en librerias[12], las cuales son invocadas a través de código.

[12]Las librerias en el .Net framework tienen una extensión .dll.

Capítulo 3

PROCESO DE IMPLEMENTACIÓN DE UN PARADIGMA DE PROGRAMACIÓN GRÁFICA

Los componentes que se requieren para ensamblar un paradigma de programación gráfica son basicamente un entorno de trabajo (workbench), los elementos de diagrama (bloques funcionales), y las reglas que relacionan los elementos de diagrama entre sí. Por ello se requiere de una libreria gráfica a partir de la cual se pueda derivar un entorno de trabajo adecuado para el sistema de programación gráfica. Estas herramientas y elementos se exponen en este capítulo.

3.1. Librería Gráfica Netron

Los sistemas de programación gráfica basados en software libre, en su mayoría, utilizan una librería que se constituya en el motor gráfico de su sistema. Por ejemplo el proyecto SciLab y en particular su herramienta SciCos utilizan la librería JGraph la cual esta basada en Java y utiliza para su GUI elementos de la librería GTK+. De igual forma aplicaciones comerciales tales como la herremienta Simulink de Matlab utilizan la librería JGraph. El inconveniente de esta librería es el hecho de que su código no es de libre utilización (además no es gratuito para aplicaciones comerciales), no esta liberado bajo licencias GPL o GNU[1] lo que dificulta su utilización y adquisición.

Existen varias alternativas de librerías que permiten el manejo de diagramas, de las cuales, como la más adecuada se determinó a la librería Netron, generada a partir

[1] Estos tópicos se analizan en mas detalle en el capítulo 8.

del proyecto del mismo nombre, y que esta liberada con licencia GNU (version 2 o superior), es de uso completamente libre, y es de código abierto[2]. Esta librería está desarrollada en Visual C#.

Esta librería ha sido modificada y ampliada para adecuarse a las necesidades del desarrollo de la aplicación de programación gráfica, de modo que la librería derivada es el motor gráfico del proyecto Visual Microcontroller.

3.1.1. Características de la Librería Netron

Las principales características de la librería Netron son:

- Código abierto programado en C#.
- Compatible con CLS, puede ser utilizado en cualquier lenguaje .Net.
- Controles compatibles con Visual Studio.
- Soporta Drag and Drop.
- Interfaz de programación para manejar grafos, nodos y conexiones.
- Edición interactiva y menus contextuales.s
- Modelo compatible y extensible de formas, conexiones y librerias de formas.
- Algorithmos de capas.
- Suporta paradigmas de programación visual de flujo de datos y entornos de desarrollo.
- Soporta zoom, tanto ampliación y reducción.
- Soporta completamente el CLR, no requiere de ensamblados adicionales.

3.1.2. Estructura de la librería gráfica Netron

El siguiente diagrama de paquetes UML[3] muestra la estructura de la librería gráfica Netron.

[2] Es decir su código fuente esta totalmente disponible y se puede modificar libremente.

[3] Los diagramas UML (Lenguaje Unificado de Modelado) sirven para describir un modelo de un software basado en programación orientada a objetos. En este caso el diagrama de paquetes muestra una división lógica del sistema, agrupados en paquetes, los cuales se presentan ordenados de forma jerárquica.

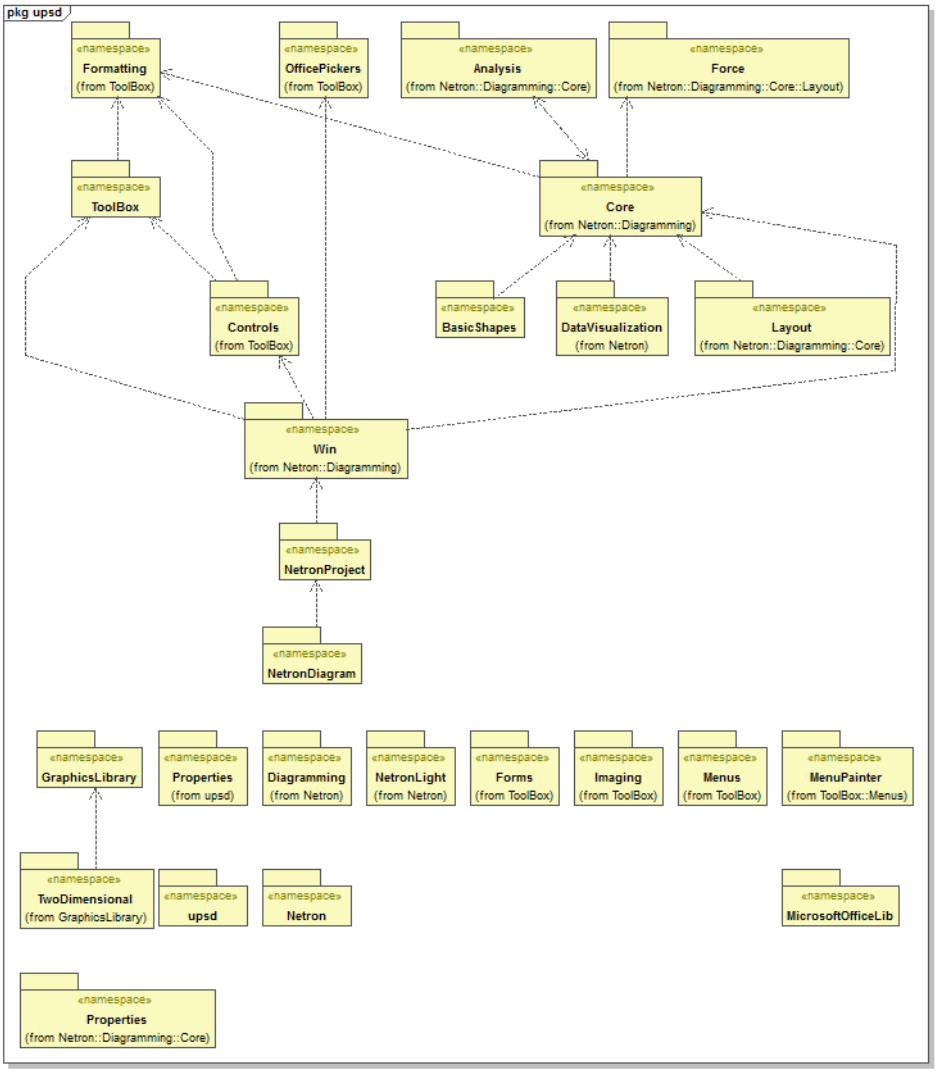

Figura 3.1.1: Estructura de la librería gráfica Netron (diagrama uml de paquetes)

3.2. Bloques Funcionales

El elemento básico de diagrama es el bloque funcional, el cual una vez configurado representará un proceso de un sistema de control tradicional. Debe observarse que cada bloque funcional en el entorno de programación es una máquina de estado

finita, ya que pueden tener mas de un estado posible[4]. Sin embargo una vez que se configura el bloque y se procede con el proceso de compilación y ejecución dicho bloque funcional tendrá un solo estado posible[5] (el equivalente a un bloque de un sistema de control tradicional); esta característica es inherente a la arquitectura del microcontrolador, ya que está optimizado para ser programado estructuralmente.

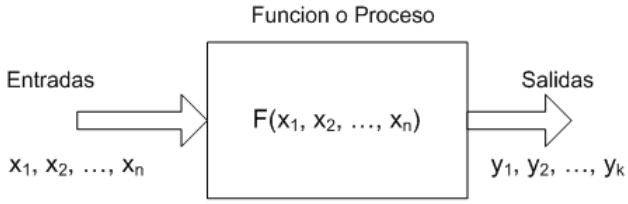

Figura 3.2.1: Modelo Matemático de un Bloque Funcional

Cada categoría[6] de bloques funcionales y sus elementos correspondientes se analizan en detalle en el capítulo 4.

3.2.1. Diseño gráfico de los bloques funcionales

Para la implementación de la parte visual de cada bloque funcional se establecieron los siguientes criterios guía:

- Cada gráfico debe ser reemplazable externamente sin afectar en lo absoluto al algoritmo de programación.
- Los gráficos deben tener la propiedad de escalabilidad, es decir se tienen que poder ampliar y reducir sin perder la calidad o nitidez.
- Se debe trabajar con una formato que permita robustez y adaptabilidad, para ser presentado en diferentes tipos de dispositivo.

Por ello se decidió trabajar con gráficos vectoriales, los cuales tienen las siguientes características:

[4]Existen varias categorias de bloques funcionales que tienen un solo estado posible, por ejemplo los Operadores Matemáticos y los Operadores Lógicos. De igual forma existen categorías de bloques con más de un estado posible tales como los Puertos (tienen el estado Port X bit N, donde X y N dependen del microcontrolador, en el caso del microcontrolador dsPic30F2010 $X \epsilon \{A, B, C, D, E, F\}$ y $N \epsilon \{A : 0, 1, 2 \cdots 6 \quad B : 0, 1, 2, \cdots 7, \cdots\}$.

[5]Como se analizó en el capítulo 1 si una máquina de estado finita tiene un solo estado, entonces es equivalente a un sistema de control tradicional

[6]Los bloques funcionales en el proyecto Visual Microcontroller estan agrupados en Categorías, las cuales contienen bloques relacionados por una funcionalidad o característica común.

- Las imágenes vectoriales están formadas por objetos geométricos tales como líneas, polígonos y arcos; los cuales se pueden representar matemáticamente mediante un lenguaje genérico y potente. Los parámetros matématicos mas importantes son la forma, la posición, el color, y la orientación.

- Los archivos de imágenes en formato vectorial tiene menor tamaño que un archivo de imagén de iguales características en mapa de bits[7].

- Las características de este tipo de gráficos permiten realizar transformación visuales y geométricas, y existen herramientas que permiten animación de imágenes vectoriales. Está se realiza de forma sencilla mediante operaciones básicas como traslación o rotación y no requiere de grandes cantidades de recursos o procesamiento.

- Los gráficos vectoriales no pierden calidad al ser escalados, rotados o deformados. Se puede hacer zoom sobre una imagen vectorial sin ningún tipo de restricción.

Para implementar la parte visual de cada bloque funcional se utilizó el programa Inkscape, el cual es un proyecto en software libre, el cual tiene características y funciones ideales para trabajar con el formato de gráficos vectoriales .svg. Este programa es de libre utilización y se utilizaron imágenes vectoriales libres de copyright[8] para facilitar la implementación de la parte gráfica de cada bloque funcional.

[7] En los mapas de bits o Bitmaps la información de una imagen esta representada mediante bits para cada pixel o punto en la imagen, por lo que este formato es muy poco practico para aplicaciones que manejen imágenes con características que pueden representarse de forma geométrica.

[8] La principal librería de imágenes utilizada en este proyecto es openclipart.org.

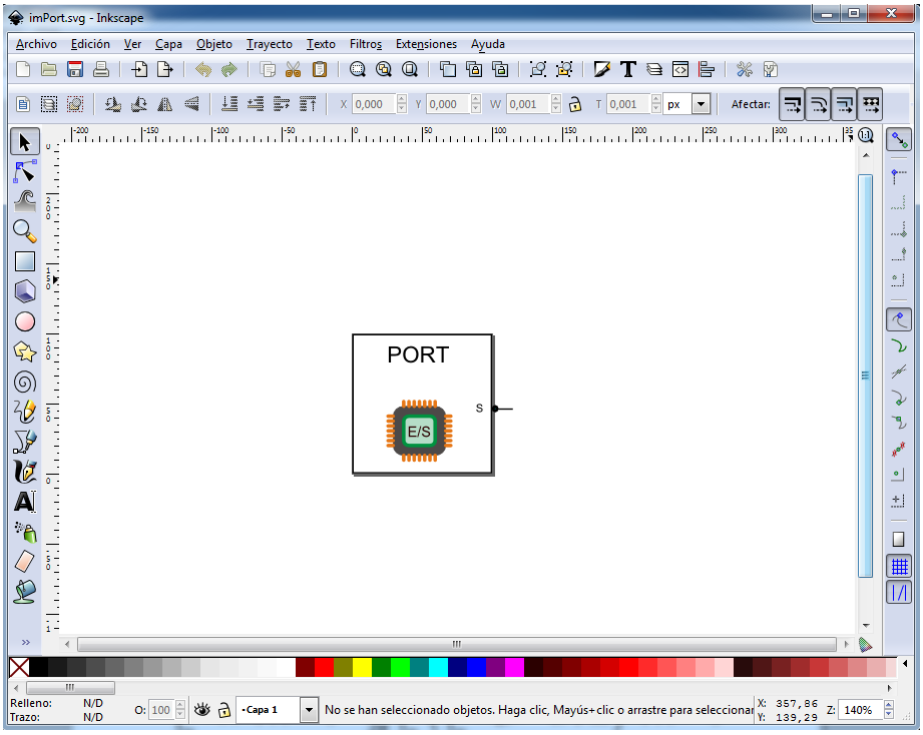

Figura 3.2.2: Interfaz de Inkscape

3.2.2. Algoritmo de implementación de un bloque funcional

Cada bloque funcional esta representado mediante una forma (o shape en el contexto de la librería Netron), a la cual se le han asignado propiedades que se adapten al propósito de la implementación del lenguaje de programación gráfico.

Notese que para cada bloque funcional es necesario adaptar este algoritmo, de forma que para cada uno de los bloques funcionales se ha programado un algoritmo siguiendo estas directrices, de forma que no se analizarán todos y cada uno de los algoritmos, sino solamente un algoritmo genérico que sirva para exponer las técnicas y recursos de programación utilizados. La estructura de la clase programada es la siguiente.

Algoritmo 3.1 Estructura de la Clase de un Bloque Funcional

```
/* Estructura de la Clase de un Bloque Funcional */

namespace Puertos
{
    [Serializable()]
    [Shape(
    "Port", // Nombre
    "Port", // Clave
    "Puertos Entrada/Salida", // Categoria
    "Puerto Entrada", // Descripción
    false)]

    public partial class Port : ComplexShapeBase
    {
        Campos
        Propiedades
        Constructor
        Serialización
        Métodos
        |-> Eventos
    }
```

Para analizar el algoritmo implementado no se utilizará la totalidad del código programado, en su lugar se hara un resumen de los procesos que tienen lugar, sin entrar en detalles técnicos de programación, y después se hará el análisis de la programación correspondiente.

Algoritmo 3.2 Pseudocódigo de Implementación de un Bloque Funcional

```
/* Creación de un Bloque Funcional */

1. Incluir librerías (instrucción using).
2. Declarar campos de la clase del bloque funcional.
3. Declarar propiedades de la clase del bloque funcional.
4. Constructor de la clase del bloque funcional.
5. Métodos de la clase del bloque funcional.
    5.1 Inicialización.
        -Crear la forma rectángular que contendrá al bloque
            funcional.
        -Crear los conectores del bloque funcional.
    5.2 Dibujar el Bloque Funcional.
        -Añadir y dibujar la imágen del bloque funcional.
        -Dibujar los conectores del bloque funcional.
6. Eventos de la Clase del Bloque Funcional.
    6.1 Doble Click -> Invocar a la herramienta de configuración
```

El código asociado a la implementación de estos procesos se expone a continuación, junto con una explicación de los objetivos y técnicas empleados en su programación:

Incluir Librerías

Para la invocación de librerías se utiliza la instrucción **using** seguida del nombre de la librería a la cual se quiere invocar. En el caso de este algoritmo se invocan las librerías principales del sistema (System), además de librerías especializadas en el manejo de imágenes, manejo de threads e interfaz gráfica. Finalmente se invoca al núcleo de la librería Netron (Netron.Diagramming.Core).

