Máster en Ingeniería de Sistemas y Control

Algoritmia de Sensor Estelar para CubeSat

(Modelado y Simulación)

Autor: Raúl Argüelles Villanueva

Director: José María Girón Sierra

Curso 2014-2015. Convocatoria de Febrero

Máster en Ingeniería de Sistemas y Control

Algoritmia de Sensor Estelar para CubeSat

(Modelado y Simulación)

Proyecto Tipo B:

Proyecto específico propuesto por el alumno

Autor: Raúl Argüelles Villanueva

Director: José María Girón Sierra

Autorización

Autorizamos a la Universidad Complutense y a la UNED a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firmado:

Raúl Argüelles Villanueva

RESUMEN

Los satélites CubeSat determinan su actitud de diversas maneras. En este proyecto se va a desarrollar la algoritmia necesaria para el cálculo de la actitud de un Cubesat por medio un sensor estelar o *Star Tracker,* consistente en la determinación de las estrellas presentes en la imagen tomada por una cámara incorporada al CubeSat, y su posterior comparación con un catálogo estelar previamente guardo en la memoria del satélite.

Vamos estudiar la algoritmia del sensor estelar, implementarla en Matlab y, posteriormente, modelar y simular el sistema propuesto por Miguel Ángel López Suesma [11], como proyecto antecesor de este. De este modelado y simulaciones, para las cuales generaremos imágenes sintéticas con una actitud aleatoria conocida, y el posterior análisis de los resultados obtenidos, sacaremos las conclusiones oportunas, donde veremos la precisión del sistema propuesto.

Por último, basado en el sistema modelado anteriormente, crearemos un sistema real de pruebas con elementos hardware de prototipado low-cost, pero con unas características muy similares al modelo anterior, en las que podremos probar nuestro hardware y software, antes de crear un prototipo real del sistema modelado, ahorrando tiempo, dinero y esfuerzo al resultado final del sensor estelar del CubeSaT.

PALABRAS CLAVES

- CubeSat, Sensor Estelar, Star Tracker, Método Voting, Centroides.
- Catálogo Estelar, Sistemas de Coordenadas, Coordenadas Sistema Inercial, Conversión de Coordenadas,
- Cálculo de la Actitud, Control de la Actitud.
- Imágenes Sintéticas, Modelado, Simulación.
- Chipkit Uno32, PIC32, RAM, EEPROM, Cámara, Lente.
- Algoritmo, MATLAB, MPIDE.

ÍNDICE

1.- INTRODUCCIÓN..... 1

1.1.- SENSOR ESTELAR..... 1

1.2.- SISTEMA DE PRUEBAS..... 2

1.3.- ESTRUCTURA DEL PROYECTO..... 2

2.- ALGORITMOS..... 4

2.1.- INTRODUCIÓN..... 4

2.2.- CENTROIDING..... 4

2.3.- STAR IDENTIFICATION..... 9

2.3.1.- ESTUDIO DE ALGORITMOS..... 9

2.3.1.1.- TÉCNICA DE IDENTIFIACIÓN PIRAMIDAL..... 9

2.3.1.2.- MÉTODO VOTING..... 11

2.3.1.3.- TÉCNICA DE IDENTIFIACIÓN POR TRIÁNGULO PLANAR..... 11

2.3.1.4.- ALGORITMO GRID..... 13

2.3.1.5.- COMPARACIÓN DE MÉTODOS..... 14

2.3.2.- MÉTODO VOTING..... 16

2.3.2.1.- GENERACIÓN DEL CATÁLOGO..... 17

2.3.2.2.- CANDIDATE MATCHING..... 18

2.3.2.3.- VERIFICACIÓN Y RESULTADO FINAL..... 19

2.4.- DETERMINACIÓN DE LA ACTITUD..... 20

3.- ARQUITECTURA SOFTWARE..... 22

3.1.- FLUJO SOFTWARE..... 22

3.2.- FUNCIONES..... 23

3.2.1.- FUNCIÓN *catinit*..... 23

3.2.2.- FUNCIÓN *loading*..... 25

3.2.3.- FUNCIÓN *centroid*..... 26

3.2.4.- FUNCIÓN *uvec*..... 27

3.2.5.- FUNCIÓN *starid*..... 28

3.2.6.- FUNCIÓN *adet*..... 31

4.- RESULTADOS SOFTWARE..... 33
4.1.- CONFIGURACIÓN HW..... 33
4.1.1.- SENSOR DE IMAGEN..... 34
4.1.2.- LENTE..... 35
4.1.3.- MICROPROCESADOR..... 37
4.1.4.- CATÁLOGO DE ESTRELLAS Y MEMORIA FLASH..... 38
4.1.5.- MEMORIA RAM..... 42
4.2.- SIMULACIONES..... 43
4.2.1.- RANDOM ATTITUD..... 43
4.2.2.- IMAGE GENERATION..... 44
4.2.3.- EJECUCIÓN DEL ALGORITMO DEL STAR TRACKER..... 49
4.3.- ANÁLISIS Y CONCLUSIONES..... 51

5.- SISTEMA DE PRUEBAS..... 57
5.1.- INTRODUCCIÓN..... 57
5.2.- HARDWARE..... 57
5.2.1.- PROCESADOR..... 59
5.2.2.- CÁMARA Y LENTE..... 62
5.2.3.- RAM..... 75
5.2.4.- EEPROM..... 77
5.3.- SOFTWARE..... 78
5.3.1.- CONTROL DE LA CÁMARA..... 81
5.3.2.- CONTROL DE LA RAM..... 89
5.3.3.- CONTROL DE LA EEPROM..... 92
5.3.4.- ALGORTIMOS DEL SENSOR ESTELAR..... 97
5.3.5.- SOFTWARE DE CONTROL EN PC..... 100

6.- CONCLUSIONES.....105

7.- LÍNEAS ABIERTAS..... 106

8.- BIBLIOGRAFÍA..... 107

9.- ANEXOS..... 109

LISTADO DE FIGURAS

2.- ALGORITMOS

Figura 2.1: Imagen con estrellas muy enfocadas..... 5
Figura 2.2: Imagen con estrellas desenfocadas..... 5
Figura 2.3: Pixels de borde para $a_{ROI} = 7$..... 7
Figura 2.4: Vector unitario de una estrella..... 8
Figura 2.5: Pirámide de estrellas para las estrellas candidatas *i, j, k,* y estrella de referencia *r*..... 10
Figura 2.6: Descripción de los pasos del algoritmo de *grid*..... 14
Figura 2.7: Diagrama de bloques del método *Voting*..... 16

3.- ARQUITECTURA DE SOFTWARE

Figura 3.1: Diagrama de flujo del sensor estelar..... 23
Figura 3.2: Diagrama de flujo de la función *catinit*..... 24
Figura 3.3: Diagrama de flujo de la función *centroid*..... 26
Figura 3.4: Diagrama de flujo de la función *starid*..... 28
Figura 3.5: Candidate Matching..... 30
Figura 3.6: Verificación final..... 31

4.- RESULTADOS SOFTWARE

Figura 4.1. Diagrama de bloques del sensor estelar..... 33
Figura 4.2. Diagrama de bloques del sensor KODAK KC-9618..... 35
Figura 4.3. Lentes del KODAK KAC-96-1/3” LENS KIT..... 36
Figura 4.4. Diagrama de bloques del PIC32..... 37
Figura 4.5. Sistema de coordenadas ecuatoriales..... 40
Figura 4.6. Acceso a memoria SRAM CY7C1061..... 42
Figura 4.7. DEC vs. RA para límite magnitud = 3,75..... 45
Figura 4.8. DEC vs. RA para límite magnitud = 5..... 46
Figura 4.9. Ejemplo de imagen generada con los IDs de las estrellas..... 49
Figura 4.10. Número de casos por cada número de estrellas identificadas …..... 53
Figura 4.11. Número de casos por cada error en el cálculo de la actitud..... 54
Figura 4.12. Error en la actitud por número de estrellas identificadas y eje..... 54

Figura 4.13. Error promedio en la actitud por número de estrellas identificadas..... 55
Figura 4.14. Izda: Imagen 20; Dcha: Imagen 22..... 56

5.- SISTEMA DE PRUEBAS

Figura 5.1. Esquema General del Hardware de Pruebas..... 58
Figura 5.2.- Aspecto del CHIPKIT UNO32 y sus características principales..... 59
Figura 5.3.- Pines del CHIPKIT UNO32..... 60
Figura 5.4.- Conexión maestro-esclavos en bus I2C. R_p = 4,7 KΩ..... 64
Figura 5.5.- Sincronización horizontal..... 65
Figura 5.6.- Timing VGA..... 67
Figura 5.7.- Timing QVGA..... 67
Figura 5.8.- OV7670 sin FIFO..... 68
Figura 5.9.- Acceso a entradas y salidas digitales mediante funciones MPIDE..... 69
Figura 5.10.- Acceso a entradas y salidas digitales mediante acceso directo a puertos PIC32..... 70
Figura 5.11.- OV7670v3 + FIFO..... 70
Figura 5.12.- Características AL422B..... 71
Figura 5.13.- Esquema OV7670v3 + FIFO..... 72
Figura 5.14.- Esquema conexión OV7670v3 + FIFO a Uno32..... 74
Figura 5.15.- Juego de lentes Cognex para sensores de 1/6”..... 74
Figura 5.16.- FOV vs distancia de trabajo para las lentes Cognex..... 75
Figura 5.17.- Pines RAM 23LC1024..... 76
Figura 5.18.- Esquema conexión RAM a Uno32..... 76
Figura 5.19.- Esquema bus SPI multi-esclavo..... 77
Figura 5.20.- Pines EEPROM 24LC256..... 77
Figura 5.21.- Pines EEPROM 24FC1024..... 78
Figura 5.22.- Esquema conexión EEPROM a Uno32..... 78
Figura 5.23.- Aspecto del MPIDE..... 79
Figura 5.24.- Fichero ov7670.h..... 81
Figura 5.25.- Byte de control I2C..... 82
Figura 5.26.- Código para iniciar y configurar cámara OV7670..... 83
Figura 5.27.- Resumen de código función *Set_Camera_Config*..... 83
Figura 5.28- Registro COM7 de la cámara OV7670..... 84
Figura 5.29- Registro COM10 de la cámara OV7670..... 84
Figura 5.30- Código función CaptureImage_to_FIFO..... 85

Figura 5.31- Timing de señales para escritura en FIFO..... 86
Figura 5.32- Código función GetImage_FIFO_to_RAM..... 87
Figura 5.33- Código función ReadOneByte..... 87
Figura 5.34- Timing de señales para lectura en FIFO..... 88
Figura 5.35- Código función ReadStart..... 88
Figura 5.36.- Código función ReadStart..... 89
Figura 5.37- Configuración SPI maestro en Uno32..... 90
Figura 5.38- Fichero SpiRam.h..... 91
Figura 5.39- Estructuras para almacenar datos en EEPROM..... 93
Figura 5.40- Contenido del fichero EEPROM.h..... 96
Figura 5.41- Comando de control de control para el catálogo de estrellas..... 97
Figura 5.42 Acceso al catálogo de estrellas con la variable Eeprom..... 98
Figura 5.43- Secuencia del Uno32 para el sensor estelar..... 98
Figura 5.44- Funciones auxiliares para el sensor estelar..... 99
Figura 5.45- Aspecto de la aplicación de control del Star Tracker..... 100