Algoritmo 3.3 Invocación de Librerías

```
/* 1. Incluir Librerías */

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Printing;
using System.IO;
using System.Drawing.Design;
using System.Runtime.CompilerServices;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Reflection;
using System.Threading;
using System.Windows.Forms;
using Netron.Diagramming.Core;
using Netron.Diagramming.Core.Analysis;
```

Declarar Campos de la Clase

Los campos de la clase, son las variables que almacenan los datos necesarios para que la clase realize sus objetivos, por regla general sus variables tienen accesibilidad privada (private) o protegida (protected), lo que implica que solo se pueden acceder desde el interior de la clase y no son accesibles desde aplicaciones externas.

Algoritmo 3.4 Declaración de Campos

```
/* 2. Declarar Campos de la clase */

        #region Campos
            protected const double portVersion = 1.0;
            #region Conectores
                private Connector cSalida;
            #endregion
        private string mPortPath;
        private Image mPort;
        string sIdentificador; //Esta variable contiene la
            numeración asignada a ese bloque.
        DataTable dtPropiedades; // Esta variable contiene todas
             las propiedades relacionadas con la programación
            del Microcontrolador.
        #endregion
```

En el presente caso los campos que se han creado son los reelevantes a la identificación del bloque funcional (para uso interno), y a los conectores, paths e imágenes.

Declarar Propiedades de la Clase

Las propiedades en C# sirven para acceder, escribir, calcular o recalcular campos privados de una clase; por tanto representan la parte de acceso público (**public**) de la clase. Las instrucciones para llevar a cabo estas tareas son:

- **get**, esta instrucción sirve para obtener el valor de un campo privado de la clase.
- **set**, mediante esta instrucción se puede establecer, calcular o determinar mediante algún método, el valor de un campo privado de la clase.

Para las diversas propiedades de la clase del bloque funcional se han utilizado estas instrucciones, dependiendo de sus características y objetivos (de modo que algunas propiedades solo tienen un ámbito get o set, y otras propiedades tienene ambos).

Algoritmo 3.5 Declaración de Propiedades

```
/* 3. Declarar Propiedades de la clase */
    #region Propiedades

    /// <summary>
    /// Obtener la versión actual.
    /// </summary>
    public override double Version
    {
        get
        {
            return portVersion;
        }
    }

    /// <summary>
    /// Obtener el nombre del Bloque Funcional para la Interfaz
        Gráfica
    /// </summary>
    /// <value>Nombre del Bloque Funcional</value>
    public override string EntityName
    {
        get
        {
            return "Port";
        }
    }

    /// <summary>
    /// Obtener o Establecer la imágen.
    /// </summary>
    /// <value>La imágen.</value>
    public Image Image
    {
        get
        {
            return mPort;
        }
        set
        {
            if (value == null) return;
            mPort = value;
            mPortPath = string.Empty;
           Transform(Rectangle.X, Rectangle.Y, mPort.Width,
               mPort.Height);
        }
    }
```

Algoritmo 3.6 Declaración de Propiedades (continuación)

```
        /// <summary>
        /// Obtener Icono de Imagen para la Libreria
        /// </summary>
        /// <value>Icono para la Librería</value>
        public override Image LibraryImage
        {
            get
            {
                Image ImPuertosES = Resource1.icPort;
                return ImPuertosES;
            }
        }

        /// <summary>
        /// Obtener o Establecer Identificador
        /// </summary>
        /// <value>Identificador</value>
        public string Identificador
        {
            get
            {
                return sIdentificador;
            }
            set
            {
                if (value == null) return;
                sIdentificador = value;
            }
        }

        /// <summary>
        /// Establecer u Obtener Propiedades(microcontrolador)
        /// </summary>
        /// <value>Propiedades(microcontrolador)</value>
        public DataTable Propiedades
        {
            get
            {
                return dtPropiedades;
            }
            set
            {
                if (value == null) return;
                dtPropiedades = value;
            }
        }
        #endregion
```

Constructores de la Clase

Los constructores de una clase sirven para instanciar a un objeto, cada constructor se aplica para parámetros diferentes, y en muchos casos se incluye en el constructor una secuencia de inicialización. En la programación de la clase para un Bloque Funcional, solo se utilizan los constructores por defecto.

Algoritmo 3.7 Constructor de la Clase

```
/* 4. Constructores de la Clase */

#region Constructor
/// <summary>
/// Constructor
/// </summary>
public Port()
: base()
{
}

/// <summary>
/// Constructor por defecto de la clase.
/// </summary>
public Port(IModel site)
: base(site)
{
}

protected Port(
SerializationInfo info,
StreamingContext context)
: base(info, context)
{
    double version = info.GetDouble("portVersion");
}
#endregion
```

Métodos de la Clase

Los métodos son el equivalente en programación orientada a objetos de las funciones[9], con la notable diferencia de que se pueden sobrecargar sus parámetros de entrada[10].

[9] Analizadas en la sección 1.1.3

[10] Es decir que la función puede procesar dependiendo de su programación, varios tipos de variable para un mismo parámetro.

En el método de inicialización de un bloque funcional, los procesos que tienen lugar son los siguientes:

- Definir las dimensiones del área rectangular.

- Crear un conector (en este caso de salida) y definir sus parámetros.

Algoritmo 3.8 Inicialización de un Bloque Funcional

```
/* 5.1 Inicialización del Bloque Funcional */

protected override void Initialize()
{
    base.Initialize();
    //Dimensiones del rectángulo
    Transform(0, 0, 160, 150);

    #region Conectores
    cSalida = new Connector(new Point(Rectangle.Right, (int)(
        Rectangle.Top + Rectangle.Height / 2)), Model);
    cSalida.Name = "salida";
    cSalida.Parent = this;
    Connectors.Add(cSalida);
    #endregion
}
```

Estos procesos pueden visualizarse mejor en el diagrama UML de secuencias.

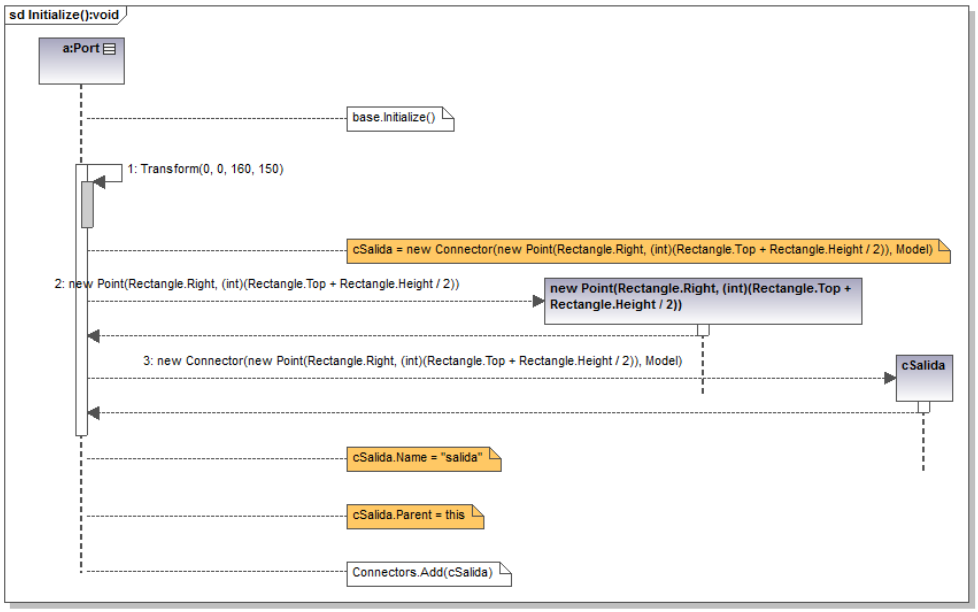

Figura 3.2.3: Inicialización de un Bloque Funcional

El método Paint sirve para dibujar sobre el entorno gráfico (Graphics g), los elementos que han sido definidos de manera lógica en la inicialización, los procedimientos realizados son:

- Cargar la imágen (mPort) que contiene la apariencia gráfica del bloque funcional.
- Dibujar la imágen sobre el área rectangular definida durante la inicialización.
- Dibujar los conectores.

Algoritmo 3.9 Método para Dibujar un Bloque Funcional

```
/* 5.2 Dibujar el Bloque Funcional */

public override void Paint(Graphics g)
{
    base.Paint(g);
    mPort = Resource1.imPort;

    if (mPort == null)
    {
        g.FillRectangle(Brush, Rectangle);
        StringFormat sf = new StringFormat();
        sf.Alignment = StringAlignment.Center;
        g.DrawString("Imagen no Encontrada", ArtPalette.
            DefaultFont, Brushes.Black, Rectangle.X + (Rectangle
            .Width / 2), Rectangle.Y + 3, sf);
    }
    else
    {
        g.DrawImage(mPort, Rectangle);
    }

    foreach (IConnector con in mConnectors)
    {
    con.Paint(g);
    }
}
```

Los procesos realizados se muestran en el diagrama de secuencias correspondiente.

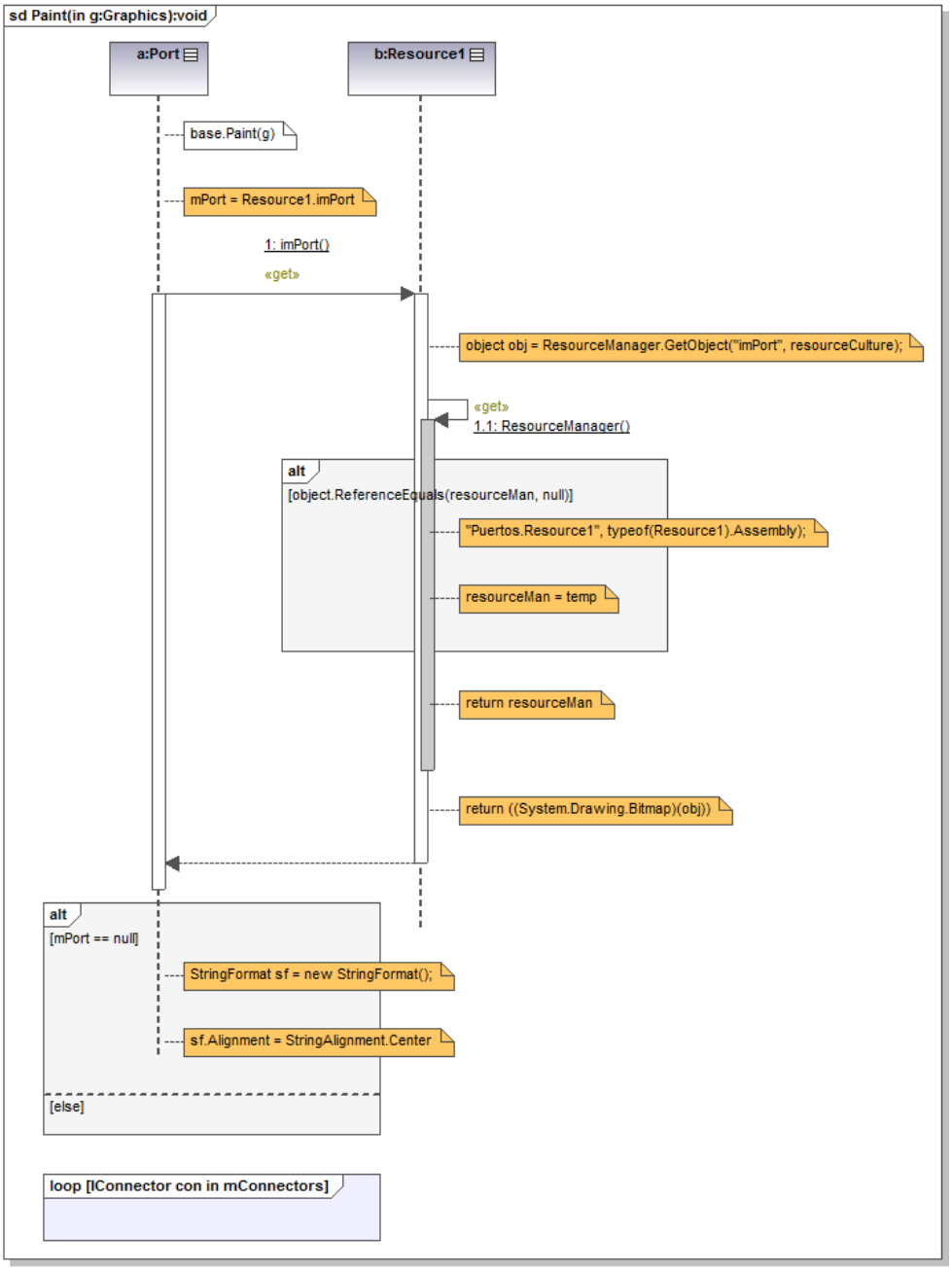

Figura 3.2.4: Método para Dibujar un Bloque Funcional

Manejo de Eventos del Bloque Funcional

Algoritmo 3.10 Eventos del Bloque Funcional

```
/* 6. Eventos del Bloque Funcional */

public override bool MouseDown(MouseEventArgs e)
{
    if ((e.Button == MouseButtons.Left) &&(e.Clicks == 2))
    {
        try
        {
            //Debug.Write("Doble Click");
            wPort cfPort = new wPort();
            cfPort.StartPosition = FormStartPosition.Manual;
            cfPort.Location = new Point(Rectangle.X, Rectangle.Y
                );
            cfPort.Visible = true;
            return true;
        }
        catch (Exception ex)
        {
            MessageBox.Show("Error en configuración de
                Propiedades de Bloque.\n" + ex.Message);
            return false;
        }
    }
    return base.MouseDown(e);
}
```

Los eventos permiten que una clase u objeto notifique cuando ocurra algo. En una aplicación común de Windows Forms los eventos son generados por controles como botones, menús y listas. La clase que envía un evento es llamada el editor, y la clase que recive o maneja el evento es llamada el suscriptor.

En esta aplicación, básicamente se requiere que se ejecute la herramienta de configuración cuando se haga doble click sobre el bloque funcional. Para implementar esta funcionalidad se tiene la clase MouseEvent, que dispara los eventos relacionados con el mouse. El evento del doble click es manejado por la clase del bloque funcional a través del método MouseDown. Estos eventos estan representados en el siguiente diagrama.

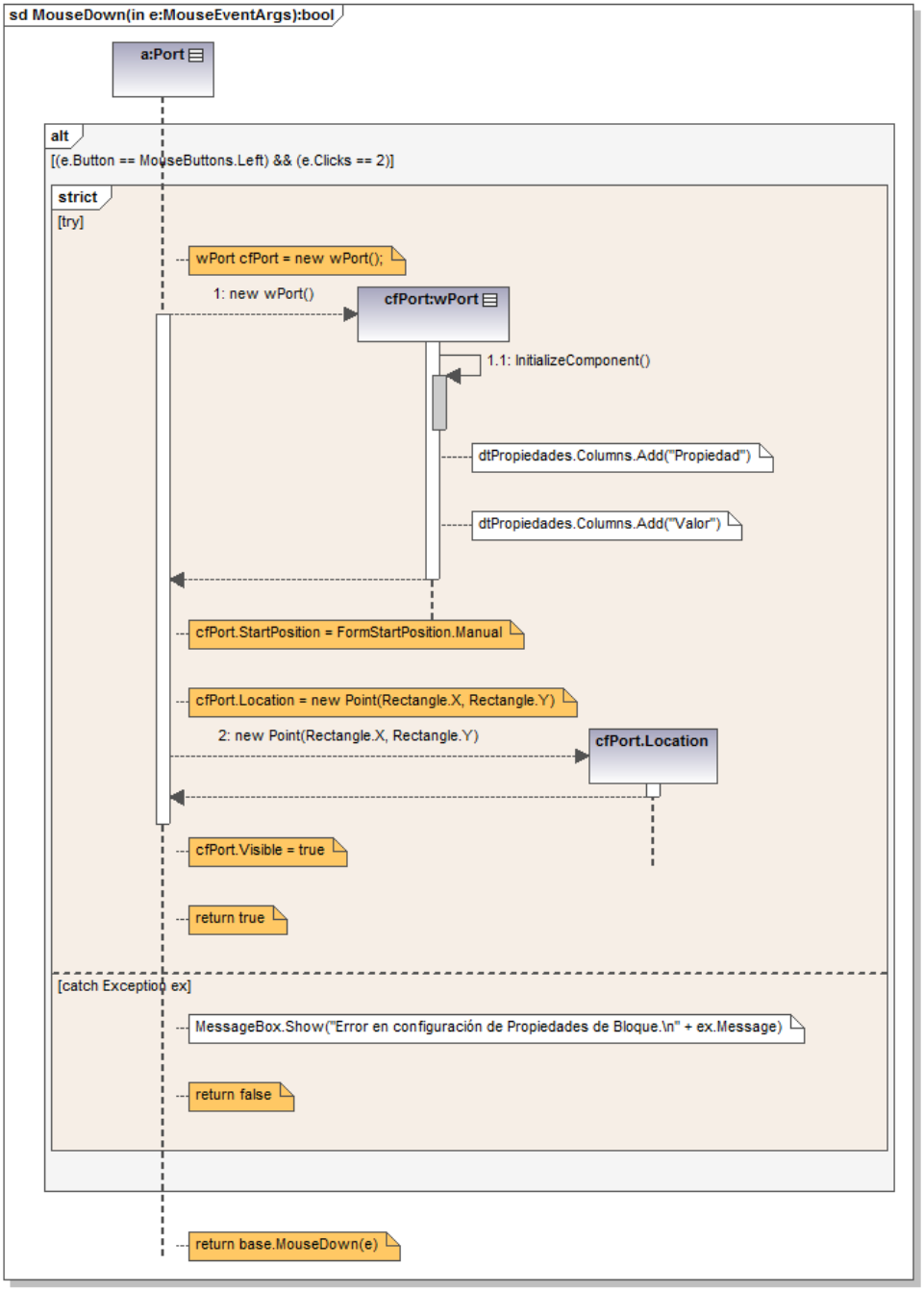

Figura 3.2.5: Diagrama de secuencia del evento MouseDown

Capítulo 4

COMPILACIÓN

De forma general la compilación tiene como objetivo convertir código programado en un lenguaje de programación en código máquina que pueda ejecutarse sobre una máquina física o virtual[1]. Aplicado al proyecto, el proceso de compilación se encarga de transformar un diagrama desarrollado en el sistema de programación gráfica en código apto para ser grabado sobre un microcontrolador.

Este capítulo expone la implementación del proceso de compilación, haciendo primero una introducción al proceso de compilación en general, y desarrollando cada instancia del proceso con los algoritmos y las herramientas aplicadas.

4.1. Introducción al Proceso de Compilación

El proceso de compilación tiene varias fases con interfaces bien definidas. Estas fases operan en secuencia[2], de modo que cada fase (excepto la primera) toma la salida de la etapa anterior como entrada. Es común que cada fase sea manejada por un módulo independiente. Algunos de estos módulos son desarrollados manualmente, mientras que otros pueden generarse a partir de especificaciones. Algunos de los módulos pueden ser compartidos por varios compiladores. Las fases del proceso de compilación se muestran en la siguiente figura.

[1] Por ejemplo el caso de la máquina virtual de java.

[2] Esta no es una regla general, y en la práctica las fases se pueden intercalar.

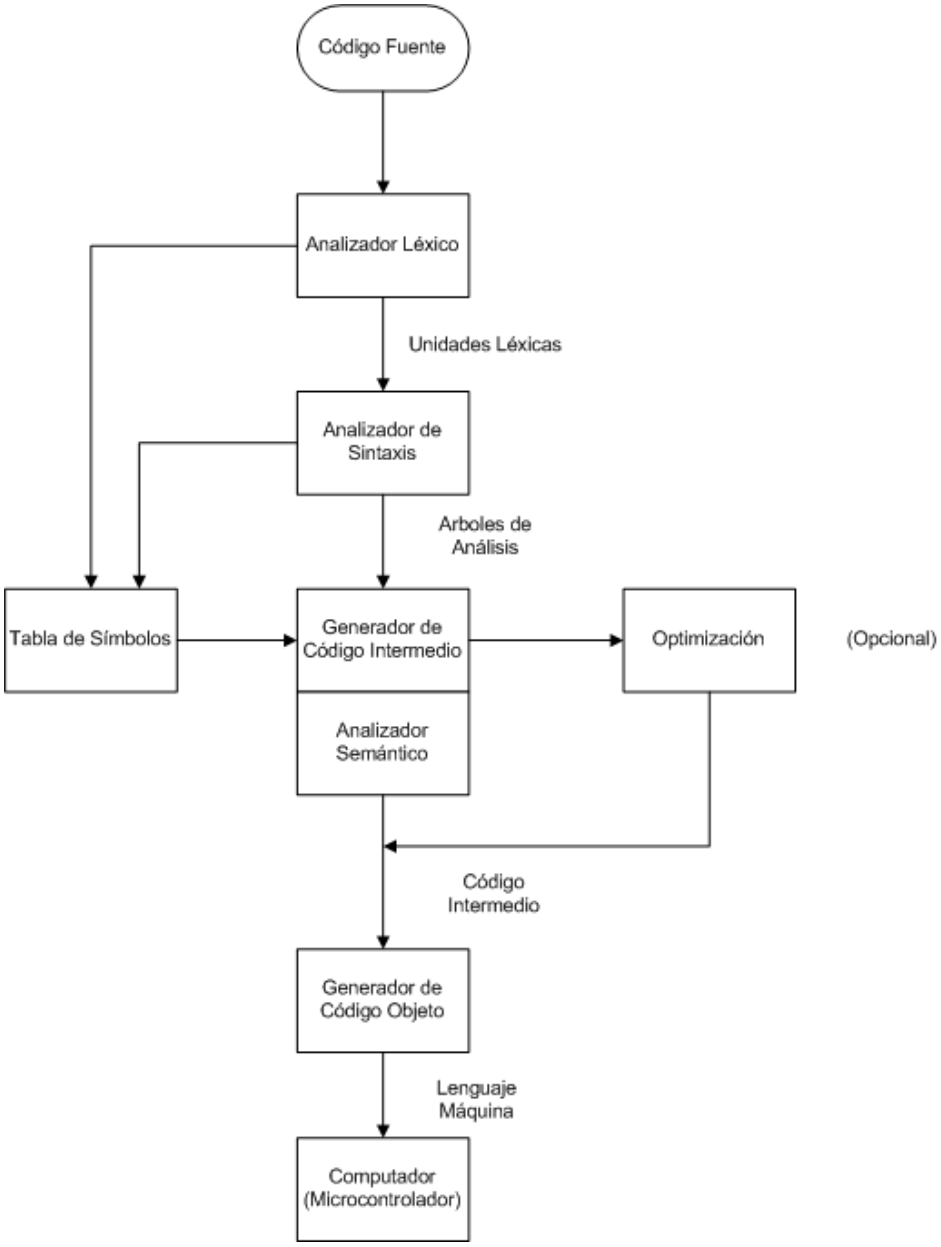

Figura 4.1.1: Fases del Proceso de Compilación

Debe notarse que no todas las fases del proceso de compilación son necesarias en todas las aplicaciones. La función de las fases del proceso de compilación se describe a continuación.

- Análisis Léxico: Es la parte inicial de la lectura y análisis del código del programa: el texto (o representación gráfica) se lee y se divide en unidades léxicas o "tokens", cada una de los cuales corresponde a un símbolo en el lenguaje de programación, por ejemplo, un nombre de variable, una palabra clave o un número.
- Análisis Sintactico: Esta fase adquiere la lista de tokens producida por el análisis léxico y las dispone en una estructura de árbol (llamado el árbol de sintaxis) que refleja la estructura del programa. Esta fase es llamada también "parsing".
- Análisis Semántico: Esta fase analiza el árbol de sintaxis para determinar si el programa viola ciertos requisitos de coherencia, por ejemplo, si una variable se utiliza pero no esta declarada o si se utiliza en un contexto que no tiene sentido dado el tipo de la variable [3].
- Generación de Código Intermedio: El programa se traduce en un lenguaje intermedio independiente de la máquina[4] de la aplicación.
- Generación de Código Objeto: El código intermedio se traduce al lenguaje máquina o a lenguaje ensamblador(una representación textual del lenguaje máquina) para una arquitectura de máquina específica.

Las tres primeras fases se conocen colectivamente como el "frontend" del compilador y los últimos tres fases se denominan el "backend". La parte intermedia del compilador en este contexto es la generación de código intermedio, pero esta frecuentemente incluye varias optimizaciones y transformaciones en el código intermedio.

4.2. Análisis Léxico

4.2.1. Introducción

El análisis léxico[5] divide el código fuente en unidades significativas. El programa cuando llega como entrada al compilador es como una enorme cadena de caracteres. Por ejemplo es fácil para un programador reconocer los elementos significativos en una expresión como

```
for(i=1;i==max;i++)
{
    x[i]=0;
}
```

[3] Por ejemplo tratar de usar un valor de tipo booleano como función de puntero
[4] Independiente del hardware o sistema.
[5] También llamado escaneo léxico (lexical scanning).

Sin embargo el compilador no reconoce estos elementos como lo hacen los humanos. Lo que el compilador ve es una secuencia de cadenas de bits, y la tarea del analizador léxico es dividir esta expresión en unidades significativas, estas unidades son llamadas tokens. En esta expresión, las unidades significativas son

- palabras clave (keywords): for
- identificadores: i, max, x
- operadores: =
- puntuación: ;
- corchetes: {}, []

El análisis léxico es relativamente fácil en la mayoría de lenguajes de programación modernos, ya que el programador necesariamente tiene que separar muchas partes de la expresión con espacios en blanco o separadores; estos espacios en blanco facilitan que el compilador pueda determinar dónde termina un símbolo y el comienza el siguiente.

El analizador léxico con frecuencia realiza también otras operaciones. Puede quitar el exceso de espacios en blanco o, en algunos casos todos los espacios en blanco. Además identifica a los comentarios y los pasa por alto, e identifica los principios y finales de las cadenas constantes.

4.3. Análisis Sintáctico

4.3.1. Introducción

El análisis sintáctico determina la estructura del programa y de las expresiones individuales. Tomando el ejemplo de la sección 4.2.1, se determina que se trata de un bucle for, i se identifica como el contador del bucle, y se reconoce 1 y max como los límites del bucle. Se determina que el cuerpo del bucle se compone de una sola instrucción e identifica la expresión como una asignación.

Los lenguajes de programación se describen a través de gramáticas generativas, las cuales muestran cómo se pueden generar expresiones válidas. El diseño de un analizador sintáctico toma la gramática como punto de partida. Estas expresiones pueden representarse mediante los árboles de análisis.

Por ejemplo un programa que calcule la velocidad de una partícula

$$distancia = velocidad \cdot tiempo;$$

El analizador léxico separa los identificadores, los operadores, el punto y coma, y los entrega al análizador semántico separados en tokens de la forma

$$id_1 = id_2 \cdot id_3;$$

donde id_1es el token que corresponde al identificador distancia, id_2 corresponde al operador velocidad, e id_3 corresponde al tiempo. El analizador sintactico analiza esta secuencia de tokens, y produce un árbol de análisis representado en la siguiente figura

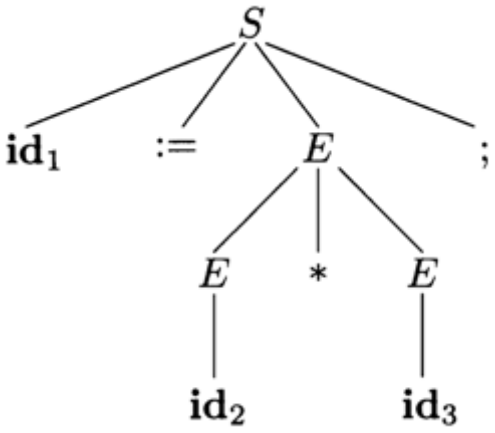

Figura 4.3.1: Ejemplo de Árbol de Análisis

El árbol muestra la estructura de la cadena de tokens. Este árbol dice que se trata de una declaración, que la declaración consta de un identificador, un operador de asignación, y una expresión, y que la expresión se compone de dos expresiones más, que son los identificadores, y un multiplicador.

4.3.2. Implementación

La implementación del análisis sintáctico implica el análisis de la estructura y sus elementos individuales, dado que en este proyecto los elementos que se manejan son elementos gráficos, los componentes que definen la estructura del programa son las Conexiones entre los Bloques Funcionales, ya que las conexiones indican el orden y la relación entre los elementos (bloques funcionales). Por lo tanto el analizar las conexiones y su orden es equivalente al análisis sintactico para este paradigma de programación gráfica.

Algoritmo de análisis de conexiones

Para analizar las conexiones involucradas en un diagrama es necesario verificar cada conector de cada elemento conectado hacia otro conector de otro elemento en particular.

Algoritmo 4.1 Análisis de Conexiones

```
/* Análisis de Conexiones */

1. Detectar conexiones.
2. Para cada conexión verificar elemento de origen, y elemento
    de destino.
3. Para cada conexión verificar conector de origen, y conector
    de destino.
4. Generar y Mostrar tabla con Resultados.
```

El código programado hace uso de la instrucción **foreach**, la cual significa "para cada"[6], la cual es ideal para la implementación eficiente del algoritmo en cuestión. El diagrama de secuencia del algoritmo implemento se muestra en la siguiente figura.

[6] Del inglés For Each.