LISTADO DE TABLAS

4.- RESULTADOS SOFTWARE

Tabla 4.1. Características del sensor KODAK KC-9618..... 34
Tabla 4.2. Lente 1..... 36
Tabla 4.3. Lente 2..... 36
Tabla 4.4. Lente 3..... 36
Tabla 4.5. Lente 4..... 36
Tabla 4.6. Características del sensor KODAK KC-9618 y su lente..... 44

5.- SISTEMA DE PRUEBAS

Tabla 5.1.- características del sensor OV7670..... 63
Tabla 5.2.- Pines de control de imagen del sensor OV7670..... 64
Tabla 5.3.- Datos guardados como palabras de 4 bytes..... 66
Tabla 5.4.- Orden de llagada de datos en YCbCr422..... 66
Tabla 5.5.- Pixels en YCbCr422..... 66
Tabla 5.6.- Descripción pines AL422B..... 71
Tabla 5.7- Tamaño del catálogo de estrellas en la EEPROM..... 95

1.- INTRODUCIÓN

Uno de los aspectos más importantes en las misiones espaciales es la determinación de la actitud de la nave y su posterior control. Para la determinación de la actitud harán falta una serie de sensores estelares, y para el control serán necesarios unos actuadores. El sistema en conjunto realizará la corrección y control de la actitud de la nave mediante los actuadores, después de haber calculado la actitud que en ese momento presenta.

El sistema encargado de esto se conoce como Subsistema de Control de Actitud (ACS), que comprende a su vez el Subsistema de Determinación y Control de la Actitud (ADCS). A menudo, la determinación de la actitud es el proceso más costoso de todos. En este proyecto, vamos a centrarnos en el proceso de determinación de la actitud de un satélite tipo CubeSat, correspondiente a un estándar de diseño de pico-satélites, utilizados en el proyecto Humsat [17]. El objetivo de este proyecto es la reducción de los costes y los tiempos de desarrollos de satélites, favoreciendo de esta manera el acceso al espacio. El CubeSat, concretamente, posee tres celdas de diez centímetros de lado y un peso máximo de 1,33 kilos por celda. Humsat lanzará una constelación de satélites de bajo coste, diseñados por estudiantes, y con fines humanitarios.

1.1.- SENSOR ESTELAR

Para calcular la actitud del CubeSat, vamos a utilizar un sensor estelar consistente en una cámara. Esta cámara tomará una imagen del firmamento y localizará los centroides de los puntos de luz presentes, y que coincidirán con las posiciones de las estrellas observadas en el firmamento por la cámara. Posteriormente, mediante un método de *Voting*, compararemos esas estrellas candidatas presentes en la imagen con un catálogo de estrellas que antes del lanzamiento del CubeSat habremos almacenado en memoria, para así identificar las estrellas reales presentes en la imagen. Mediante la comparación de las estrellas candidatas y las reales, seremos capaces de calcular la actitud del satélite, que pasaremos al sistema de control para que tome las medidas oportunas en el control de la actitud del CubeSat.

Estudiaremos el algoritmo que realiza este cálculo de la actitud, y modelaremos el sistema hardware previamente diseñado en su proyecto por López Suesma [11]. Con este modelo, realizaremos una simulación con imágenes sintéticas con actitud aleatoria que previamente también

habremos generado. Mediante la comparación de la actitud conocida que hemos generado en las imágenes, y el cálculo que nuestro algoritmo realiza, estudiaremos el error cometido por nuestro algoritmo, y con ello sacaremos las conclusiones sobre la precisión y robustez del sistema.

1.2.- SISTEMA DE PRUEBAS

Antes de proceder a la fabricación del hardware propuesto por López Suesma, vamos a realizar un sistema de pruebas real donde probar todo nuestro algoritmo. Para ello seleccionamos la plataforma de prototipado rápido Chipkit Uno32, que contiene precisamente el mismo procesador PIC32 del diseño original, con lo que podremos probar todo lo que necesitemos de una manera más rápida y económica, sin necesidad de la fabricación y posterior depuración de diseños tanto hardware como software. Para poder depurar el sistema, crearemos una aplicación de control sobre PC que hará las funciones de controlador general del Cubesat.

1.3. ESTRUCTURA DEL PROYECTO

Siguiendo el guión de los dos puntos anteriores, en el capítulo 2 de este proyecto vamos a ver en términos generales los algoritmos implicados en la identificación de estrellas y cálculo de la actitud mediante un sensor estelar formado por una cámara y el análisis de las imágenes por ella tomada. Estudiaremos los algoritmos para los tres principales procedimientos implicados en el proceso: el centroiding, la identificación de estrellas (*Star Matching*) y la determinación de la actitud.

En el capítulo3 vamos a definir la estructura software que dará forma en Matlab a los algoritmos descritos en el capítulo 2. Veremos su diagrama de flujo y estudiaremos en detalle las funciones que lo componen.

En el capítulo 4 se modelará el hardware propuesto por López Suesma y se realizarán las simulaciones del sistema con imágenes sintéticas que también serán generadas por nuestro algoritmo. Realizaremos un análisis de los resultados de la simulación y sacaremos las conclusiones oportunas.

El capítulo 5 describe el sistema de pruebas diseñado para probar los algoritmos sobre un sistema real. Se pasará el código generado en Matlab a C/C++ del IDE de desarrollo MPIDE que presenta la plataforma Chipkit. Veremos el diseño hardware, su control mediante el software/ firmware que llevará incorporado el PIC32 del ACS, y probaremos todo el sistema hardware con la ayuda de una aplicación de control en PC diseñada para tal fin.

El capítulo 6 presentará las conclusiones obtenidas durante todo el proyecto, dejando las líneas abiertas y futuras mejoras para el capítulo 7.

Por último, presentaremos una serie de anexos que nos ayudarán a entender mejor algunos conceptos como los sistemas de coordenadas, y también el código fuente principal desarrollado, así como las hojas de datos con las principales características de los elementos hardware utilizados.

Junto con esta memoria, organizada en carpetas, se presenta toda la documentación anexa al proyecto, como los códigos fuente de todos los programas desarrollados, los ficheros de resultados de las simulaciones y las hojas de características de todos los elementos utilizados.

2.- ALGORITMOS

2.1.- INTRODUCCIÓN

El proceso de *"star tracking"* consiste en tres principales pasos:

- Centroiding
- Star identification o identificación de estrellas
- Attitude determination o determinación de la actitud.

El paso de *centroiding* toma una imagen de la cámara y determina las coordenadas de las fuentes de luz en el plano de la imagen, coordenadas que pueden ser convertidas a vectores unitarios en el sistema de referencia del tracker. La identificación de estrellas es el paso crucial del sistema. Los vectores unitarios en el sistema de coordenadas del tracker son analizados y comparados con el catálogo de estrellas para determinar qué estrellas se hayan en el plano de la imagen y consecuentemente generar sus vectores unitarios en el sistema de referencia inercial. Finalmente, la lista de vectores unitarios obtenidos en el sistema de referencia del tracker y en el sistema de referencia inercial son utilizados por un algoritmo para determinar la actitud del tracker en el sistema de referencia inercial.

2.2.- CENTROIDING

El primer paso en cualquier Star Tracker consiste en determinar la localización de las estrellas en el plano de la imagen tomada con la cámara. Si la imagen capturada contiene estrellas muy enfocadas, la luz que proviene de cada estrella puede incidir en sólo uno o dos pixels, pudiendo saturarlos, dando como resultado una imagen con una gran precisión a nivel de pixel, como se muestra en la figura 2.1.

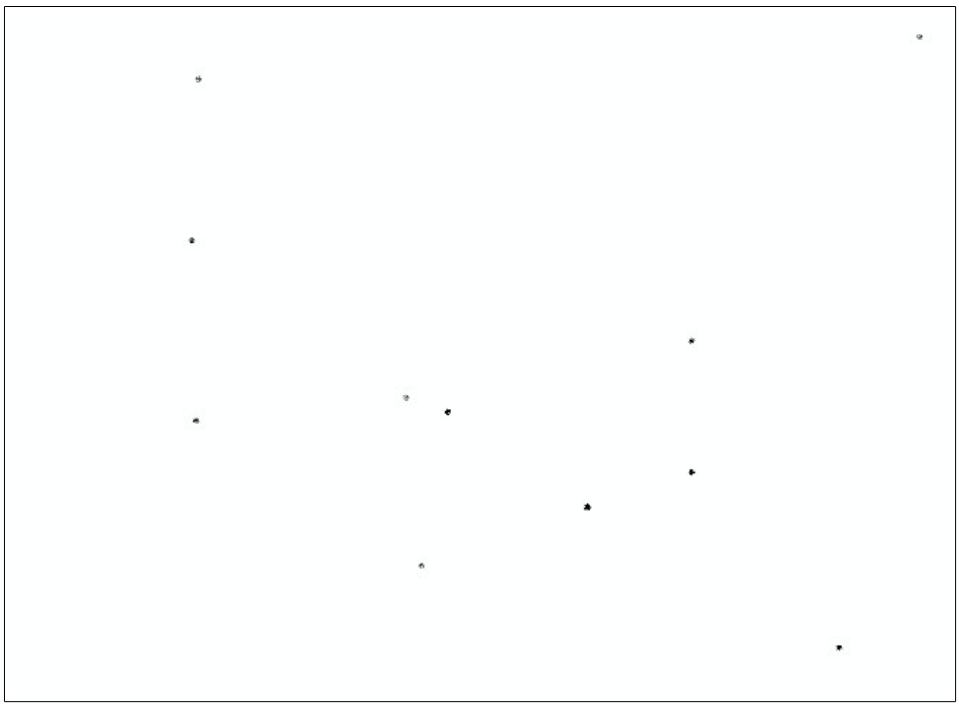


Figura 2.1: Imagen con estrellas muy enfocadas

La mayoría de los star trackers, como es nuestro caso, utiliza imágenes intencionadamente desenfocadas, como la mostrada en la figura 2.2, con el objetivo de propagar los fotones sobre más pixels, haciendo que el algoritmo de centroiding alcance precisiones a niveles de sub-pixel [1].