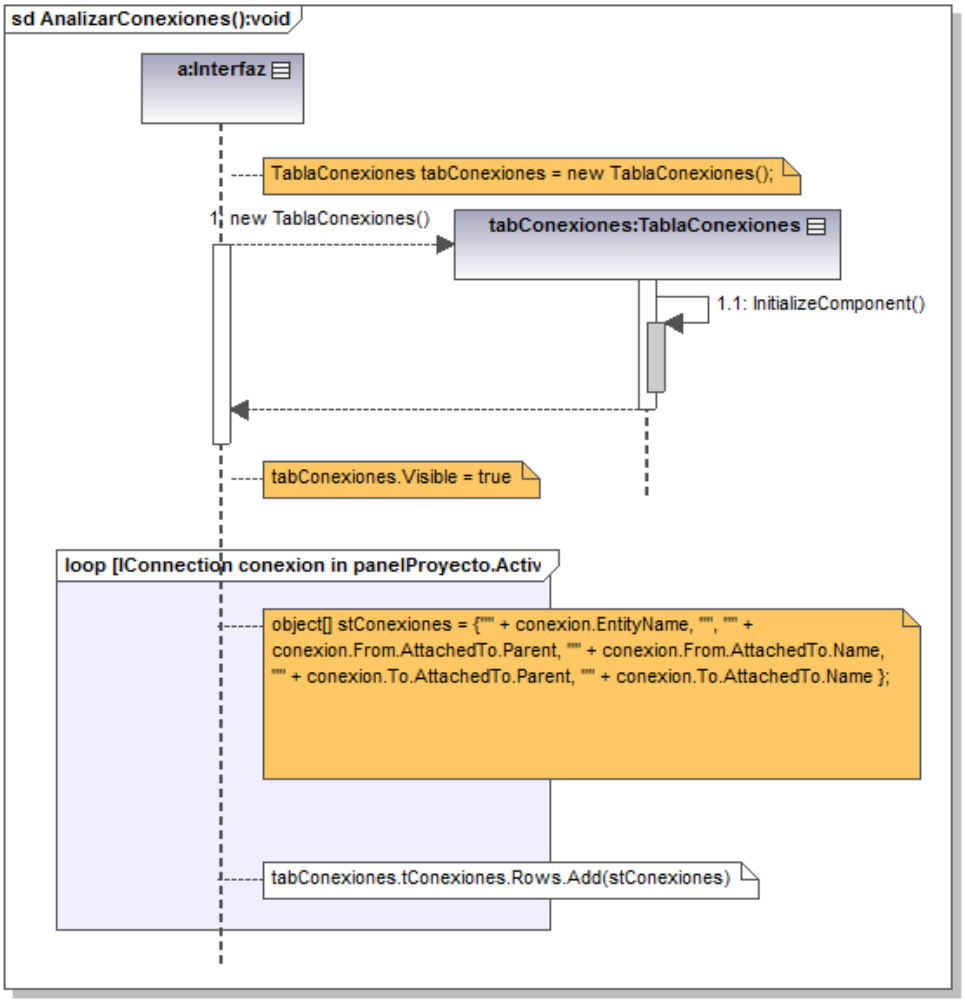

Figura 4.3.2: Análisis de Conexiones

4.4. Generación de Código Intermedio

4.4.1. Introducción

El generador de código intermedio crea una representación interna del programa que refleja la información revelada por el analizador sintáctico. Cada declaración en un lenguaje de alto nivel corresponde normalmente a varias instrucciones en lenguaje máquina, esto implica que en cierto punto la declaración tuvo que ser dividida en piezas pequñas que correspondan a esas instrucciones. Este proceso es llamado generación de código intermedio.

El código intermedio está a un nivel intermedio entre el alto nivel y lenguaje de máquina. Es una forma en la cual las piezas pequeñas de programa son claramente visibles, pero que todavía no es lenguaje de máquina o en muchos casos incluso no es lenguaje ensamblador. Esto se debe a la optimización está todavía por hacer, y lenguaje de máquina no es generalmente conveniente para los propósitos de optimización.

Hay varias representaciones utilizadas para el código intermedio, de los cuales el más importante y más utilizado es el código de tres direcciones. Las instrucciones en código de tres direcciones tiene la forma

$$resultado = [objeto][operador][objeto]$$

Por ejemplo en la declaración

$$T = distancia \cdot tiempo$$

Los objetos son las variables distancia y tiempo, el operador $\cdot$, y el resultado va en una variable temporal T. Las instrucciones en el código de tres direcciones son fáciles de generan bajo el control del analizador sintáctico, y son tan sencillos que pueden ser fácilmente traducidos a lenguaje máquina en el momento objeto de generación de código.

4.4.2. Implementación

Cada bloque funcional tiene información asociada, la cual define sus características y especificaciones. Esta información es almacenada en tablas, las cuales para propósitos de respaldo y manejo de proyecto son almacenados en archivos. Para la generación de código intermedio estas tablas requieren ser leídas a partir del archivo, y decodificadas nuevamente para obtener la información relacionada.

El algoritmo que permite obtener la información almacenada en las tablas de características de cada bloque funcional se muestran en el siguiente diagrama de secuencias

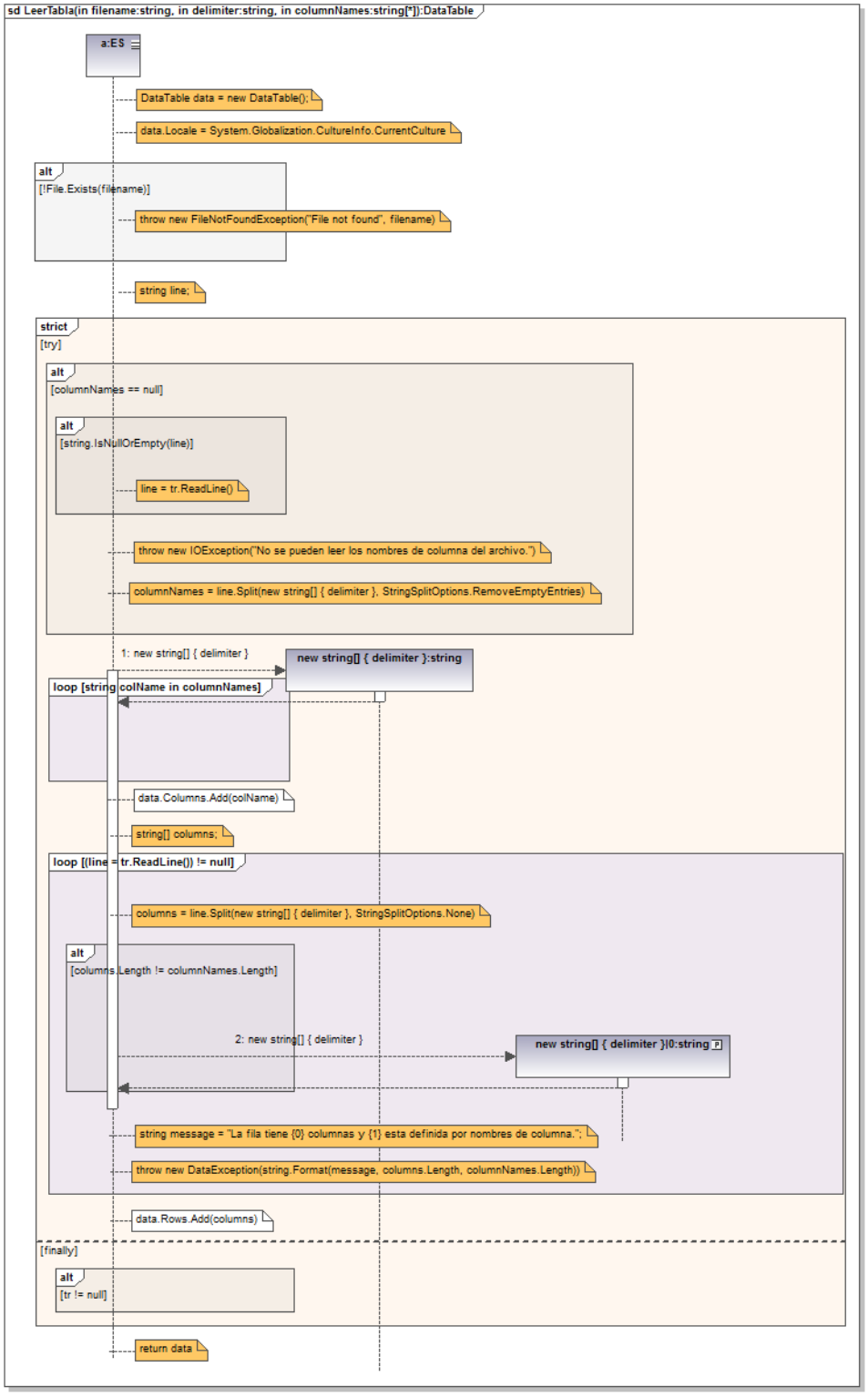

Figura 4.4.1: Lector de Tablas

Utilizando la información proporcionada por el algoritmo anterior, se puede generar código en lenguaje de programación C, apto para ser compilado a través del compilador C30 (basado en GCC[7]), utilizando una estructura de casos, la cual básicamente es una máquina de estados que asigna líneas de código C a cada una de las especificaciones contenidas (los cuales serían los estados) en la información proporcionada por las tablas. El algoritmo implementado se muestra en el siguiente diagrama de secuencias

[7] El compilador GCC (GNU Compiler Collection) es uno de los pilares del proyecto GNU, contiene una colección de compiladores de software libre con código abierto, además de herramientas de generación, que permiten generar automáticamente a partir de especificaciones, compiladores para dispositivos específicos.

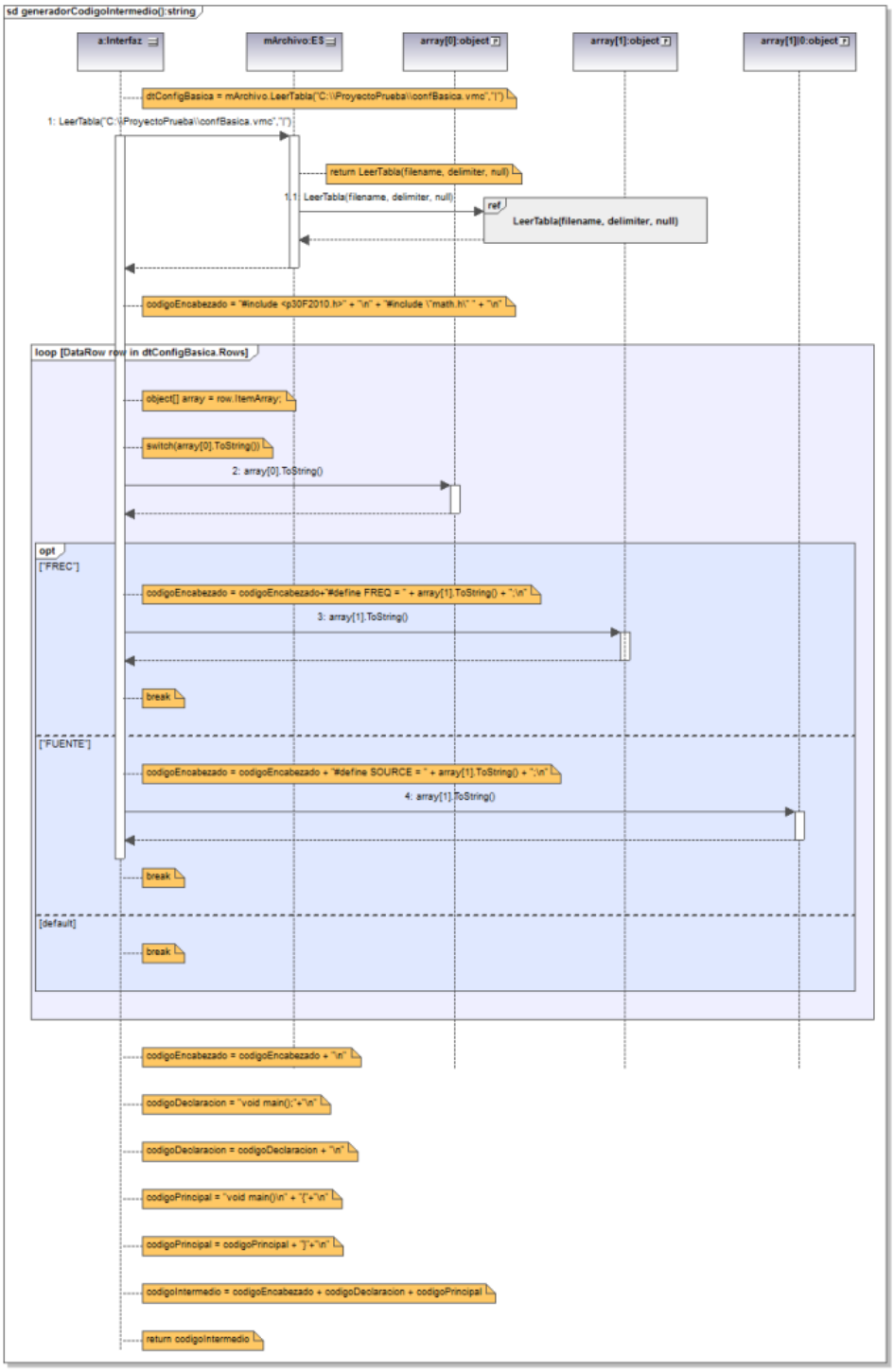

Figura 4.4.2: Generador de Código Intermedio

4.5. Generación de Código Objeto

4.5.1. Introducción

La generación de código objeto, la cual es por lo general llamada simplemente la generación de código, se encarga de traducir el código intermedio en el lenguaje de la máquina de destino. Por ejemplo, en un conjunto de instrucciones assembler, el código de tres direcciones de la ecuación $x = a \cdot y + z$ se traduce en

Algoritmo 4.2 Ejemplo de Código Objeto

```
LE  4,A Asignar variable A de punto flotante al registro 4
ME  4,Y Multiplicar por Y
AE  4,Z Sumar Z
STE 4,X Almacenar en X
```

Del mismo modo que las fases anteriores del proceso de compilación dependen del lenguaje, la fase generación de código es dependiente de la máquina. Esta es una de las partes más complejas de un compilador, ya que depende en gran medida del conjunto de instrucciones y arquitectura de la máquina objetivo. Las cuestiones a considerar en esta fase se asocian sobre todo con el orden en que se generan las instrucciones de la máquina y cómo se utilizan los registros de la máquina.

4.5.2. Interfaz entre Consola y Windows Forms

Para poder acceder a la consola de línea de comandos desde una aplicación con interfaz gráfica basada en Windows Forms es necesario implementar una interfaz que realize las siguientes funciones

- Iniciar el proceso o comando a ejecutar, con sus argumentos.
- Obtener la salida de este proceso a medida que se vaya ejecutando.

Para implementar esta funcionalidad se ha reutilizado la aplicación CommandLineHelper, adaptándo sus principales métodos a la aplicación.