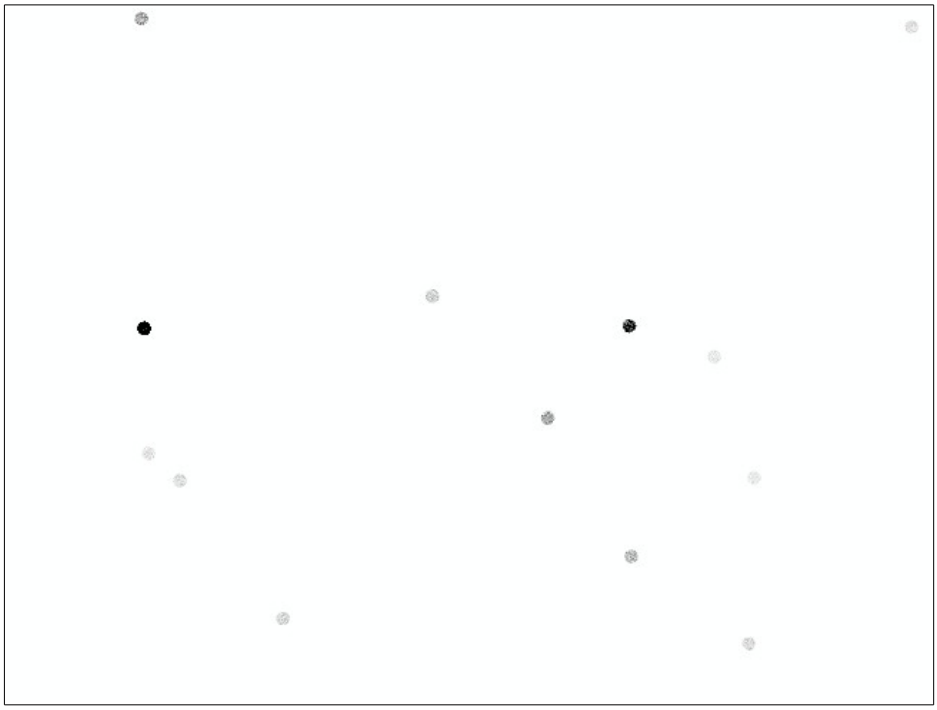


Figura 2.2: Imagen con estrellas desenfocadas

Después de que la imagen desenfocada es tomada por la cámara, el centroide de cada estrella se calcula de una manera muy similar al centroide de un array de puntos de masa, pero con un par de diferencias. La primera diferencia consiste en que se utiliza la intensidad de la luz en lugar de la masa, y la segunda es que dicha intensidad de luz se normaliza con los pixels de alrededor de la estrella con el objetivo de eliminar los deslumbramientos o el ruido de fondo. La salida resultante del algoritmo de centroiding es una array de coordenadas bidimensionales en el plano de la imagen con origen en el centro de la imagen. Este sistema de coordenadas permite que posteriormente las coordenadas de las estrellas sean fácilmente convertidas a vectores unitarios.

A continuación se describe el algoritmo de centroiding utilizado para este proyecto, tomado de McBride [2], el cual está adaptado del método presentado por Liebe [1]. EL algoritmo requiere dos parámetros de entrada:

- Límite o *threshold* de intensidad de luz I_{thresh}.
- Tamaño de la región de interés (ROI) a_{ROI} en pixels.

Estos valores se pueden ajustar para obtener el mejor comportamiento en el algoritmo de centroiding. Por ejemplo, un valor más elevado de I_{thresh} hace al algoritmo más robusto frente al ruido pero puede hacer desaparecer alguna estrella de la imagen. De igual modo, un valor más elevado de a_{ROI} implica una mayor precisión en el valor del centroide pero puede dar lugar a la lectura de una estrella donde realmente se encuentran dos estrellas muy próximas. Para que el algoritmo funcione de forma correcta el valor de a_{ROI} debe ser impar.

El algoritmo de centroiding utilizado en este proyecto consta de los siguientes pasos:

1. Para cada pixel en las coordenadas de la imagen (x,y) con un valor de intensidad $I(x,y)>I_{thresh}$, definimos su ROI como un cuadrado de a_{ROI} pixels de lado, con la esquina superior izquierda situada en (x_{start},y_{start}), dada por las ecuaciones 2.1 y 2.2 y la esquina inferior derecha situada en (x_{end},y_{end}) dada por las ecuaciones 2.3 y 2.4.

$$x_{start} = x - \frac{a_{ROI} - 1}{2} \tag{2.1}$$

$$y_{start} = y - \frac{a_{ROI} - 1}{2} \tag{2.2}$$

$$x_{end} = x_{start} + a_{ROI} \tag{2.3}$$

$$y_{end} = y_{start} + a_{ROI} \tag{2.4}$$

2. Si $x_{start} < 0$ ó $y_{start} < 0$, se descarta el pixel y se vuelve al paso 1 para el siguiente pixel.
3. Se calcula el valor de intensidad media de los pixels del borde I_{border} dados por las ecuaciones 2.5 y mostrados en la figura 2.3 para un valor de $a_{ROI} = 7$.

$$I_{botton} = \sum_{i=x_{start}}^{x_{end}-1} I(i, y_{start}) \tag{2.5a}$$

$$I_{top} = \sum_{i=x_{start}+1}^{x_{end}} I(i, y_{end}) \tag{2.5b}$$

$$I_{\text{left}} = \sum_{j=y_{start}}^{y_{end}-1} I(x_{start}, j) \tag{2.5c}$$

$$I_{\text{right}} = \sum_{j=y_{start}+1}^{y_{end}} I(x_{end}, j) \tag{2.5d}$$

$$I_{border} = \frac{I_{top} + I_{botton} + I_{\text{left}} + I_{\text{right}}}{4(a_{ROI} - 1)} \tag{2.5e}$$

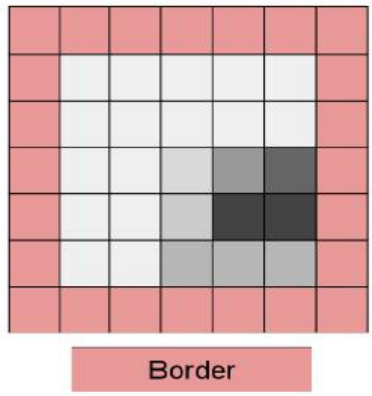


Figura 2.3: Pixels de borde para $a_{ROI} = 7$. Las sombras corresponden al negativo de la estrella

4. Restar a *I(x,y)* el valor de I_{border} calculado previamente para todos los pixels que no pertenezcan al borde, obteniendo así la matriz normalizada de intensidad de luz $\tilde{I}$

$$\tilde{I}(x, y) = I(x, y) - I_{border} \tag{2.6}$$

5. Calcular la localización del centroide (x_{CM}, y_{CM}) utilizado las ecuaciones 2.7, 2.8 y 2.9. Nótese que el brillo B calculado en la ecuación 2.7 es equivalente a la masa total en un array de puntos de masa:

$$B = \sum_{i=x_{start}+1}^{x_{end}-1} \sum_{j=y_{start}+1}^{y_{end}-1} \tilde{I}(i,j) \tag{2.7}$$

$$x_{CM} = \sum_{i=x_{start}+1}^{x_{end}-1} \sum_{j=y_{start}+1}^{y_{end}-1} \frac{i\,x\,\tilde{I}(i,j)}{B} \tag{2.8}$$

$$y_{CM} = \sum_{i=x_{start}+1}^{x_{end}-1} \sum_{j=y_{start}+1}^{y_{end}-1} \frac{j\,x\,\tilde{I}(i,j)}{B} \tag{2.9}$$

6. Una vez calculada la posición del centroide (x_{CM},y_{CM}) para cada pixel que supere el valor I_{thresh}, recorremos todos los centroides calculados y calculamos el promedio de los centroides que se encuentren relativamente cerca. Suponemos que estos valores representan la misma estrella aunque cabe la posibilidad de que dos estrellas se encuentren muy próximas. Sin embargo, esta es una suposición razonable si el límite de la magnitud de la cámara es lo suficientemente alto.

   El proceso de agrupamiento se puede realizar comprobando la posición de cada centroide frente a una lista de posiciones de centroides ya previamente comprobadas. Si, por ejemplo, la nueva posición se encuentra dentro de una región de 5 pixels, promediamos ambos valores. Este paso devuelve una lista de coordenadas de centroides promediadas, cada una de las cuales representará una fuente de luz independiente.

7. Convertimos la lista de coordenadas de centroides promediadas en vectores unitarios. Para ello usaremos el tamaño del pixel de la cámara μ y la longitud focal de su lente f. La geometría de esta transformación se muestra en la figura 2.4.

$$u = \frac{[\mu * x_{CM}, \mu * y_{CM}, f]^T}{\left\| [\mu * x_{CM}, \mu * y_{CM}, f] \right\|} \tag{2.10}$$

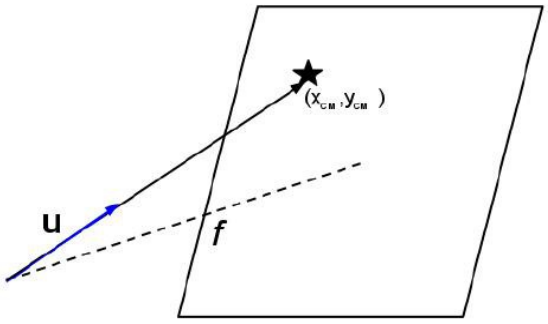


Figura 2.4: Vector unitario de una estrella

2.3.- STAR IDENTIFICATION

Como hemos comentado anteriormente, el proceso de identificación de estrellas es el paso crucial del sistema. Los vectores unitarios en el sistema de coordenadas del satélite calculados con el algoritmo de centroiding son analizados y comparados con el catálogo de estrellas para determinar qué estrellas se hayan en el plano de la imagen y generar sus vectores unitarios en el sistema de referencia inercial. Existen varios algoritmos que realizan este proceso de identificación de estrellas. Veamos en el siguiente apartado alguno de ellos.

2.3.1.- ESTUDIO DE ALGORITMOS

El problema de la identificación de estrellas ha sido muy investigado y numerosos estudios han sido realizados dando lugar a varios métodos [8, 9, 10, 3, 4]. Todos los métodos aquí tratados utilizan los vectores unitarios calculados en el algoritmo de centroiding del apartado anterior y dados por la ecuación 2.20, y algunos utilizan otra información como el brillo aparente de las estrellas. Además, los algoritmos de identificación de estrellas pueden ser divididos en dos tipos: lost-in-space (LIS, perdidos en el espacio en castellano) y tracking (seguimiento). Los primeros intentan la identificación de estrellas en la imagen basándose estrictamente en la información de la imagen, mientras que los segundos también utilizan los datos de la actitud previa o *a priori*, como por ejemplo la localización previa de las estrellas identificadas, o la actitud estimada por otro sensor o filtro dinámico. La mayoría de los algoritmos de esta sección pueden ser usados igualmente como LIS y como algoritmos de tracking.

Todos los algoritmos que se detallan a continuación comparten una secuencia en común:

1. Generar una lista de geometrías posibles partiendo de un catálogo de estrellas dado.
2. Asociar las estrellas observadas a las geometrías del catálogo.
3. Evaluar la confianza de la identificación de estrellas y desechar los resultados falsos.