Inicio del Proceso de Línea de Comandos

El método utilizado para iniciar el proceso, tomando en cuenta todos los parámetros y argumentos, se muestra en el siguiente diagrama

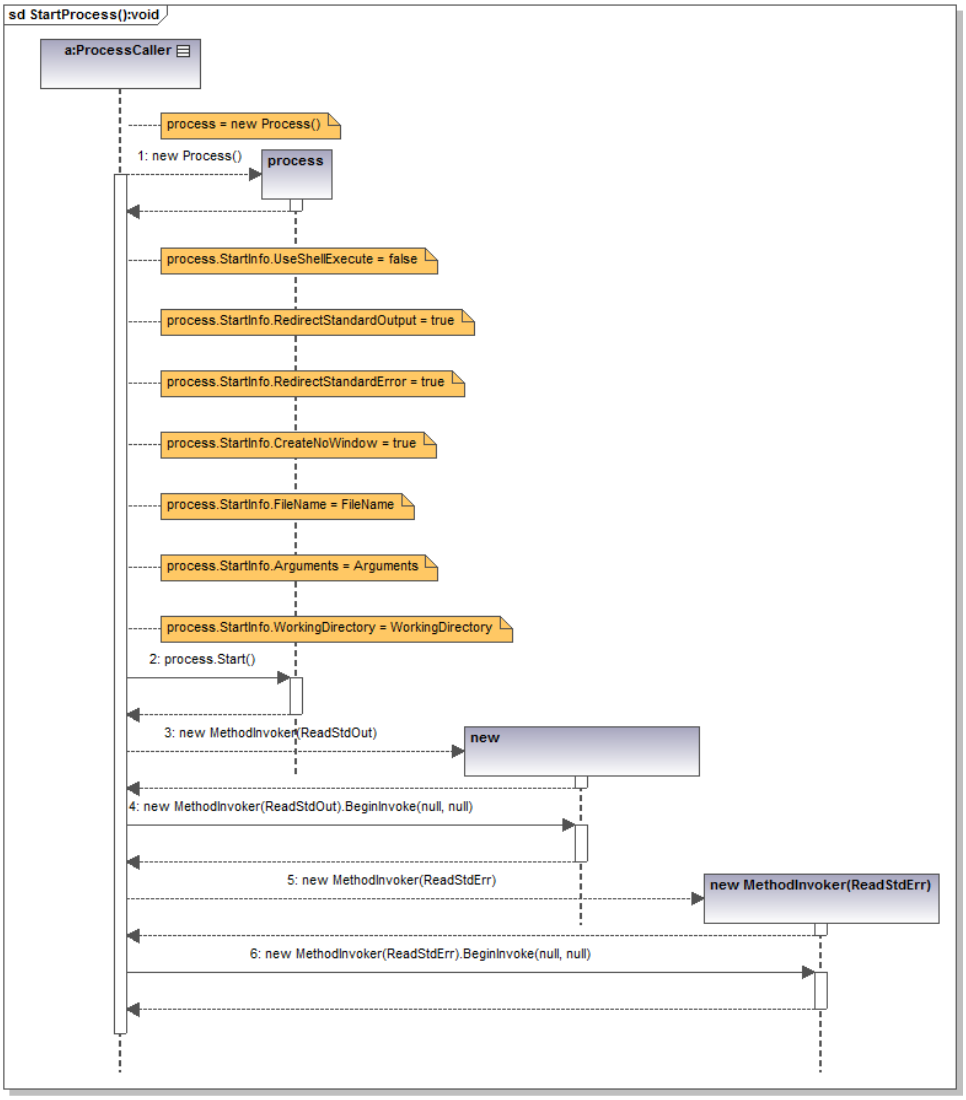

Figura 4.5.1: Inicio de Proceso de Línea de Comandos

Los parámetros más importantes que toma este método son:

- Filename: El nombre del archivo, especificando adecuadamente su ubicación (path[8] completo).

[8] El path o ruta es la especificación de unidad, directorio y subdirectorio en el cual se encuentra un archivo. Por ejemplo C:\Directorio\Subdirectorio\Archivo.Extension

- Arguments: Son los argumentos de entrada para la aplicación. Dependen de la aplicación, y sirven para indicar parámetros o especificaciones relacionadas con el proceso.

- Working Directory: Sirve para la implementación de paths relativos, lo que facilita que el programa sea portable y robusto.

Salida del Proceso

El método utilizado para obtener la salida del proceso se muestra en el siguiente diagrama

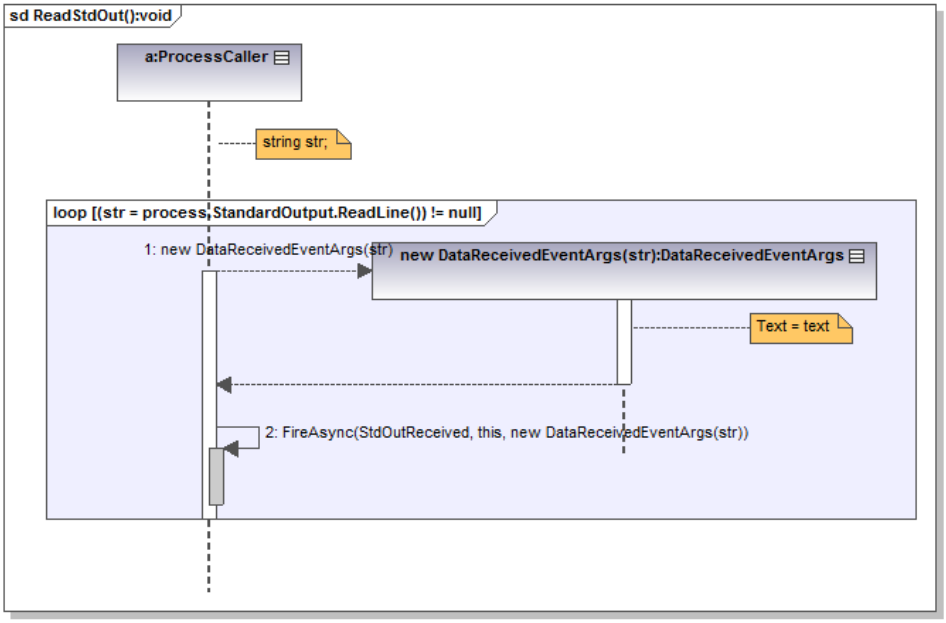

Figura 4.5.2: Salida del Proceso de Línea de Comandos

Este método dispara un evento asíncrono[9] cada vez que se recibe una salida del proceso de línea de comandos. Esta salida se almacena en forma de texto, el cual puede ser leído por la aplicación y mostrado en la interfaz de usuario.

[9] Un evento asíncrono al no tener sincronización y ser independiente de temporizaciónes, da robustez al programa, ya que no se presentarán desfases ni errores de sincronización.

4.5.3. Invocación a compilador C30 basado en GCC

El programa executable del compilador C30 es llamado pic30-gcc, este programa funciona bajo línea de comandos. La sintaxis del comando para invocar al compilador es la siguiente:

Algoritmo 4.3 Sintaxis Compilador C30

```
pic30-gcc [opciones] archivos
```

Por ejemplo para compilar un archivo simple, la instrucción de línea de comandos sería la siguiente:

Algoritmo 4.4 Ejemplo Compilador C30

```
pic30-gcc -o ejemplo.o ejemplo.c
```

La opción -o indica al compilador que la salida será un archivo de código objeto (con extensión .o), en la sintaxis se puede observar que primero se especifica el archivo de salida, y después el archivo (o archivos[10]) fuente.

Algoritmo para Invocar al Compilador C30

Dado que el ejecutable del compilador C30 funciona bajo línea de comandos, para invocarlo se requerirá de la interfaz expuesta en la sección anterior, utilizando los siguientes parámetros

- Filename: Path del compilador C30, seguido de la aplicación (pic30-gcc).
- Arguments: Opciones del compilador, de modo simple sería -o (específica extensión .o para código objeto), path del archivo de salida y path del archivo fuente.

El algoritmo que implementa esta llamada al compilador C30 se expone en el siguiente diagrama de secuencias

[10] Se puede compilar más de un archivo fuente, como sucede en el caso de que las funciones estén en archivos separados, o de que se esté compilando una librería.

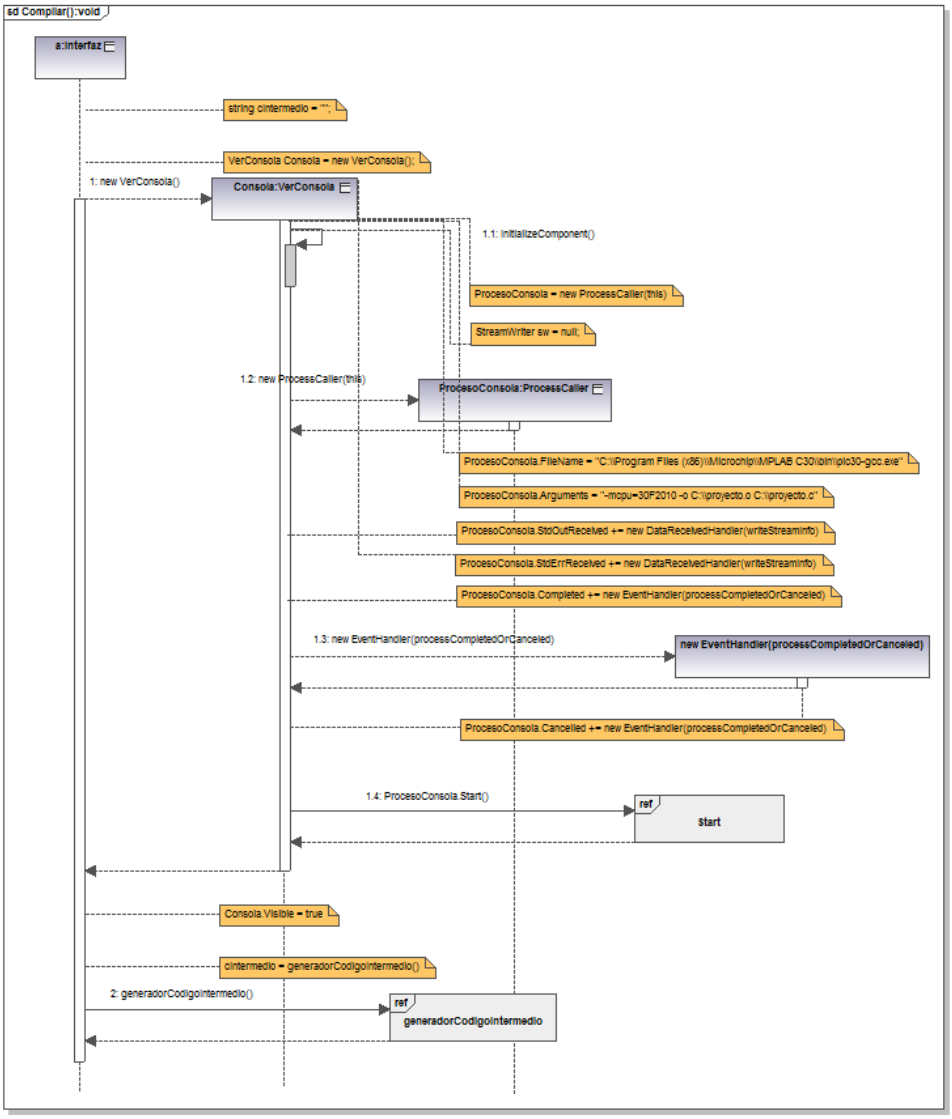

Figura 4.5.3: Invocacion a Compilador C30

4.6. Conversión de Código Objeto a Código Hex para microcontrolador

El procedimiento final para obtener código que pueda ser grabado sobre un microcontrolador es convertir el código objeto a código en formato hexadecimal (.hex[11])

[11] Este formato hexadecimal fue introducido por Intel, por ello también se le conoce como "Intel hex".

el cual puede ser grabado por cualquier programador que soporte el dispositivo microcontrolador que se esté utilizando.

Para convertir a código hexadecimal se puede utilizar una herramienta de conversión provista por Microchip, la cuál es pic30-bin2hex.

El funcionamiento de este conversor es sumamente simple, basta con especificar la dirección del código objeto fuente, por lo general esta herramienta no requiere de argumentos adicionales aparte de la dirección del archivo fuente.

Algoritmo 4.5 Sintaxis Conversor Hexadecimal

```
pic30-bin2hex archivo
```

Algoritmo para Conversión a Código Hexadecimal

Este algoritmo utiliza de igual forma la interfaz expuesta en la sección anterior, invocando a la herramienta mediante la sintaxis simple especificada. Los parámetros involucrados son

- Filename: Path del conversor hexadecimal, seguido de la aplicación (pic30-bin2hex).
- Arguments: Path del archivo fuente.

El diagrama de secuencias correspondiente se muestra en la siguiente figura

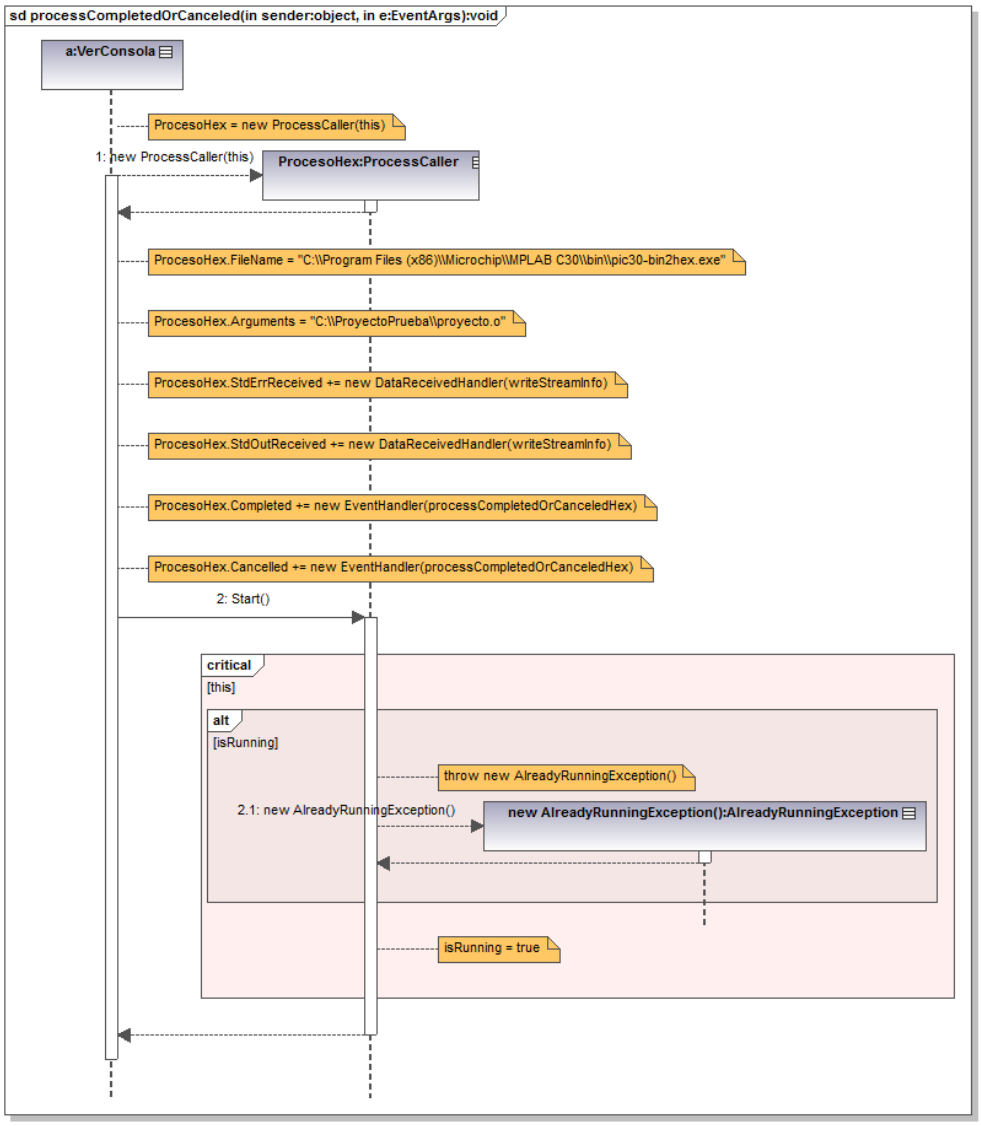

Figura 4.6.1: Conversión a Código Hexadecimal

Capítulo 5

GRABACIÓN DE DISPOSITIVOS: INTEGRACIÓN EN EL IDE

Este capítulo trata sobre los recursos disponibles para grabar los dispositivos con el código de la aplicación en formato hexadecimal (.hex), una vez que se ha compilado y generado a partir de los diseños realizados sobre la aplicación de programación gráfica.