2.3.1.1.- TÉCNICA DE IDENTIFIACIÓN PIRAMIDAL

En contraposición a las técnicas de identificación de estrellas por triángulos, Mortari [8] presenta un método relacional basado en pirámides. Este método consiste en los siguientes pasos:

1. Crear un *k-vector* de posibles pares de estrellas. En lugar de una lista de geometrías estándar que implique una búsqueda lineal, Mortari emplea un algoritmo de búsqueda basado en *k-vector*s. Un *k-vector* es una lista ordenada de ángulos entre estrellas, permitiendo la identificación de los potenciales pares de estrellas utilizando un factor calculado en lugar de métodos lineales de búsqueda u otros métodos [4].

2. Comenzamos el proceso de identificación de estrellas escaneando las estrellas observadas para un único triángulo. En lugar de una secuencia que recorra cada combinación de estrellas empezando por las tres primeras (es decir, 1-2-3, 1-2-4, 1-2-5...), se usa una secuencia priorizada que recorre todas las estrellas más rápido (1-2-3, 2-3-4, 3-4-5...). También Mortari define de manera unívoca, utilizando una fórmula, una probabilidad de que los triángulos coincidan. Si esta probabilidad es muy alta, el triangulo es dado como salida del algoritmo.

3. Recorremos las estrellas restantes para una estrella que forme triángulos adicionales con las tres estrellas del triángulo. Si no encontramos dichas estrellas, volvemos al paso 2 y probamos con el siguiente triángulo de la secuencia.

4. Una vez hayamos encontrado una combinación de 4 estrellas con alta confianza, obtenemos la pirámide de la figura 2.5, que da nombre al método. El resto de estrellas observadas pueden ser identificadas o rechazadas como ruido, usando de nuevo la fórmula mencionada anteriormente.

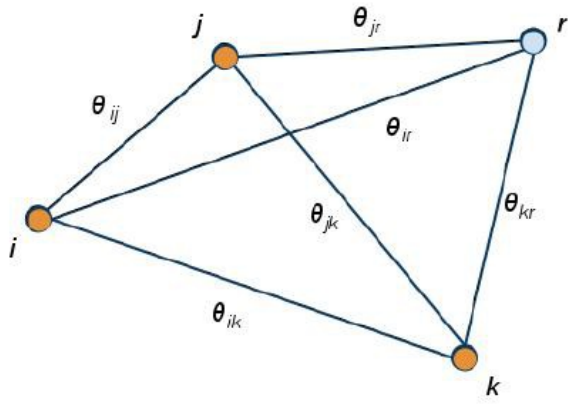


Figura 2.5: Pirámide de estrellas para las estrellas candidatas i, j, k, y estrella de referencia r

2.3.1.2.- MÉTODO VOTING

De manera similar a la técnica de identificación de estrellas por pirámides de estrellas presentado en el punto anterior, el método *Voting* usa información adicional sobre la estrella para mejorar la precisión de la identificación. Sin embargo, este método no se limita tan sólo a una estrella adicional, sino que usa información de cada estrella en la imagen para evaluar democráticamente la identidad de cada estrella. El método consiste en los siguientes pasos:

1. El catálogo de estrellas generado por el método de *Voting* es una simple serie de posibles pares de estrellas y sus distancias angulares. Si se desea, y como mejora en el rendimiento del algoritmo total, se puede usar el enfoque de *k-vectores* [4] comentado en el algoritmo piramidal del punto anterior.

2. Para cada par de estrellas en la imagen tomada por la cámara, se calcula su distancia angular. Se recorre el catálogo y se añaden los números de identificación de las estrellas cuya distancia es inferior a la del par más una tolerancia dada, a una lista para cada estrella. Ambas identidades se añaden a ambas listas puesto que ambas son posibles para cada par de estrellas candidatas.

3. Una vez todos los pares de estrellas han sido analizados, se selecciona la identidad de cada estrella del catálogo que ha recibido la mayor cantidad de votos.

4. Se verifica ahora la precisión de los pares de estrellas identificados. Para cada par de estrellas identificado, se calcula la distancia angular usando el catálogo. Si esta distancia cae dentro de una determinada tolerancia, esas estrellas reciben sendos votos.

El algoritmo genera como salida los vectores unitarios de las estrellas que han recibido una cantidad de votos superior a una determinado valor, normalmente el máximo numero de votos que cualquier estrella ha recibido menos uno. Si ese valor es menor o igual que 0, se trata de una falsa identificación de la estrella.

2.3.1.3.- TÉCNICA DE IDENTIFIACIÓN POR TRIÁNGULO PLANAR

Cole [10] presenta un diferente enfoque para la identificación de estrellas. Aunque este método también usa triángulos de estrellas, Cole analiza los triángulos en sí mismos, realizando

búsquedas de patrones usando características del triángulo en su conjunto, en lugar de cada lado, del modo en el que se realiza en los dos métodos presentados anteriormente. Este algoritmo se presenta a continuación:

1. Se construye una lista de posibles triángulos de estrellas, de las estrellas *p, q y r* del catálogo, así como sus áreas y momentos polares de inercia. El momento polar, dado en la ecuación 2.13 como *J*, predice la resistencia a la torsión de la figura planar sobre su centro de masas en la dirección *z*. Las tres estrellas deben satisfacer los requerimientos de campo de visión y magnitud, y el área y el momento polar de cada triángulo se pueden calcular usando las ecuaciones 2.11, 2.12, conocidas como fórmula Heron, y la ecuación 2.13. Se repite este paso para cada potencial combinación de estrellas del catálogo.

$$s = \frac{1}{2}(a+b+c) \tag{2.11a}$$

$$a = \|u_p - u_q\| \tag{2.11b}$$

$$b = \|u_q - u_r\| \tag{2.11c}$$

$$c = \|u_p - u_r\| \tag{2.11d}$$

$$A = \sqrt{s(s-a)(s-b)(s-c)} \tag{2.12}$$

$$J = A\frac{(a^2+b^2+c^2)}{36} \tag{2.13}$$

2. Se comienza el proceso de identificación seleccionando tres estrellas y calculando el área y momento polar del triángulo planar que forman usando las ecuaciones 2.12 y 2.13. Además, se calcula la varianza del área y del momento polar. El proceso de cálculo de estas varianzas se presenta en [10]. Utilizando un método de búsqueda basado en *k-vectors*, u otro método, se encuentran todos los triángulos de la lista formada en el paso 1 cuyas áreas y momentos polares estén dentro de la desviación estándar del triangulo observado. Si sólo un triángulo cumple este criterio, se continúa en el paso 4.

3. Si más de un triángulo del catálogo cumple los criterios de área y momento polar, se selecciona otra estrella de la imagen para identificar, y se comprueba cuantas estrellas aparecen en las dos listas. Si sólo son dos, se identifica el triángulo. Si no, se saca el resultado y el algoritmo vuelve al paso 3 con la siguiente combinación de tres estrellas.

4. Una vez las tres estrellas han sido identificadas las estrellas restantes de la imagen se pueden identificar si es necesario utilizando el mismo proceso de pivotaje descrito en el paso 4. Si no, se procede a la determinación de la actitud.

2.3.1.4.- ALGORITMO GRID

Padgett y Kreutz-Delgado [9] presentan un método conocido como algoritmo grid o de rejilla. En lugar de leer series de estrellas e identificar una serie de posiciones relativas, como se hace en los dos primeros métodos, el algoritmo grid asocia un catálogo de patrones con estrellas observadas. Para lograr esto, el campo de visión de la cámara se divide en una rejilla y se forma una matriz con ceros y unos, dependiendo de si una estrella existe o no en cada celda de la rejilla. El proceso consta de los siguientes pasos:

1. Construir una lista de patrones para cada estrella del catálogo. Para cada estrella del catálogo, rotar las otras estrellas del catálogo de tal forma que la estrella elegida caiga en el eje *z* positivo del plano de la cámara. Si otra estrella del catálogo está fuera de un determinado radio pero dentro del FOV (*field-of-view*) de la cámara, colocar la estrella en una imagen ficticia y cambiar el valor para esa celda de cero a uno.

2. Una vez se ha completado el paso 1, deberá de haber una matriz de ceros y unos para cada estrella del catálogo.

3. Para realizar la identificación de estrellas, se ordenan las estrellas identificadas en la imagen por brillo. Empezando por las estrellas más brillantes, se trasladan las localizaciones de las otras estrellas de tal manera que la estrella más brillante caiga en el centro de la imagen.

4. Se aplica el mismo tamaño de rejilla del paso 1 y se forma una matriz de unos y ceros basada en si existe o no una estrella en cada celda.

5. Se compara la matriz con los patrones del catálogo, y se guarda el patrón con la matriz con mayor número de valores distintos de cero que coinciden entre la imagen y los patrones del catálogo, así como cuantos valores coinciden.

6. Continuar con este proceso para cada estrella de la imagen y seleccionar como referencia la estrella cuyo patrón ha tenido el mayor número de coincidencias.

7. Finalmente, verificar las distancias entre estrellas que hemos intentado identificar y la estrella referencia. Si suficientes distancias son correctas, se devuelven los identificadores de las estrellas.

Este método de rejilla pude resultar un tanto complicado de entender. La figura 2.6., tomada de Padgett [9], da una representación visual de los pasos del algoritmo.

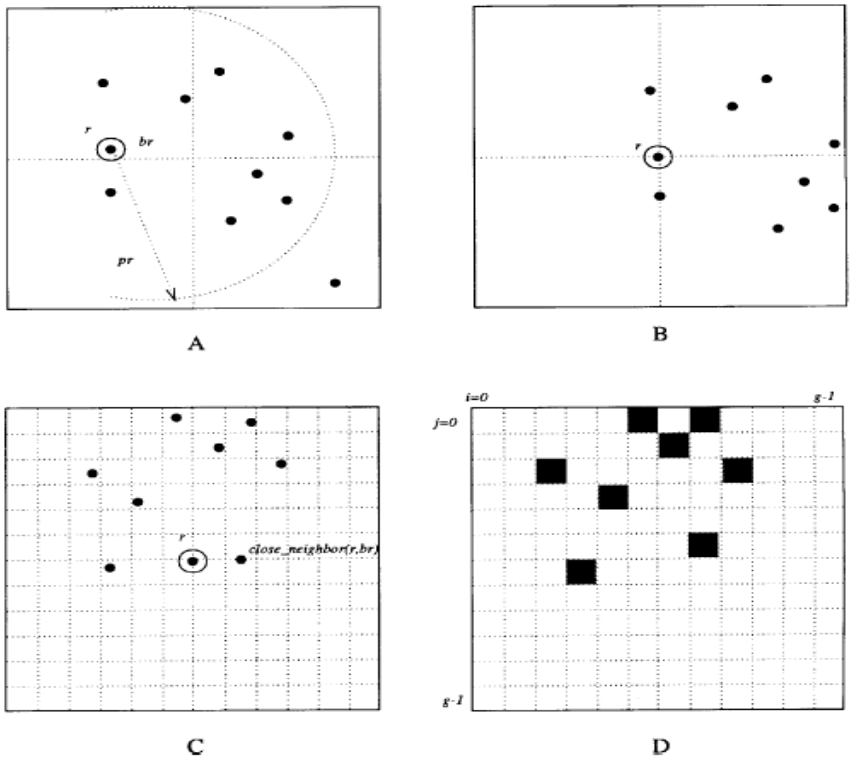


Figura 2.6: Descripción de los pasos del algoritmo de grid. ***A****: estrella más brillante identificada.* ***B****: imagen trasladada con respecto a la estrella más brillante.* ***C****: rejilla alineada con el vecino más cercano.* ***D****: patrón de bits.*

2.3.1.5.-COMPARACIÓN DE MÉTODOS

Cada uno de los cuatro métodos descritos anteriormente tiene distintas ventajas y desventajas frente a los demás. Antes de comparar sus méritos relativos, es importante definir qué

características son importantes para un *CubeSat*. Primero, el algoritmo debe ser robusto, es decir, tolerante al ruido y a los errores en la imagen. Los *CubeSat* son pequeños y deben tener un peso muy pequeño, lo que normalmente implica que el apantallamiento para la radiación sea menor que en otro tipo de satélites que vuelan en órbitas bajas (*Low Earth Orbit LEO*). Por tanto hay una gran probabilidad de errores en las imágenes tomadas debidos a radiación perdida o rayos cósmicos. Además, los satélites en *LEO* tienen gran posibilidad de observar otros satélites o basura espacial y confundirlos con estrellas. También, para los *Cubesat* se prefieren cámaras con baja resolución. Estas cámaras son pequeñas y consumen menos energía, y reducen la carga en el sistema de comandos y manejo de datos (*Command and Data Handling, CDH*). Otro aspecto importante en el *CDH* es el almacenamiento y acceso de datos. Cuanto menor número de llamadas al catálogo y menor sea este catálogo, mas eficiente será el *Star Tracker* y el *CDH*. A pesar de estas consideraciones, los *Star Tracker* en los *CubeSat* deben proporcionar una mejora significativa en la precisión frente a otros instrumentos. El volumen a bordo de un *CubeSat* es escaso y el *Star Tracker* tiene que justificar su presencia.

El método de identificación piramidal puede ser considerado como bastante robusto. Pero como debe de identificar al menos cuatro estrellas, cualquier presencia parásita o falsa estrella puede hacer que el algoritmo falle y la localización sea rechazada. Sin embargo, una localización rechazada implica que el algoritmo debe empezar con una nueva combinación de estrellas. El método de *Voting* es extremadamente robusto frente a falsas estrellas. En los tests descritos por Kolimenkin en [3], el método Voting conserva el número correcto de estrellas identificadas incluso cuando el número de falsas estrellas es tres veces el número de estrellas actual, y logra este resultado sin necesidad de volver a empezar el algoritmo. El algoritmo del triángulo planar es menos robusto. Requiere una identificación del área y del momento polar que ayuda a eliminar la ambigüedad, pero una falsa estrella mal localizada puede hacer que el método devuelva un falso positivo. Finalmente, el método *Grid* es muy robusto. Una falsa estrella simplemente añade un uno en la matriz, pero no tiene un impacto significativo en la precisión de la identificación a menos que haya una gran cantidad de estrellas falsas.

El requerimiento de baja resolución no tiene un efecto significativo en los tres primeros métodos. Utilizando el desenfoque y el método centroiding, obtenemos una precisión de sub-pixel para cualquiera de estos tres métodos. Esto no ocurre con el método *Grid*. Una baja resolución significa que la rejilla debe ser de celdas pequeñas, y esto incrementa la probabilidad de que se encuentren estrellas en los bordes o que las estrellas caigan en una o dos celdas.

En cuanto a los requerimientos del catálogo, el algoritmo piramidal y el de *Voting* son similares. Los catálogos tienen el mismo tamaño, ambos usan pares de estrellas y el mismo número de llamadas al catálogo, aunque la demanda puede ser mayor en el método *Voting* si se observan gran cantidad de estrellas en la imagen. El método de triángulo planar requiere dos catálogos muy grandes para el área y el momento polar, pero no tiene un aumento significativo en el número de llamadas al catálogo. El método de rejilla puede tener también un extenso catálogo puesto que debe de haber una matriz de bits para cada estrella del catálogo.

Sopesando los tres criterios antes mencionados, los métodos piramidal y *Voting* parecen los más adecuados para nuestras pretensiones, aunque finalmente nos quedaremos con el método *Voting*, debido a su alta robustez, que nos dará una mayor confianza frente a las inciertas condiciones de operación de *Star Tracker* en un *CubeSat*.

2.3.2.- MÉTODO VOTING

De todos los algoritmos presentados para la identificación de estrellas en el apartado anterior, seleccionamos el algoritmo de *Voting* descrito por Kolomenkin en [3] y presentado en la sección 2.3.1. Este algoritmo, descrito por el diagrama de bloques de la figura 2.7, lo podemos definir mediante las secciones que en los siguientes apartados se detallan.

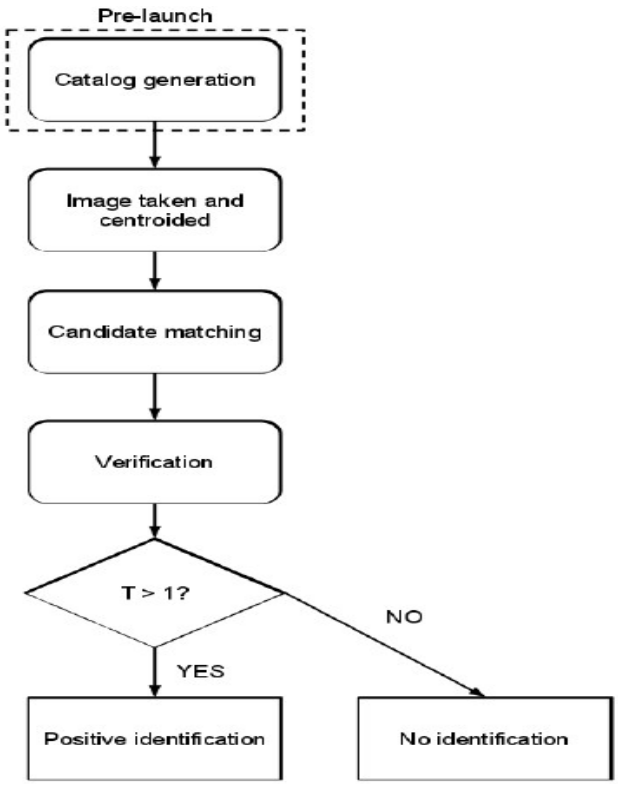


Figura 2.7: Diagrama de bloques del método Voting

2.3.2.1.- GENERACIÓN DEL CATÁLOGO

El proceso de generación del catálogo para el método de *Voting* es muy similar para el mismo proceso en otras técnicas. Existen varios catálogos de estrellas que recogen varios miles de estrellas identificadas con sus propiedades de luminosidad y su posición en diferentes sistemas de coordenadas. Partiendo de uno de estos catálogos generales, crearemos nuestro propio catálogo con la información necesaria y ordenada de la manera en que mejor nos convenga para los posteriores pasos del algoritmo de identificación de estrellas.

Partimos de un valor mínimo de intensidad, que determinará lo que podemos considerar como una estrella, y del campo de visión (*field-of-view* ***FOV***) de la lente de la cámara, para generar una lista de posibles pares de estrellas con sus distancias angulares. Este paso se realiza antes de la instalación y el catálogo obtenido se almacena en la memoria del satélite para su posterior uso. Cabe la posibilidad de actualizar o modificar este catálogo para un mejor funcionamiento del algoritmo. El catálogo se generará de la siguiente forma:

1. Convertimos las posiciones de las estrellas del catálogo en vectores unitarios. La mayoría de los catálogos de estrellas existentes almacenan las posiciones de las estrellas en coordenadas inerciales de ascensión recta y declinación. Aunque es un sistema muy práctico para astrónomos, para nuestro algoritmo transformaremos estas coordenadas en vectores unitarios en el sistema de coordenadas inercial. La ecuación 2.14 muestra cómo se obtiene el vector unitario de una estrella en términos de su ascensión recta α y declinación δ. En el ANEXO I podemos ver las definiciones de los sistemas de coordenadas que utilizaremos a lo largo de la memoria de este proyecto.

$$u = \begin{bmatrix} \cos(\alpha)\cos(\delta) \\ \sin(\alpha)\cos(\delta) \\ \sin(\alpha) \end{bmatrix} \tag{2.14}$$

2. Para cada par de estrellas con números de identificación a y b, verificar que las dos estrellas de ese par cumplen las condiciones de magnitud y distancia angular dadas por las ecuaciones 2.15, 2.16 y 2.17. A tener en cuenta que cuanto más brillante es un objeto en el cielo, su magnitud en los catálogos de estrellas es menor:

$$m_a \leqslant m_{max} \tag{2.15}$$

$$m_b \leqslant m_{max} \tag{2.16}$$

$$u_a^T u_b \geqslant \cos(\theta_{FOV}) \tag{2.17}$$

3. Si las condiciones anteriores se cumplen, almacenamos los números de identificación de las estrellas así como el valor del escalar $u_a^T u_b$ en un fichero para su posterior acceso. El valor de $u_a^T u_b$ es igual al coseno del ángulo formado por los dos vectores unitarios de cada estrella del par. Al verificar que este valor es mayor que el coseno del ángulo del campo de visión, realmente estamos comprobando que la distancia angular entre ambos vectores unitarios es menor que el ángulo FOV, asegurándonos de que ambas estrellas pueden ser vistas por la cámara al mismo tiempo.

2.3.2.2.- CANDIDATE MATCHING

Este paso es el primero en el proceso que tiene lugar con cada operación del *Star Tracker*. En este punto, tendremos disponible una lista con los vectores unitarios correspondientes a las fuentes de luz observadas por la cámara en su sistema de referencia y que el algoritmo de centroiding nos ha devuelto. De aquí en adelante, a estos puntos los denominaremos ***estrellas candidatas***, pues no tenemos manera de saber si se tratan realmente de estrellas, o falsas estrellas, como planetas, satélites u otras fuentes de ruido en la imagen. El proceso consta de los siguientes pasos:

1. Para cada par de estrellas candidatas *i* y *j*, calculamos el coseno de la distancia angular entre ellas, denominado d_{ij}, mediante la ecuación 2.18:

$$d_{ij} = u_i^T u_j \tag{2.18}$$

2. Buscamos cada par de estrellas *p* y *q* en el catálogo cuya distancia angular d_{pq} satisfaga la ecuación 2.19 para una tolerancia ε dada. El enfoque de *k-vector* y la búsqueda binaria usados en [4] y mencionados en [3] pueden ser usados para este punto.

$$d_{ij} - \varepsilon \leqslant d_{pq} \leqslant d_{ij} + \varepsilon \tag{2.19}$$

3. Para cada posible par de estrellas encontrado en el punto 2, añadimos el numero de identificación de ambas estrellas *p* y *q* en el catálogo, a los arrays de estrellas candidatas para las estrellas *i* y *j*. Ambos números de identificación son añadidos a ambas listas, puesto que cada estrella identificada en el catálogo es una posibilidad para ambas estrellas.