El proceso de grabación es el último antes de que el microcontrolador este listo para realizar sus funciones en una aplicación práctica, por lo que se ha hecho un especial énfasis en facilitar dentro del entorno de programación todos los recursos y herramientas para este objetivo.

Se ha elegido al programador Pickit 2 como recurso para la grabación, ya que se tratar de una herramienta de uso libre (tanto el hardware, firmware y software), y además se dispone de las fuentes (diagramas de hardware, código abierto para el firmware y el software). Estas prestaciones hacen que esta herramienta sea idónea para el proyecto.

De igual forma se ha escogido una alternativa basada en Bootloader[1], esta herramienta es de software libre con código abierto, liberada bajo licencia GPL[2] v2.

5.1. Integración con programador Pickit 2

5.1.1. Características técnicas Pickit 2

El programador Pickit 2 (programmer-debugger) es una herramienta de desarrollo de bajo costo, con una interfaz fácil de usar para la programación y depuración de

[1] La funcionalidad del Bootloader se explica en la sección 5.2

[2] Este tipo de licencia se analiza en el capítulo 6.

microcontroladores Flash de Microchip.

Figura 5.1.1: Programador Pickit 2

Está abierto al público, incluyendo sus esquemas de hardware, el código fuente del firmware (en lenguaje C) y los programas de aplicación (en lenguaje C #). Los usuarios finales y terceras personas pueden modificar fácilmente el hardware y el software para obtener una funcionalidad mejorada. De igual forma existe una version GNU-Linux del software de aplicación Pickit 2, para Mac Os, y una herramienta para línea de comandos (CMD).

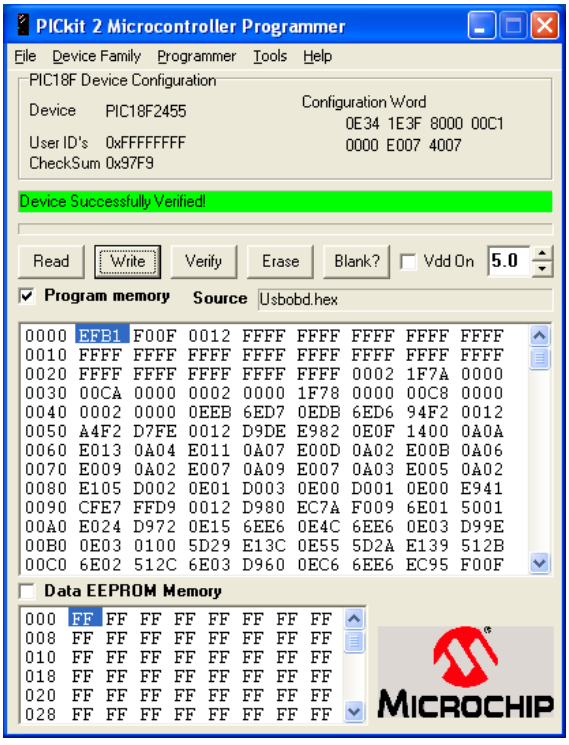

Figura 5.1.2: Herramienta Pickit 2 (GUI)

El programador Pickit 2 incluye la tecnología ICSP (In-Circuit Serial Program-

ming) la cual permite que sea posible utilizar el programador para programar los dispositivos sin necesidad de removerlos[3] del circuito o tarjeta en que se encuentren. Para facilitar la programación es recomendable diseñár el circuito de forma que incluya un header[4] o conector para la programación. Sin embargo debe tomarse en cuenta que característica no está destinada a la llamada programación de "producción"[5].

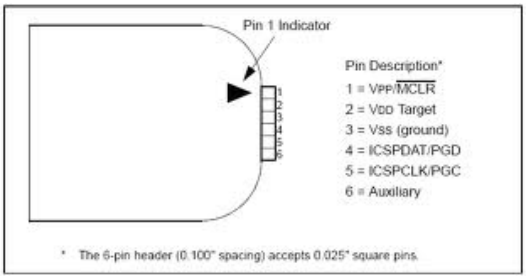

Figura 5.1.3: Header ICSP Pickit 2

El Pickit 2 utiliza internamente un PIC18F2550 con USB a velocidad completa. El firmware permite al usuario programar y depurar la mayor parte de los microcontroladores PIC de 8 y 16 bit, además de los dsPIC.

Figura 5.1.4: Circuito Interno Pickit 2

El Pickit 2 tiene una funcionalidad llamada “programmer to go”, la cual puede descargar el archivo .hex y las instrucciones de programación en la memoria interna del progrmador (128K bytes de memoria EEPROM I2C), por lo que bajo esta configuración no se necesita un computador para realizar el proceso de grabación.

El programador Pickit 2 tiene una herramienta UART, el cual es básicamente un terminal de comunicación serial.

[3] De ahí viene la denominación “In-Circuit”.

[4] Un header de 6 pines macho.

[5] Es decir programación que no esta diseñado para una programación masiva de dispositivos en una línea de producción.

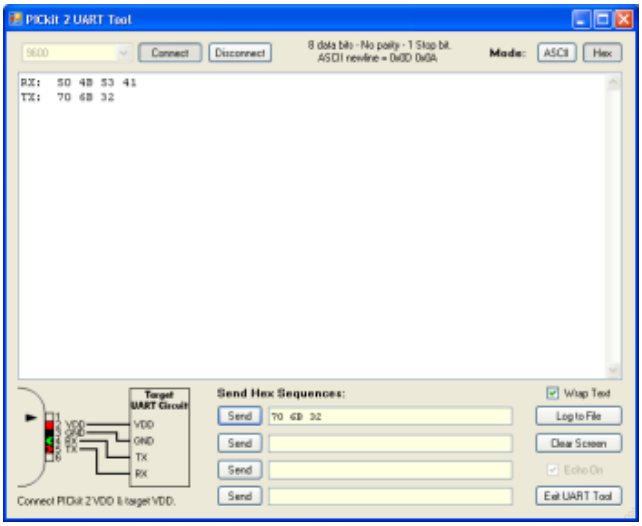

Figura 5.1.5: Herramienta Serial Pickit 2

Además, tiene un analizador lógico de tres canales y hasta 500 MHz.

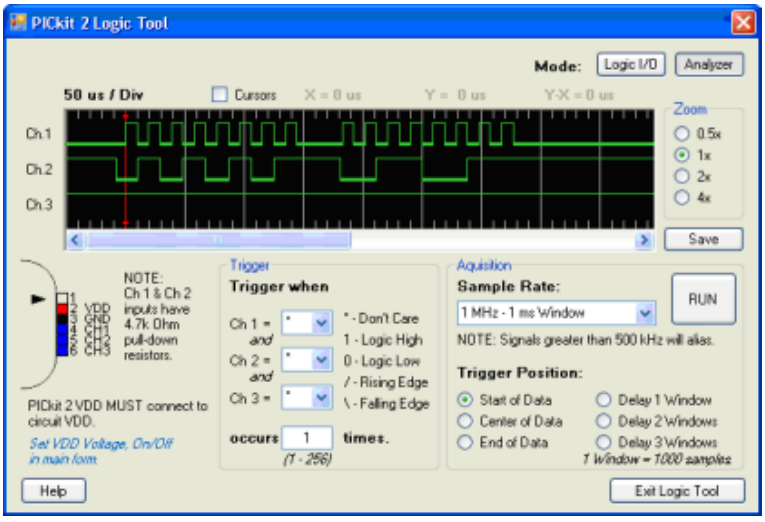

Figura 5.1.6: Analizador Lógico Pickit 2

5.2. Integración con Bootloader (ds30Loader)

5.2.1. Introducción Bootloader

En todo sistema computarizado, cuando se encienden la fuente de energía y esta se aplica a la tarjeta del procesador, muchos elementos de hardware deben ser inicializados antes de que se puede ejecutar cualquier programa. Cada arquitectura y procesador tiene un conjunto predefinido de acciones y configuraciones, que incluyen ejecutar una búsqueda de algún código de inicialización de un dispositivo de almacenamiento integrado (generalmente memoria Flash). Este código de inicialización temprana es conocido como Bootloader y es responsable de cargar la configuración inicial del procesador y los componentes de hardware relacionados.

La mayoría de los procesadores tienen una dirección predeterminada en la cual los primeros bytes del código se cargan cuando se enciende o reinicia el sistema. Los diseñadores de hardware de utilizan esta información para organizar la distribución de la memoria Flash y para seleccionar que rango de direcciones de la memoria Flash correspondiente. De esta manera, cuando el sistema se enciende, el código se carga a partir de una dirección conocida y previsible, y el software de control puede ser establecido.

El Bootloader proporciona el código de inicialización primario y es responsable de inicializar la placa de hardware de modo que otros programas se pueden ejecutar. Este código de inicialización primario es casi siempre escrito en lenguaje ensamblador nativo del procesador.

5.2.2. Características Técnicas ds30Loader

El ds30Loader es un Bootloader que soporta los microcontroladores Microchip de las familias PIC16, PIC18, PIC24 y dsPIC. Es compatible con todos los dispositivos en cada familia, requiriendo sólo pequeños ajustes en el firmware.

El ds30Loader consta de tres partes separadas:

- Firmware para el PIC, este es el programa que reside en el microcontrolador.
- Motor de Bootloader, esta parte contiene la funcionalidad para leer el archivo hexadecimal y comunicarse con el firmware.
- La aplicacion cliente, tanto en interfaz gráfica de usuario y aplicación de consola.

La organización de memoria en un microcontrolador utilizando ds30Loader esta representada en la siguiente figura

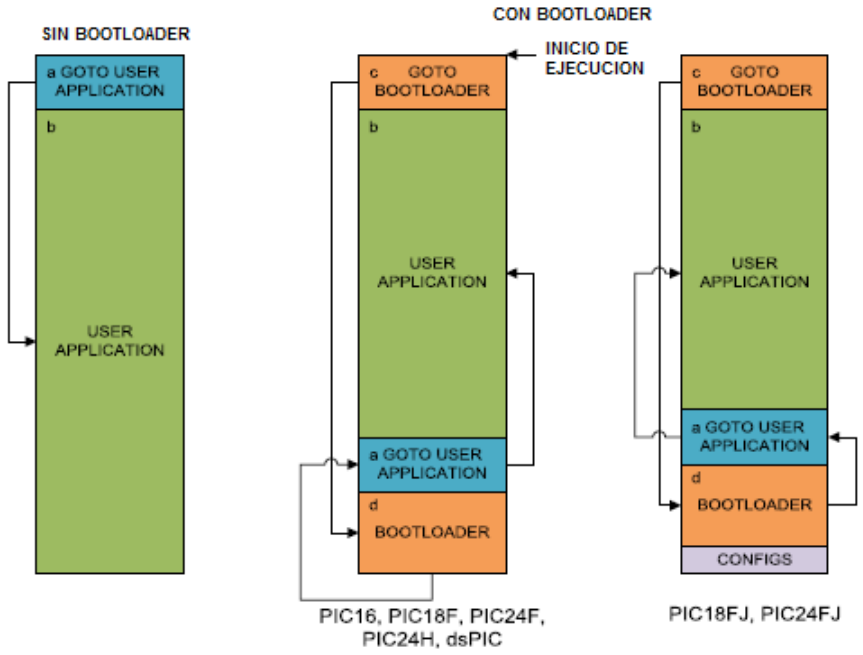

Figura 5.2.1: Organización de Memoria utilizando Bootloader

Características del Firmware

- Utiliza firmware único por familia de microcontroladores.
- Código altamente optimizado.
- No requiere de secuencia de comandos personalizados del linker[6].
- Tamaño pequeño (100 a 200 palabras).
- Control mediante checksum (suma de comprobación).
- Verificación de escritura.
- Protección de Bootloader.
- Escritura en la EEPROM.
- Configuración de escritura UART operación RS232 / RS485.
- Configuración automárica de velocidad de transferencia.

[6] El linker de MPLAB

Características del Motor de Bootloader

- Reinicio de dispositivo mediante dtr[7], rts o comando.
- Verifica el archivo hex para detectar código que pueda sobreescribir al bootloader.
- Verifica el archivo hex para detectar si hay un redireccionamiento de inicio[8].
- Configuración automática de velocidad de transmisión.
- Verificación mediante eco de transmisión.

[7] Dtr y rts son señales de control de la comunicación serial (UART).

[8] Es decir que detecta si se realiza un direccionamiento al inicio del programa en la dirección 0x00.

Capítulo 6

ASPECTOS DE LICENCIA

Este capítulo trata sobre los aspectos de licencia, aspectos legales y de propiedad intelectual, relacionados con el desarrollo y publicación de software. Estos aspectos son fundamentales en el proyecto, ya que es necesario especificar bajo que licencia se publicará el software desarrollado, y es de igual forma fundamental aclarar que recursos (software, hardware, firmware) se han utilizado[1], y si su utilización esta permitida y bajo que condiciones.

Con el fin de cumplir todos estos requerimientos se analizarán los aspectos de propiedad intelectual, los diferentes tipos de licencia, y el panorama en relación al software en el Ecuador.

6.1. Introducción

En esta sección se presentan los principales aspectos de licencia, en especial licencias de software libre. Para ponerlos en contexto, se empieza por una pequeña introducción a los conceptos más básicos de la propiedad intelectual e industrial, antes de exponer la definición detallada de software libre, software de fuente abierta y otros conceptos relacionados.

Se analizan también con cierto detalle las licencias de software libre más habituales y su impacto sobre los modelos de negocio y los modelos de desarrollo. También se comentarán licencias para otros recursos libres distintos del software.

[1] En el anexo A se especifica la licencia bajo la cual se han utilizado estas herramientas.

6.1.1. Introducción a la Propiedad Intelectual

Con el término propiedad intelectual se agrupan distintos privilegios que se otorgan sobre bienes intangibles con valor económico. De ellos podemos destacar los de copyright (derechos de autor) y similares, que protegen de la copia no autorizada los trabajos literarios o artísticos, programas de ordenador, recopilaciones de datos, diseños industriales, etc.; las marcas, que protegen símbolos; las indicaciones geográficas, que protegen denominaciones de origen; el secreto industrial, que respalda la ocultación de información, y las patentes, que otorgan monopolio temporal sobre un invento a cambio de desvelarlo. En muchas legislaciones, se distingue la propiedad intelectual, que se refiere a los derechos de autor, de la propiedad industrial, que abarca las figuras restantes.

En el ámbito internacional, la OMPI (Organización Mundial de la Propiedad Intelectual) promueve ambos tipos de propiedad en todos sus aspectos, mientras que el acuerdo TRIPS (Aspectos Comerciales de la Propiedad Intelectual) establece unos mínimos de protección y obliga a todos los países miembros de la OMC (Organización Mundial del Comercio) a desarrollarlos en unos plazos que dependen del nivel de desarrollo del país.

Derechos de Autor

Los derechos de autor (copyright) protegen la expresión de un contenido, no el contenido en sí mismo. Se desarrollaron para recompensar a los autores de libros o de arte. Las obras protegidas pueden expresar ideas, conocimientos o métodos libremente utilizables, pero se prohíbe reproducirlas sin permiso, total o parcialmente, con o sin modificaciones. Esta protección es muy sencilla, ya que entra automáticamente en vigor en el momento de publicación de la obra con ámbito casi universal. Modernamente, se ha extendido a los programas de ordenador y a recopilaciones de datos.

Éstos se dividen en derechos morales y patrimoniales. Los primeros garantizan al autor el control sobre la divulgación de su obra, con nombre o seudónimo, el reconocimiento de autoría, el respeto a la integridad de la obra y el derecho de modificación y retirada. Los segundos dan derecho a explotar económicamente la obra y pueden ser cedidos total o parcialmente, de forma exclusiva o no, a un tercero. Los derechos morales son vitalicios o indefinidos, mientras que los patrimoniales tienen una duración que depende de la regulación local.

La cesión de derechos se especifica por un contrato denominado licencia. En el caso de programas propietarios, éstos generalmente se distribuyen por medio de licencias de uso no exclusivo que se entiende que se aceptan automáticamente al abrir o instalar el producto. No es necesario pues firmar el contrato, ya que, en el caso de no aceptarlo el receptor, rigen automáticamente los derechos por omisión de la ley, es decir, ninguno. Las licencias no pueden restringir algunos derechos,

como el de hacer copias privadas de arte o música, lo que nos permite regalar una copia de una grabación a un amigo, pero este derecho no es aplicable a los programas.

Las nuevas tecnologías de la información, y en especial Internet, han trastocado profundamente la protección de los derechos de autor, ya que las expresiones de contenidos son mucho más fáciles de copiar que los contenidos mismos. Y en el caso de los programas y algunas obras de arte (música, imágenes, películas, e incluso literatura) funcionan automáticamente en el ordenador, sin necesidad de un esfuerzo humano apreciable. En cambio, los diseños o inventos hay que construirlos y, posiblemente, ponerlos en producción. Esta posibilidad de crear riqueza sin coste ha llevado a gran parte de la sociedad, en particular a los países pobres, a duplicar programas sin pagar licencia, no existiendo una conciencia social de que el actuar de este modo sea una mala acción (por contra, sí que la suele haber con respecto al robo de bienes físicos, por ejemplo). Por otro lado, los fabricantes de programas, solos o en coalición (p. ej.: la BSA o Business Software Alliance) presionan fuertemente para que las licencias se paguen y los gobiernos persigan lo que se ha dado en llamar "piratería".

Secreto Comercial

Otro de los recursos de que disponen las empresas para rentabilizar sus inversiones es el secreto comercial, protegido por las leyes de propiedad industrial, siempre que las empresas tomen las medidas suficientes para ocultar la información que no quieren desvelar. En el caso de productos químicos o farmacéuticos que requieran aprobación gubernamental, el Estado se compromete a no desvelar los datos entregados que no sea obligatorio hacer públicos.

Una de las aplicaciones del secreto comercial más conocidas se encuentra en la industria del software propietario, que generalmente comercializa programas compilados sin dar acceso al código fuente, para impedir así el desarrollo de programas derivados.

A primera vista parece que la protección del secreto comercial es perversa, ya que puede privar indefinidamente a la sociedad de conocimientos útiles. En cierto modo así lo entienden algunas legislaciones, permitiendo la ingeniería inversa para desarrollar productos sustitutos, aunque la presión de las industrias ha conseguido que en muchos países ésta sea una actividad prohibida y en otros sólo esté permitida en aras de la compatibilidad.

Sea perverso o no el secreto comercial, en muchos casos es mejor que una patente, ya que permite una ventaja competitiva al que pone un producto en el mercado, mientras la competencia trata de imitarlo con ingeniería inversa. Cuanto más sofisticado sea el producto, más costará a la competencia reproducirlo, mientras que si es trivial, lo copiará rápidamente. La imitación con mejoras ha sido fundamental para el desarrollo de las que hoy son superpotencias (Estados Unidos y Japón) y es muy importante para la independencia económica de los países en desarrollo.

Patentes

A cambio de un monopolio cuya duración varía entre los 25 a 100 años[2] y un determinado costo económico, un invento es revelado públicamente, de forma que sea fácilmente reproducible. Con la patente se pretende promover la investigación privada, sin coste para el contribuyente y sin que el resultado se pierda. El poseedor de una patente puede decidir si permite a otros utilizarla y el precio que debe pagar por la licencia.

Lo que se considera un invento ha ido variando con el tiempo, con grandes presiones para ampliar la cobertura del sistema, incluyendo algoritmos, programas, modelos de negocio, sustancias naturales, genes y formas de vida, incluidas plantas y animales. TRIPS exige que el sistema de patentes no discrimine ningún ámbito del saber. Las presiones de la Organización Mundial de la Propiedad Intelectual (OMPI o WIPO) pretenden eliminar la necesidad de que el invento tenga aplicación industrial y también rebajar los estándares de inventiva exigibles en una patente. Estados Unidos está a la cabeza de los países con un mínimo de exigencias sobre lo que es patentable, siendo además el más beligerante para que otros países adopten sus estándares.

Una vez obtenida una patente, los derechos del poseedor son independientes de la calidad del invento y del esfuerzo invertido en obtenerlo. Dado el coste de mantenimiento de una patente y los costes de litigación, solamente las grandes empresas pueden mantener y mantienen una amplia cartera de patentes que la sitúan en una posición que les permite ahogar cualquier competencia. Dada la facilidad para colocar patentes sobre soluciones triviales o de muy amplia aplicabilidad, pueden monopolizar para sí un espacio muy amplio de actividad económica.

Con patentes, muchas actividades, especialmente la programación, se hacen extremadamente arriesgadas, ya que es muy fácil que en el desarrollo de un programa complicado se viole accidentalmente alguna patente. Cuando dos o más empresas están investigando para resolver un problema, es muy probable que lleguen a una solución similar casi al mismo tiempo, pero sólo una (generalmente la de más recursos) logrará patentar su invento, perdiendo las otras toda posibilidad de rentabilizar su inversión. Todo desarrollo técnico complejo puede convertirse en algo extremadamente complejo si para cada una de las soluciones de sus partes es necesario investigar si la solución encontrada está patentada (o en trámite), para intentar obtener la licencia o para buscar una solución alternativa. Este problema deviene especialmente grave en el software libre, donde las violaciones de patentes de algoritmos son evidentes por simple inspección del código.

6.1.2. Intodrucción a las Licencias de Software

Estrictamente hablando, lo que diferencia al software libre del resto del software es un aspecto legal: la licencia. Se trata, en palabras de uso común, de un contrato

[2] La cantidad de años depende de la regulación de cada país.

entre el autor (o propietario de los derechos) y los usuarios, que estipula lo que los éstos pueden hacer con su obra: uso, redistribución, modificación, y en qué condiciones.

Aunque en esencia software libre y software propietario se diferencien en la licencia con la que los autores publican sus programas, es importante hacer hincapié en que las diferencias entre las diferentes licencias, aunque puedan parecer pequeñás, suelen suponer condiciones de uso y redistribución totalmente diferentes y, como se ha podido demostrar a lo largo de los últimos años, han desembocado no sólo en métodos de desarrollo totalmente diferentes, sino incluso en una forma alternativa de entender las tecnologías de la información.

Las condiciones y/o restricciones que imponen las licencias sólo pueden ser precisadas por los propios autores, que según la normativa de propiedad intelectual son los propietarios de la obra. En cualquier caso, la propiedad de la obra será de los autores, ya que la licencia no supone transferencia de propiedad, sino solamente derecho de uso y, en algunos casos, de distribución.

También es necesario saber que cada nueva versión de un programa es considerada como una nueva obra. El autor tiene, para cada versión, plena potestad para hacer con su obra lo que desee, incluso distribuirla con términos y condiciones totalmente diferentes (o sea, una licencia diferente a la anterior). De este modo, si se es autor único de un programa se podrá publicar una versión bajo una licencia de software libre y, si lo considerara conveniente, otra posterior bajo una licencia propietaria. En caso de existir más autores, y que la nueva versión contenga código cuya autoría les corresponda y que se vaya a publicar bajo otras condiciones, todos ellos han de dar el visto bueno al cambio de licencia.

Un tema todavía abierto, es conocer la licencia que cubre a las contribuciones externas[3]. Generalmente, se supone que una persona que contribuya al proyecto acepta tácitamente el hecho de que su contribución se ajuste a las condiciones especificadas por la licencia del proyecto.

Tomando en cuenta todos estos aspectos, se analizarán diversos tipos de licencias utilizados en la publicación de software.

6.2. Tipos de Licencias

6.2.1. Licencias tipo BSD

La licencia BSD (Berkeley Software Distribution) tiene su origen en la publicación de versiones de UNIX realizadas por la universidad californiana de Berkeley, en

[3]En un proyecto de software libre cuyo código este disponible, terceras personas pueden realizar contribuciones, añádiendo, mejorando, o adaptando el código ya existente.

EE.UU. La única obligación que exige es la de dar crédito a los autores, mientras que permite tanto la redistribución binaria, como la de loscódigos fuentes, aunque no obliga a ninguna de las dos en ningún caso. Asimismo, da permiso para realizar modificaciones y ser integrada con otros programas casi sin restricciones.

La licencia BSD es ciertamente muy popular, como se puede ver a partir del hecho de que existen varias licencias de condiciones similares (XWindow, Tcl/Tk, Apache), que se han venido a llamar licencias tipo BSD. Estas licencias reciben el nombre de minimalistas, ya que las condiciones que imponen son pocas, básicamente asignar la autoría a los autores originales. Su concepción se debe al hecho de que el software publicado bajo esta licencia era software generado en universidades con proyectos de investigación financiados por el gobierno de los Estados Unidos; las universidades prescindían de la comercialización del software creado, ya que asumían que ya había sido pagado previamente por el gobierno, y por tanto con los impuestos de todos los contribuyentes, por lo que cualquier empresa o particular podía utilizar el software casi sin restricciones, incluso redistribuyendo modificaciones al mismo de manera binaria sin tener que entregar las fuentes.

Este último punto hace que a partir de un programa distribuido bajo una licencia de tipo BSD pueda crearse otro programa (en realidad otra versión del programa) propietario, o sea, que se distribuyera con una licencia más restrictiva. Los críticos de las licencias BSD ven en esta característica un peligro, ya que no se garantiza la libertad de versiones futuras de los programas. Los partidarios de la misma, por contra, ven en ella la máxima expresión de la libertad y argumentan que, a fin de cuentas, se puede hacer casi lo que se quiera con el software.

Una de las consecuencias prácticas de las licencias tipo BSD ha sido la difusión de estándares, ya que los desarrolladores no encuentran ningún obstáculo para realizar programas compatibles con una implementación de referencia bajo este tipo de licencia. De hecho, ésta es una de las razones de la extraordinaria y rápida difusión de los protocolos de Internet y de la interfaz de programación basada en sockets, ya que la mayoría de los desarrolladores comerciales derivó su realización de la de la Universidad de Berkeley.

```
Esquema resumen de la licencia BSD: Copyright © el propietario. Todos
     los derechos reservados.

Se permite la redistribución en fuente y en binario con o sin
    modificación, siempre que se cumplan las condiciones siguientes:

1. Las redistribuciones en fuente deben retener la nota de copyright
    y listar estas condiciones y la limitación de garantía,

2. Las redistribuciones en binario deben reproducir la nota de
    copyright y listar estas condiciones y la limitación de garantía
    en la documentación.

3. Ni el nombre del propietario ni de los que han contribuido pueden
    usarse sin permiso para promocionar productos derivados de este
    programa.

ESTE PROGRAMA SE PROPORCIONA TAL CUAL, SIN GARANTÍAS EXPRESAS NI
    IMPLÍCITAS, TALES COMO SU APLICABILIDAD COMERCIAL O SU ADECUACIÓN
     PARA UN PROPÓSITO DETERMINADO. EN NINGÚN CASO EL PROPIETARIO
    SERÁ RESPONSABLE DE NINGÚN DAÑO CAUSADO POR SU USO (INCLUYENDO
    PÉRDIDA DE DATOS, DE BENEFICIOS O INTERRUPCIÓN DE NEGOCIO).
```

Figura 6.2.1: Esquema de Licencia tipo BSD

6.2.2. Licencia Pública General de GNU (GNU GPL)

La Licencia Pública General del proyecto GNU (más conocida por su acrónimo en inglés GPL) es la licencia más popular y conocida de todas las licencias del mundo del software libre. Su autoría corresponde a la Free Software Foundation (FSF) [4] y en un principio fue creada para ser la licencia de todo el software generado por la FSF. Sin embargo, su utilización ha ido más allá hasta convertirse en la licencia más utilizada, incluso por proyectos clave del mundo del software libre, como es el caso del núcleo Linux.

La licencia GPL es interesante desde el punto de vista legal porque hace un uso tan curioso de la legislación de copyright que haciendo estricto uso del término llega a una solución totalmente contraria a la original, hecho por el que también se ha venido a llamar una licencia copyleft.

En líneas básicas, la licencia GPL permite la redistribución binaria y la de las fuentes, aunque, en el caso de que redistribuya de manera binaria, obliga a que también se pueda acceder a las fuentes. Asimismo, está permitido realizar modificaciones sin restricciones, aunque sólo se pueda integrar código licenciado bajo GPL con otro código que se encuentre bajo una licencia idéntica o compatible, lo

[4] La FSF es la promotora del proyecto GNU

que ha venido a llamarse el efecto viral de la GPL, ya que el código publicado una vez con esas condiciones nunca puede cambiar de condiciones[5].

La licencia GPL está pensada para asegurar la libertad del código en todo momento, ya que un programa publicado y licenciado bajo sus condiciones nunca podrá ser hecho propietario. Es más, ni ese programa ni modificaciones al mismo pueden ser publicados con una licencia diferente a la propia GPL. Los partidarios de las licencias tipo BSD ven en esta cláusula un recorte de la libertad, mientras que sus seguidores ven en ello una forma de asegurarse que ese software siempre va a ser libre. Por otro lado, se puede considerar que la licencia GPL maximiza las libertades de los usuarios, mientras que las de tipo BSD lo hacen para los desarrolladores. Nótese, sin embargo, que en el segundo caso estamos hablando de los desarrolladores en general y no de los autores, ya que muchos autores consideran que la licencia GPL es más beneficiosa para sus intereses, ya que obliga a sus competidores a publicar sus modificaciones (mejoras, correcciones, etc.) en caso de redistribuir el software, mientras que con una licencia tipo BSD éste no tiene por qué ser el caso. De cualquier modo, podemos ver que la elección de licencia no es una tarea fácil y que hay que tener multitud de factores en cuenta.

En cuanto a la naturaleza contraria al copyright, esto se debe a que la filosofía que hay detrás de esta licencia (y detrás de la Free Software Foundation) es que el software no debe tener propietarios. Aunque es cierto que el software licenciado con la GPL tiene un autor, que es el que a fin de cuentas permite la aplicación de la legislación de copyright sobre su obra, las condiciones bajo las que publica su obra confieren a la misma tal carácter que podemos considerar que la propiedad del software corresponde a quien lo tiene y no a quien lo ha creado.