4. Una vez todos los posible pares de estrellas candidatas han sido procesados, asignamos a cada estrella candidata la estrella del catálogo que más votos ha recibido en los arrays.

En este punto, cada fuente de luz candidata tendrá una estrella del catálogo asignada a ella. Sin embargo, todavía no tenemos forma de saber si alguna de esas estrellas se trata en realidad de una falsa estrella que pueda ocasionar una solución incorrecta en la actitud final calculada.

2.3.2.3.- VERIFICACIÓN Y RESULTADO FINAL

Este paso de verificación es el encargado de eliminar las posibles falsas estrellas candidatas. Se realiza ahora otra ronda en el algoritmo de *Voting* obteniendo al final de él una lista de estrellas observadas y estrellas candidatas. Para ello, realizaremos los siguientes pasos:

1. Para cada par de estrellas candidatas *i* y *j*, calcular el coseno del ángulo entre sus correspondientes estrellas del catálogo r_{ij} usando la ecuación 2.20, donde v_i es el vector unitario de la estrella del catálogo asociada a la estrella candidata *i*.

$$r_{ij} = v_i^T v_j \tag{2.20}$$

2. Si se satisface la ecuación 2.21, añadiremos un voto en la lista para cada una de las estrellas candidatas *i* y *j*.

$$d_{ij} - \varepsilon \leqslant r_{ij} \leqslant d_{ij} + \varepsilon \tag{2.21}$$

3. Una vez que cada par de estrellas candidatas ha sido procesado, calcular el valor umbral T para estrellas reales dado por la ecuación 2.22:

$$T = max(votes(i)) - 1 \tag{2.22}$$

4. Cada estrella candidata con más votos que el umbral ***T*** representa a una estrella real.

La potencia del algoritmo de *Voting* reside en este paso. Las estrellas reales tienen en este paso un número de votos agrupados y sobre el valor umbral T, mientras que las falsas estrellas no reciben más de uno o dos votos, siendo removidas en este paso. Si asumimos que al menos tres estrellas han sido positivamente identificadas, las dos listas de estrellas candidatas identificadas por el algoritmo y sus correspondientes estrellas del catálogo, pueden ser pasadas al algoritmo de determinación de la actitud.

2.4.- DETERMINACIÓN DE LA ACTITUD

Una vez las estrellas candidatas han sido identificadas con sus correspondientes estrellas del catálogo, tenemos dos listas con sus vectores unitarios correspondientes. La primera de las listas hace referencia al sistema de coordenadas de la cámara, y la segunda al sistema de coordenadas inercial. Para encontrar la relación entre ambos sistemas, y en consecuencia, conocer la actitud del satélite, deberemos aplicar un método para la determinación de dicha actitud. Existen varios métodos conocidos que devuelven cuaternios, como los métodos Davenport y QUEST descritos en [5] y en [6]. Sin embargo, hemos seleccionado el método desarrollado por Markley en [7] basado en matrices coseno directrices y descomposición en valores singulares. El método queda representado por los siguientes pasos:

1. Calcular la matriz B usando la ecuación 2.23, con las *n* estrellas que provienen del algoritmo de identificación de estrellas, donde b_i y r_i para la estrella real *i* son respectivamente los vectores unitarios de la estrellas candidata y la estrella correspondiente en el catálogo.

$$B = \sum_{i=1}^{n} b_i r_i^T \tag{2.23}$$

2. Realizar la descomposición en valores singulares de la matriz B, es decir, encontrar las matrices ortogonales U y V y la matriz diagonal de valores singulares S, que satisfagan la ecuación 2.24:

$$B = U\,S\,V^T \tag{2.24}$$

3. Definimos las matrices ortogonales U_+ y V_+ mediante las ecuaciones 2.25 y 2.26:

$$U_+ = U\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & det(U) \end{bmatrix} \tag{2.25}$$

$$V_+ = V\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & det(V) \end{bmatrix} \tag{2.26}$$

4. Finalmente, la matriz coseno directriz A se calcula mediante la ecuación 2.27:

$$A = U_+ V_+^T \tag{2.27}$$

3.- ARQUITECTURA SOFTWARE

Una vez presentados en el capítulo anterior los algoritmos necesarios para nuestro *Star Tracker*, pasaremos a continuación a explicar en qué consiste su implementación a nivel software. McBride [2] presenta una arquitectura que tomaremos como modelo para nuestro sistema.

3.1.- FLUJO SOFTWARE

El flujo de software que describe la funcionalidad que va a desempeñar nuestro *Star Tracker* viene representado por el diagrama de bloques que se presenta en la figura 3.1. Antes del lanzamiento del satélite, se llama a la función *catinit* para generar la lista de posibles pares de estrellas y sus ángulos relativos. El fichero generado se guarda en la ROM del satélite. Ya a bordo, la función *catinit* se llama de nuevo por el controlador del satélite, pero esta vez para cargar el catálogo en memoria. La función *loading* toma una imagen por medio de la cámara y devuelve una matriz de intensidad de luz. Esta información es pasada a la función *centroid*, que es la encargada de encontrar las fuentes de luz de la imagen, devolviendo sus coordenadas en el plano de la imagen. La función *uvec* usa la geometría de la cámara para convertir estas coordenadas del plano de la imagen en vectores unitarios en el sistema de coordenadas del satélite. Estos vectores unitarios son entregados a la función *starid*, que es la encargada de identificar las estrellas presentes en la imagen y tiene como salida sus vectores unitarios en el sistema de coordenadas del satélite y el sistema de coordenadas inercial. Finalmente, estas dos listas son entregadas a la función *attdet*, que calcula la matriz coseno directriz DCM que transforma las coordenadas desde el sistema inercial al sistema de coordenadas del satélite y devuelve esta información al ordenador de a bordo del satélite.

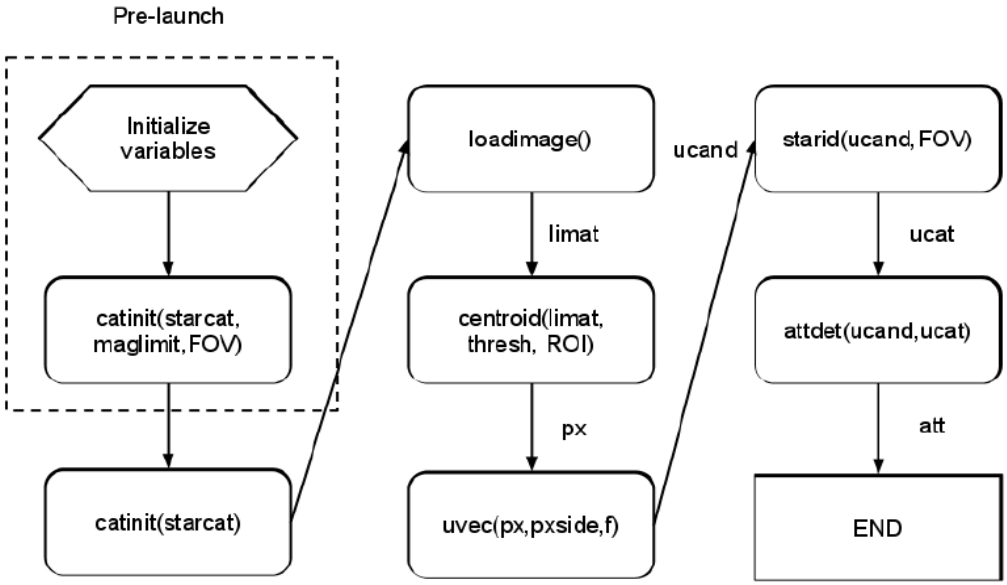


Figura 3.1: Diagrama de flujo del sensor estelar

3.2.- FUNCIONES

El programa para la identificación de estrellas de nuestro *Star Tracker* consiste en unas funciones que aproximadamente coinciden con los tres pasos principales que todo identificador de estrellas debe de presentar, y que han sido descritos en el capítulo 2: centroiding, identificación de estrellas y determinación de la actitud. En muchos casos, estos pasos principales pueden ser descompuestos en varios sub-pasos, dependiendo del lugar que ocupen en el flujo de software las limitaciones del lenguaje de programación seleccionado. Para probar y analizar nuestro algoritmo seleccionamos MATLAB como lenguaje de programación, para más tarde convertirlo a C o C++ para su ejecución en tiempo real en nuestro sistema empotrado de pruebas. Adaptando el algoritmo a las particularidades del hardware seleccionado. Pasamos a continuación a detallar cada una de las funciones implementadas en nuestro *Star Tracker*, cuyo código se adjunta con la memoria de este proyecto.

3.2.1.- FUNCIÓN *catinit*

Esta función tiene el propósito de generar los posibles pares de estrellas y sus distancias angulares relativas basados en la información sobre la magnitud de las estrellas y el campo de visión FOV de la cámara, y devuelve esos datos a la función principal del *Star Tracker*. Además salva esta información en un fichero de texto con formato *.mat*. Si este fichero ha sido

anteriormente generado, y por tanto no pasamos como argumentos el FOV ni la magnitud, leeremos la información de este catálogo directamente. La figura 3.2 muestra el diagrama de flujo de esta función.

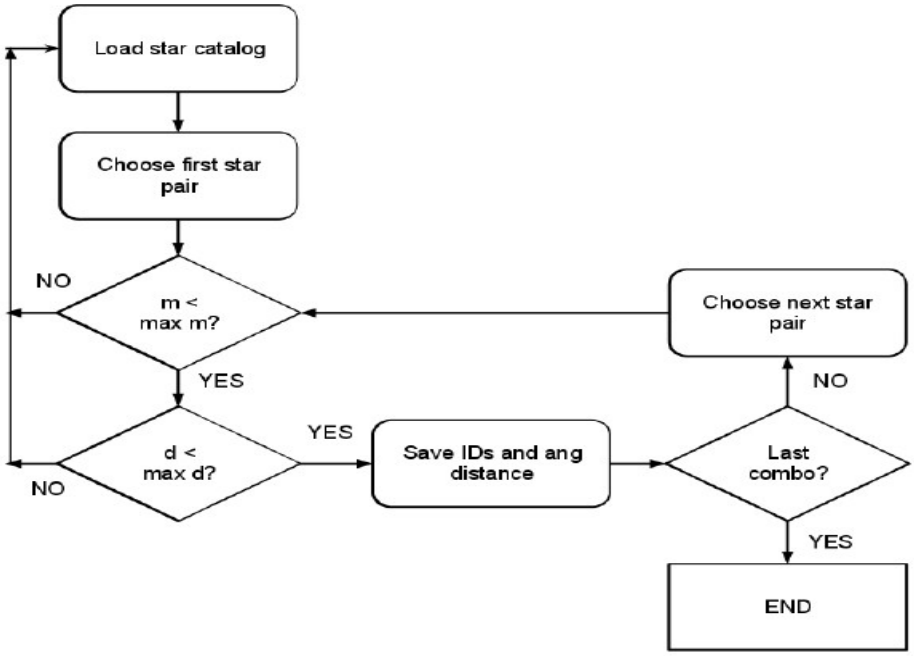


Figura 3.2: Diagrama de flujo de la función catinit

El prototipo de la función tiene la siguiente forma:

```
function [Star_pairs,Unit_vectors,Cat_StarsID]=catinit(starcat, FOV, Mag_limit)
```

Inputs

La función toma como parámetros de entrada una o tres entradas. En los dos casos toma el nombre del fichero con la información del catálogo, ya sea del catálogo completo para calcular los pares de estrellas y sus distancias angulares relativas, o esta información previamente calculada. En el caso de querer generar la información de los pares de estrellas y sus distancias deberemos de aportar además la información del límite de magnitud a partir del cual las estrellas deberán de tenerse en cuenta, y el campo de visión de la cámara, para identificar aquellos pares de estrellas de magnitud superior a la magnitud límite y que puedan ser observadas por la cámara en la misma instantánea.

Outputs

La función *catinit* devolverá tres salidas: la tabla de los posibles pares de estrellas y sus distancias angulares relativas, la lista de los vectores unitarios de las estrellas del catálogo de

estrellas que cumplan el criterio de magnitud, así como los identificadores de dichas estrellas en el catálogo. Si el FOV y la magnitud límite son pasados como parámetros de entrada, es decir, tenemos tres parámetros de entrada, la función guardará el fichero .mat con toda la información antes reseñada y también con las magnitudes de las estrellas del catálogo, información que más adelante utilizaremos para generar imágenes artificiales para realizar pruebas. Esta funcionalidad se ejecutará antes del lanzamiento del satélite.

```
save 'cat_hyg.mat' Star_pairs Unit_vectors Cat_StarsID Magnitudes;
```

Si tan sólo pasamos un parámetro a la función, cargaremos los datos anteriormente guardados en el fichero .mat, sin generar el fichero. Esta funcionalidad será la que desarrolle el satélite en su vida en el espacio.

3.2.2.- FUNCIÓN *loading*

Esta función es la encargada de leer la imagen, ya sea de un fichero o de la cámara. Para nuestro análisis en Matlab, esta función se limitará a leer una imagen desde una fichero pasado como parámetro. La salida de la función consiste en una matriz con los valores de intensidad de la imagen en escala de grises de 8 bits de resolución. El prototipo de la función tiene la siguiente forma:

```
function [Image] = loading(Image_filename)
```

Inputs

La función toma como parámetro de entrada el nombre del fichero que contiene la imagen que posteriormente se va a analizar. Cuando esta función sea implementada en el hardware real del satélite, tomará la imagen real de la cámara.

Outputs

La función *loading* devuelve como única salida una matriz de valores de intensidad en formato de 8 bits sin signo, es decir, en valores de intensidad de 0 a 255, asociados a la imagen tomada.

3.2.3.- FUNCIÓN *centroid*

La función *centroid* toma la matriz generada por la función anterior y devuelve una lista con las coordenadas correspondientes a los centros de las estrellas que se encuentran en la imagen. El diagrama de flujo de esta función viene dado por la figura 3.3, que corresponde con el procedimiento para el algoritmo de centroiding explicado en el apartado 2.2.

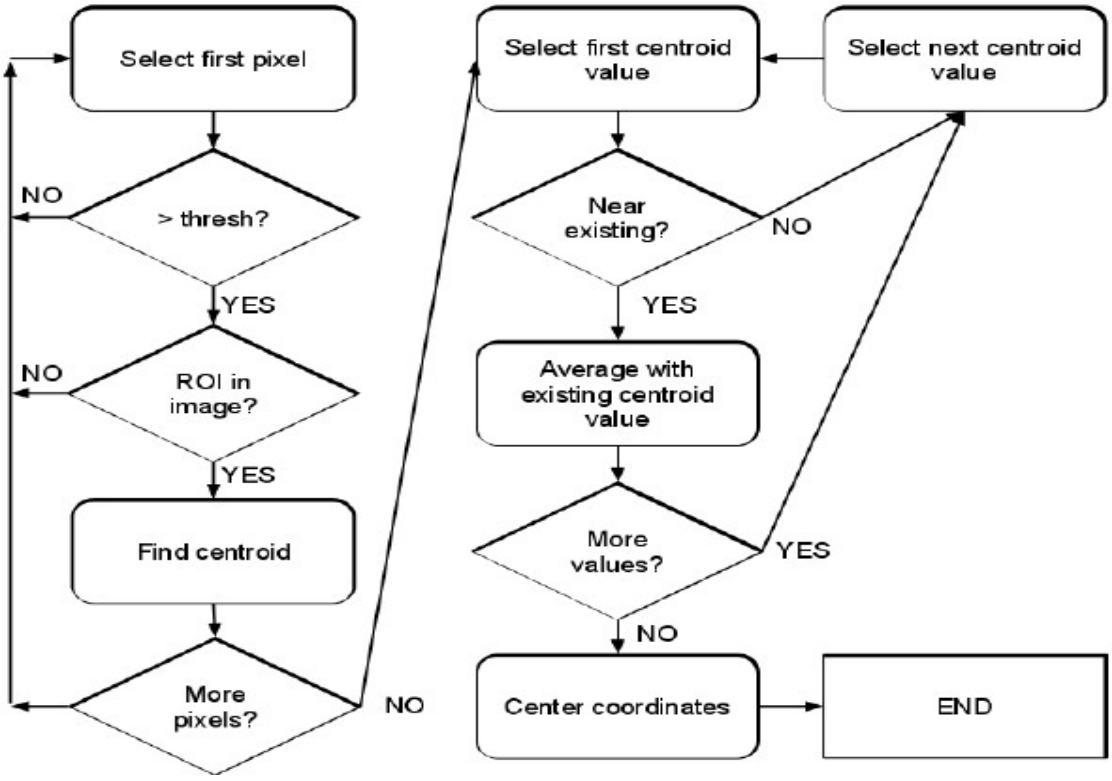


Figura 3.3: Diagrama de flujo de la función centroid

El prototipo de la función tiene la siguiente forma:

```
function [Star_Centers] = centroid(Image, Ithresh, aRoi)
```

Inputs

La función toma como parámetros de entrada la imagen en intensidad de grises de 256 niveles que genera la función *loading*, así como la longitud de la región de interés *aRoi* en pixels (este valor debe de ser siempre un valor impar) y un valor umbral de intensidad *Ithresh* para seleccionar los pixels a tener en cuenta por el algoritmo de centroiding.

Outputs

La función *centroid* devolverá una única salida: la lista con las coordenadas (x,y) en pixels para los centros de las estrellas. A tener en cuenta que para facilitar la programación del algoritmo tomaremos el centro de coordenadas de la imagen en la esquina superior izquierda, aunque para el correcto funcionamiento del algoritmo de identificación de estrellas, las coordenadas de la imagen en pixels se deberán transformar al sistema con referencia en el centro de la imagen.

3.2.4.- FUNCIÓN *uvec*

Esta función toma la lista de coordenadas de la función anterior y la información de la geometría de la cámara y su lente para devolver los vectores unitarios correspondientes a las coordenadas en el sistema de referencia con origen en el centro de la cámara. Como la función *centroid* devuelve las coordenadas en pixels con respecto al borde superior izquierdo, por motivos de facilitar el algoritmo, deberemos en primer lugar realizar la trasformación a un sistema de coordenadas de la imagen con origen en el centro de la imagen. El prototipo de la función tiene la siguiente forma:

```
function [Image_Unit_Vectors] = uvec(image_coordinates, focal, Pixel_Size)
```

Inputs

La función *uvec* tiene tres parámetros de entrada: por una lado la lista de coordenadas dadas por la función anterior, y por otro la longitud focal de la lente de la cámara y el tamaño de los pixels del sensor en milímetros, necesarios para el cálculo de los vectores unitarios, y que serán constantes para una configuración hardware determinada.

Outputs

La función devolverá una única salida que corresponderá con los vectores unitarios en el sistema de coordenadas de la cámara de los centros de estrellas proporcionados por la función *centroid*.

3.2.5.- FUNCIÓN *starid*

Esta función contiene el código que implementa el algoritmo de Voting descrito en la sección 2.3.2. El diagrama de flujo de esta función viene representado en la figura 3.4.