Por supuesto, también incluye negaciones de garantía para proteger a los autores. Asimismo, y para proteger la buena fama de los autores originales, toda modificación de un fichero fuente debe incluir una nota con la fecha y autor de cada modificación.

La GPL contempla también a las patentes de software, exigiendo que si el código lleva algoritmos patentados (algo legal y usual en Estados Unidos, y práctica irregular en Europa), o se concede licencia de uso de la patente libre de tasas, o no se puede distribuir bajo la GPL.

La licencia GPL se encuentra en la actualidad, y desde hace más de diez años, en su segunda versión y actualmente esta en la tercera versión. Generalmente, esto no supone mayor problema, ya que la mayoría de los autores suelen publicar los programas bajo las condiciones de la segunda versión de la GPL o cualquier posterior publicada por la Free Software Foundation.

[5] Una licencia es incompatible con la GPL cuando restringe alguno de los derechos que la GPL garantiza, ya sea explícitamente contradiciendo alguna cláusula, ya implícitamente, imponiendo alguna nueva. Por ejemplo, la licencia BSD actual es compatible, pero la original (o la de Apache), que exige en los materiales de propaganda que se mencione explícitamente que el trabajo combinado contiene código de todos y cada uno de los titulares de derechos, la hace incompatible.

Dada la extensión de la documentación de una licencia GPL[6], se muestra su índice de contenidos en Inglés[7], los cuales implementan todas las características de la licencia.

```
1. PREAMBLE

2. APPLICABILITY AND DEFINITIONS

3. VERBATIM COPYING

4. COPYING IN QUANTITY

5. MODIFICATIONS

6. COMBINING DOCUMENTS

7. COLLECTIONS OF DOCUMENTS

8. AGGREGATION WITH INDEPENDENT WORKS

9. TRANSLATION

10. TERMINATION

11. FUTURE REVISIONS OF THIS LICENSE
```

Figura 6.2.2: Estructura de una Licencia GPL

[6]Esta disponible una plantilla de la licencia GPL por parte de la Free Software Foundation (FSF), para su libre uso.

[7]Dada la universalidad planteada por la licencia, es recomendable publicar la licencia en este idioma.

Capítulo 7

RESUMEN, CONCLUSIONES Y RECOMENDACIONES

7.1. Resumen, Conclusiones y Recomendaciones

En el capítulo 1 se analizaron conceptos básicos de la programación en general, tales como automátas, estructuras condicionales, estructuras iterativas (bucles) y los tipos de variables. Se realizó una exposición concisa de los principales paradigmas tradicionales de programación, los cuales estan basados en líneas de código e instrucciones, a diferencia de la programación gráfica que se basa en sistemas visuales. Se expuso la diferencia entre los lenguajes de alto y bajo nivel, su relación con la abstracción del hardware y con la forma de interpretar la información para los usuarios. Posteriormente se analizaron aspectos de arquitectura de software, en particular se analizó el modelo de desarrollo en cascada, el cual consiste en iterar el proceso de desarrollo hasta llegar al modelo definitivo del software; se expusieron también factores de calidad en software, los cuales se tuvieron en consideración para el desarrollo del proyecto. Las conclusion de este capítulo es: Debido a que el nivel de abstracción planificado para el proyecto es bastante alto se requiere facilitar al maximo la ocultación del hardware del microcontrolador, sin embargo se requiere que el potencial usuario tenga conocimientos elementales de programación y del funcionamiento básico de un microcontrolador.

En el capítulo 2 se expuso brevemente el .NET framework, el cual es el entorno de trabajo que se utilizó para la aplicación. Se analizó de forma concisa lenguaje de programación C# elegido por sus características. Posteriormente se analizó la herramienta integrada de desarrollo Visual C# 2010 Express Edition, la cual es de libre uso y presta funcionalidades que facilitaron el desarrollo de la aplicación. Posteriormente se estudiaron las herramientas para interfaz gráfica basada en Windows Forms. Finalmente se expuso el resultado de la implementación de la interfaz gráfica, la cual se implementó a través de los elementos expuestos en

el capítulo. Las conclusion a las que se llegó es: Se ha elegido utilizar el lenguaje de programación C#, debido a las prestaciones que ofrece para el desarrollo de aplicaciones gráficas, gracias a la potencia del .NET framework.

En el capítulo 3 se analizó la librería gráfica Netron, la cual es el motor gráfico que potenció la aplicación desarrollada; se indicaron sus principales características y su estructura a través de un diagrama. Posteriormente se analizaron los bloques funcionales, los cuales tienen un aspecto gráfico el cual se realizó a través de la herramienta de software libre Inkscape, utilizando imágenes basadas en formato vectorial. Finalmente se analizó el aspecto funcional y lógico del bloque funcional, a través de un análisis detallado de su implementación, estudiando el código implementado y exponiendo las técnicas utilizadas. La conclusión de este capítulo es: El motor gráfico utilizado se eligió a partir de diversas opciones, siendo la mayor parte de estas basadas en Java y con código abierto pero con derechos de utilización y distribución restringidas, de modo semejante se analizaron diversas opciones basadas en C#, emergiendo como alternativa principal la librería Netron, debido a que esta liberado bajo licencia GPL y se dispone de documentación.

En el capítulo 4 se estudió el proceso de Compilación, tomando en cuenta los procesos de análisis involucrados, tanto para lenguajes de programación tradicionales como su implementación en el marco del paradigma de programación gráfica. Posteriormente se analizó la generación de código objeto a través de la invocación al compilador C30 basado en GCC. Finalmente se expuso el método para convertir el código objeto a código hex, el cual se puede grabar sobre el dispositivo. Las conclusiones de este capítulo son:

- Se ha implementado un proceso compilacion para sistemas gráficos, basado tanto en sistemas tradicionales de programación como en paradigmas modernos, basados en aspectos visuales y representación de los mecanismos de diseño y pensamiento de los usuarios.
- El conjunto de compiladore GCC es la principal herramienta de compilación para aplicaciones de software libre y comerciales, debido a la amplitud del soporte de dispositivos objetivo (target) y a que es una herramienta con licencia GPL.

En el capítulo 5 se analizó la grabación de dispositivos, en particular se estudió el uso de un programador serial, asi como su funcionalidad y características. Finalmente se expuso la opcion de Bootloader para programar los dispositivos sin necesidad de hacer una programación física (soft programming). Las conclusiones de este capítulo son:

- Se eligió el programador Pickit debido a que es un dispositivo de código abierto y de libre distribución, por lo que este recurso es accesible para los usuarios independientemente de si se trata de estudiantes o profesionales.

- De manera semejante, se escogió al Bootloader ds30Loader como alternativa para realizar soft programming, gracias a su tecnología de grabación a través del puerto serial, y debido a que su licencia es libre y de código abierto.

En el capítulo 6 se estudió el importante tema de las licencias para software, haciendo primero una introducción a la propiedad intelectual, al secreto comercial y a las patentes. También se analizaron aspectos generales de las licencia de software. Posteriormente se estudó la licencia BSD, la cual es una licencia de software libre permisiva, de modo que ofrece alternativas y recursos en caso de que se requiera cambiar el tipo de licencia. Finalmente se estudió la licencia GPL la cual es una de las licencias de software libre más rigurosas desde el punto de vista de la propiedad intelectual y disponibilidad de código, ya que el código y software licenciado bajo GPL debe permanecer como tal sin ninguna alternativa de cambio. La conclusión de este capítulo es:

- Considerando las características de las licencias, se optó por una licencia tipo BSD, para de este modo no tener ninguna restricción en caso de que se desee realizar una aplicación comercial del software desarrollado.

ANEXOS

Anexo A: Especificación de Licencias de Herramientas Utilizadas

Software

Nombre	S. O.	Autor	Licencia	Open Source	Observaciones
Visual C# 2010 Express Edition	Win32	Microsoft Inc.	Freeware	No	IDE usado en el desarrollo del proyecto.
Netron	Win32/64, Linux	Francoise Vanderseipen PhD	GPL v2.0	Si	Librería gráfica usada en el proyecto.
Pickit 2 Application	Win32, Linux, Mac OS	Microchip Inc.	Propietaria	Si	Aplicación recomendada para grabar dispositivos.
Pickit 2 CMD	Línea de Comandos	Microchip Inc.	Propietaria	Si	Aplicación integrada en el proyecto.
Inkscape	Win32/64, Linux	Inkscape.org	GPL v2.0	Si	Herramienta utilizada para diseño gráfico de bloques funcionales.
ds30Loader	Win32/64, Linux	Mikael Gustaffson	GPL v2.0	Si	Bootloader recomendado e integrado en el proyecto
Lyx	Win32/64, Linux	Lyx.org	GPL v2.0	Si	Editor LaTeX para documentación del proyecto.

Hardware

Nombre	Autor	Licencia	Observaciones
Pickit 2 Programmer/Debbuger	Microchip Inc.	Propietaria	Esquemas y Especificaciones del Hardware del programador Pickit2.

Firmware

Nombre	Autor	Licencia	Observaciones
Pickit 2 Firmware	Microchip Inc.	Propietaria	Firmare del programador pickit 2.
ds30Loader Bootloader Firmware	Mikael Gustaffson	GPL v2.0	Firmware del Bootloader (programado en el microcontrolador).

Anexo B: Diagramas UML Principales del Proyecto Visual Microcontroller

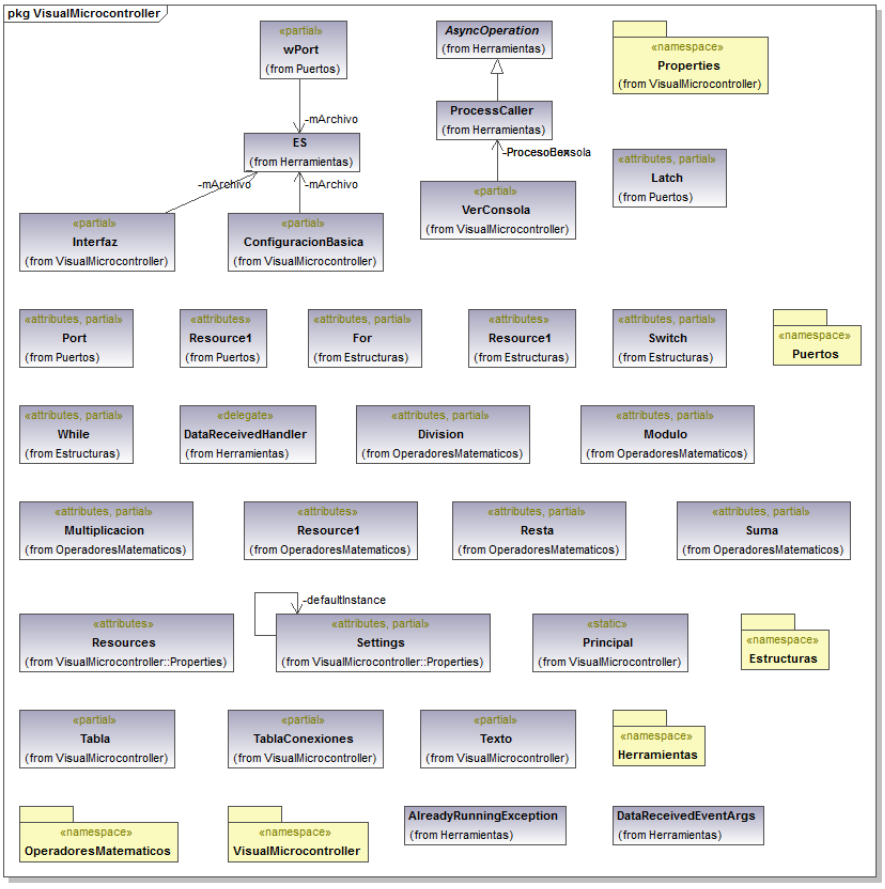

Figura A.1 Diagrama de Paquetes Visual Microcontroller

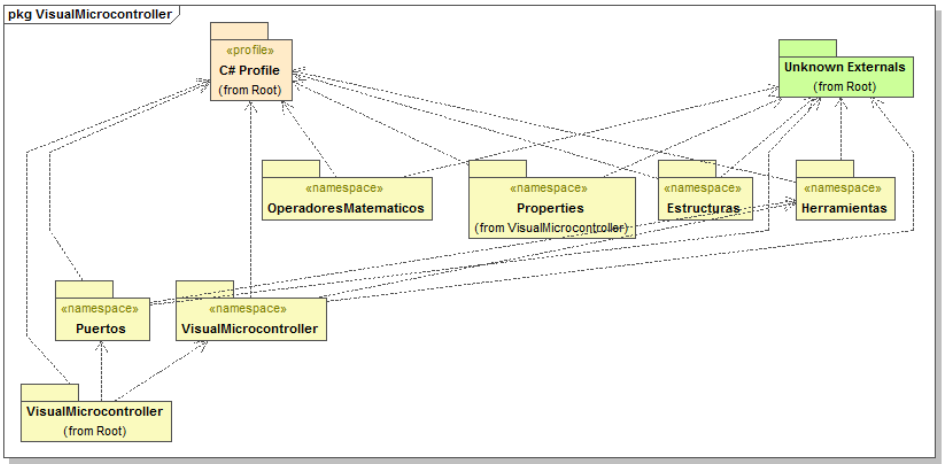

Figura A.2 Diagrama de Dependencia Visual Microcontroller

Anexo C: Licencia BSD del Proyecto Visual Microcontroller

Licence of Visual Microcontroller

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of autor nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Bibliografía

[1] Ian Sommerville. *Ingeniería de Software.* Septima Edición. Pearson Educación. 2005.

[2] Keneth Rosen. *Matemáticas Discretas y sus Aplicaciones.* Quinta Edición. McGraw Hill.

[3] Tomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest. *Introduction to Algorithms.* First Edition. The MIT Press. 2000.

[4] Karli Watson, Christian Nagel, Jacob Pedersen, Jon Reid, Morgan Skinner. *Beginning Visual C# 2010.* Wiley Publishing. 2010.

[5] Paul Deitel, Harvey Deitel. *C# 2010 for Programmers.* Fourth Edition. Pearson Education 2011.

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Priciples, Techniques, & Tools.* Pearson. Second Edition. 2007.

[7] Daniel P. Friedman, Mitchel Wand, Christopher T. Haynes. *Essentials of Programming Languages.* Second Edition. 2001.

[8] *dsPic30F2010 Family Reference Manual.* Microchip Inc.

[9] Ricardo Mazza. *Introduction to Information Visualization.* Springer-Verlag. 2009.

[10] Francoise Vanderseipen. *Netron Library Architecture. 2003.*

[11] Francoise Vanderseipen. *Netron Library White Paper. 2004.*

[12] Brian W. Kernighan, Dennis M. Ritchie. *El lenguaje de programación C.* Segunda Edición. Pearson Educación. 1991.

[13] Brian Berenbachm Daniel J. Paulish, Juergen Kazmeier, Arnold Rudorfer. *Software & Systems Requirements Engineering In Practice.* McGraw-Hill. 2009

[14] Mike Mayer. *Launching a Process and Displaying its Standard Output.* www.codeproject.com.

[15] Margaret Burner, Marla Baker. *Visual Language Research.* Computer Science Department. Oregon State University. web.engr.oregonstate.edu/~burnett/vpl.html.

[16] Des Watsin. *High Level Languages and Their Compilers.* Addison-Wesley. 1999.