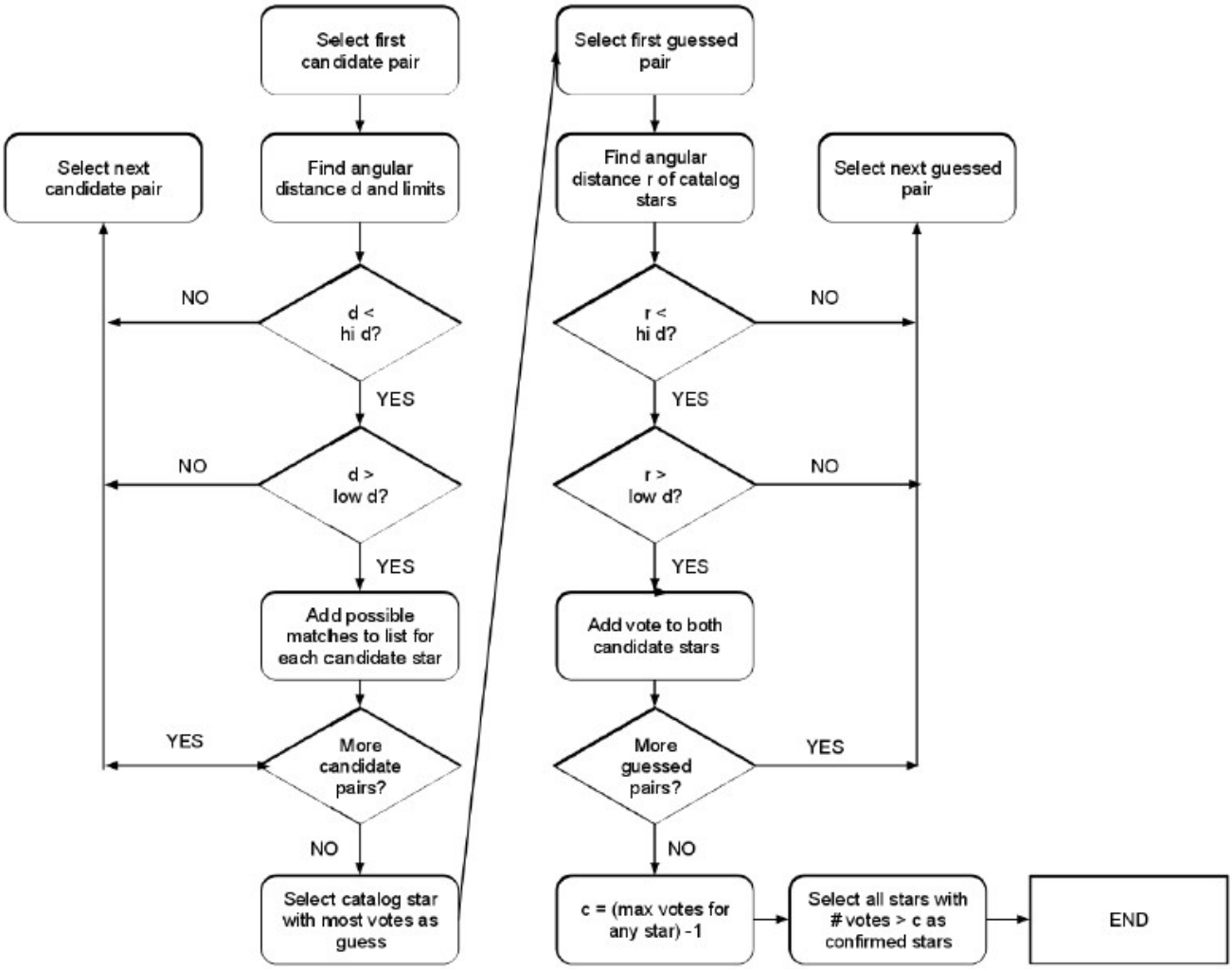


Figura 3.4: Diagrama de flujo de la función starid

La función *starid* toma la lista de vectores unitarios de las estrellas candidatas, información de la geometría de la cámara y la lista de distancias angulares del catálogo para devolver un par de listas con los vectores unitarios de las estrellas reconocidas en el sistema de referencia de la cámara y en el sistema inercial. El prototipo de la función tiene la siguiente forma:

```
function[Candidate_Uvectors,Catalog_Uvectors,Catalog_Star_Ids] =
        starid(Image_UnitVectors,Catalog_StarPairs,Catalog_UnitVectors,Cat_StarsID,
                FOV,Pixel_Size,tol)
```

Inputs

La función *starid* tiene siete parámetros de entrada. Por una lado la lista de vectores unitarios devuelta por la función anterior. Por otro lado, la información del catálogo devuelta por la función *catinit*, es decir, los pares de estrellas con sus distancias angulares relativas, los vectores unitarios y los identificadores de las estrellas que conforman el catálogo. También deberemos de proporcionar información relativa a la cámara y la lente de nuestro *Star Tracker*, como es el campo de visión de la cámara *FOV* y el tamaño del pixel del sensor de la cámara. Por último proporcionaremos la tolerancia para determinar si la distancia entre dos estrellas de la imagen corresponde a la distancia entre dos estrellas del catálogo.

Outputs

La función devolverá tres parámetros de salida. Por un lado, devolverá la lista de los vectores unitarios de las estrellas identificadas en el sistema de coordenadas de la cámara (estrellas candidatas) y la lista de los vectores unitarios de las estrellas del catálogo que el algoritmo ha identificado como correspondientes a las candidatas, en el sistema referencia inercial

Algoritmo

Vamos a hacer una descripción más detallada de este algoritmo, parte esencial de nuestro identificador de estrellas. El proceso de identificación de estrellas del algoritmo de *Voting*, según vimos en el apartado 2.3.2, podemos dividirlo en dos fases:

- *Candidate Matching* (correspondiente al capítulo 2.3.2.2): en esta primera fase, calcularemos la distancia angular entre los vectores unitarios de las estrellas candidatas y compararemos esta distancia con la distancia entre los pares de estrellas de nuestro catálogo. El código empleado para este paso viene representado en la figura 3.5. Para ver el código completo, abrir el código adjunto a la memoria de este proyecto o véase el ANEXO II.

  De este código cabe destacar que el barrido de los pares de estrellas candidatas para el cálculo de sus distancias angulares, al ser idéntica la distancia entre la estrella *i* y *j,* y la distancia entre *j* y *i*, se realiza para tan sólo la mitad de las combinaciones. Usaremos la distancia angular en grados para una mayor claridad en la depuración del algoritmo.

```
for i=1:num_candidates-1
   for j=(i+1):num_candidates % Symetric distance a->b and b->a
        dist = Image_UnitVectors(i,1:3)*Image_UnitVectors(j,1:3)';
        dist_deg = acos(dist)*180/pi;
        if (dist_deg <= FOV)
             start_index = BinarySearch(Catalog_StarPairs(:,3),(dist_deg - tol));
             for h=uint32(start_index):uint32(num_catalog_starspairs)
                 dist_catalog = Catalog_StarPairs(h,3);
                 if ( (dist_deg - tol)<=dist_catalog && dist_catalog<=(dist_deg + tol) )
                     matches(i,Catalog_StarPairs(h,1)) =  matches(i,Catalog_StarPairs(h,1))+1;
                     matches(i,Catalog_StarPairs(h,2)) =  matches(i,Catalog_StarPairs(h,2))+1;
                     matches(j,Catalog_StarPairs(h,1)) =  matches(j,Catalog_StarPairs(h,1))+1;
                     matches(j,Catalog_StarPairs(h,2)) =  matches(j,Catalog_StarPairs(h,2))+1;
                 end
                 if ( dist_catalog>(dist_deg + tol) )
                    break;
                 end
             end
        end
  end % For j
end % For i
```

Figura 3.5: Candidate Matching

La distancia angular entre dos estrellas candidatas *i* y *j* deberá ser inferior al campo de visión de la cámara *FOV*, es decir, deben de ser vistas por la cámara en una misma instantánea.

Buscaremos todos los pares de estrellas del catálogo cuya distancia sea inferior a la calculada para las estrellas candidatas en curso. Esta búsqueda en el catálogo se realiza mediante un algoritmo de búsqueda binaria, que representa una grandísima mejora en el coste computacional de nuestro algoritmo.

El valor de la tolerancia para la identificación de los pares de estrellas candidatas y del catálogo es pasado como parámetro y es determinante a la hora de tener unos buenos resultados. En el siguiente capítulo se explicará como seleccionar esta tolerancia.

En *matches* tendremos para cada estrella candidata un array con la cantidad de veces que ha dado positivo el algoritmo de *matching* por cada estrella del catálogo. Cada posición de este array representa la posición de una estrella en el array de estrellas del catálogo

- *Verificación y Resultado Final* (correspondiente al capítulo 2.3.2.3): en esta segunda fase, eliminaremos las falsas estrellas que el paso anterior nos haya podido devolver. El código empleado para este paso viene representado en la figura 3.6. Para ver el código completo, abrir el código adjunto a la memoria de este proyecto o véase el ANEXO II.

```
% PASOS 1 y 2 del apartado 2.3.2.3
for i=1:num_candidates
  for j=(i+1):num_candidates % Symetric distance a->b and b->a
    dist_cat =
Catalog_UnitVectors(final_matching_index(i),1:3)*Catalog_UnitVectors(final_matching_index(j),1:3)';
    dist_cat = acos(dist_cat)*180/pi;
    dist = Image_UnitVectors(i,1:3)*Image_UnitVectors(j,1:3)';
    dist = acos(dist)*180/pi;
    if ( (dist - tol)<=dist_cat && dist_cat<=(dist + tol) )
         matches_2(i,final_matching_index(i)) =  matches_2(i,final_matching_index(i))+1;
         matches_2(i,final_matching_index(j)) =  matches_2(i,final_matching_index(j))+1;
         matches_2(j,final_matching_index(i)) =  matches_2(j,final_matching_index(i))+1;
         matches_2(j,final_matching_index(j)) =  matches_2(j,final_matching_index(j))+1;
    end
  end % For j
end % For i

% PASOS 3 y 4 del apartado 2.3.2.3
cont=0;
T = max(matches_2(:)) -1;
for k=1:num_candidates
    if (sum(matches_2(k,:)) >  T)
         cont = cont +1;
         Catalog_Star_IDs(cont) =  final_matching_stars(1,k);
         FINAL_index(cont) =  final_matching_index(1,k);
         Candidate_Uvectors(cont,:) = Image_UnitVectors(k,:);
         Catalog_Uvectors(cont,:) = Catalog_UnitVectors(FINAL_index(cont),1:3);
    end
end
```

Figura 3.6: Verificación final

Como en el paso anterior, recorreremos la mitad de las combinaciones de estrellas, debido al simetrismo de las distancias, y transformaremos las distancias a grados. Tras el primer barrido tendremos en *matches_2*, para cada estrella candidata, un array con la cantidad de veces que ha dado positivo el algoritmo de *matching* en la segunda ronda de votos. Tras calcular el valor de *T* como el máximo de los votos recibidos por cualquier estrella menos uno, tendremos finalmente la lista con las estrellas candidatas y sus correspondientes estrellas del catálogo identificadas, eliminadas las falsas estrellas en este segundo paso de la verificación final.

3.2.6.- FUNCIÓN *adet*

Esta función realiza el método de descomposición en valores singulares desarrollado por Markley [7] utilizando las dos listas de vectores unitarios devueltas por la función *starid* y devuelve una matriz directriz coseno con la actitud de la cámara en el sistema inercial. El prototipo de la función tiene la siguiente forma:

```
function [ A ] = adet( Candidate_Uvectors, Catalog_Uvectors )
```

Inputs

La función *adet* tiene dos parámetros de entrada: la lista de vectores unitarios de las estrellas candidatas en el sistema de referencia de la cámara, y la lista de vectores unitarios de las estrellas del catálogo correspondientes a las estrellas candidatas, e identificadas por la función *starid*, en el sistema de coordenadas inercial.

Outputs

La función devolverá como única salida la matriz coseno de transformación entre el sistema de coordenadas de la cámara y el sistema de coordenadas inercial.

4.- RESULTADOS SOFTWARE

Una vez presentados en los capítulos anteriores los algoritmos necesarios para nuestro *Star Tracker*, así como su implementación en Matlab, en este capítulo vamos a aplicar este algoritmo a la configuración hardware que López Suesma [11] propone. López Suesma presenta un desarrollo y un esquema hardware en el que un procesador PIC32MX320F128H de Microchip es el encargado de controlar e implementar toda la algoritmia de nuestro *Star Tracker*, al que se le añade una memoria RAM donde almacenar las imágenes tomadas por el sensor KODAK KC-9618, y una memoria flash donde se almacena el catálogo de estrellas. Basándonos en este modelo hardware, crearemos un modelo que utilizaremos para comprobar la funcionalidad, la eficiencia y la precisión de nuestro algoritmo. En el siguiente apartado se detallan los elementos hardware seleccionados por López Suesma [11] y como modelarlos para nuestras pruebas y simulaciones. En un posterior apartado, realizaremos varias simulaciones y finalmente analizaremos y sacaremos conclusiones sobre el sistema.

4.1.- CONFIGURACIÓN HW

El diagrama de bloques que representa el sistema hardware que López Suesma diseña se presenta en la figura 4.1.

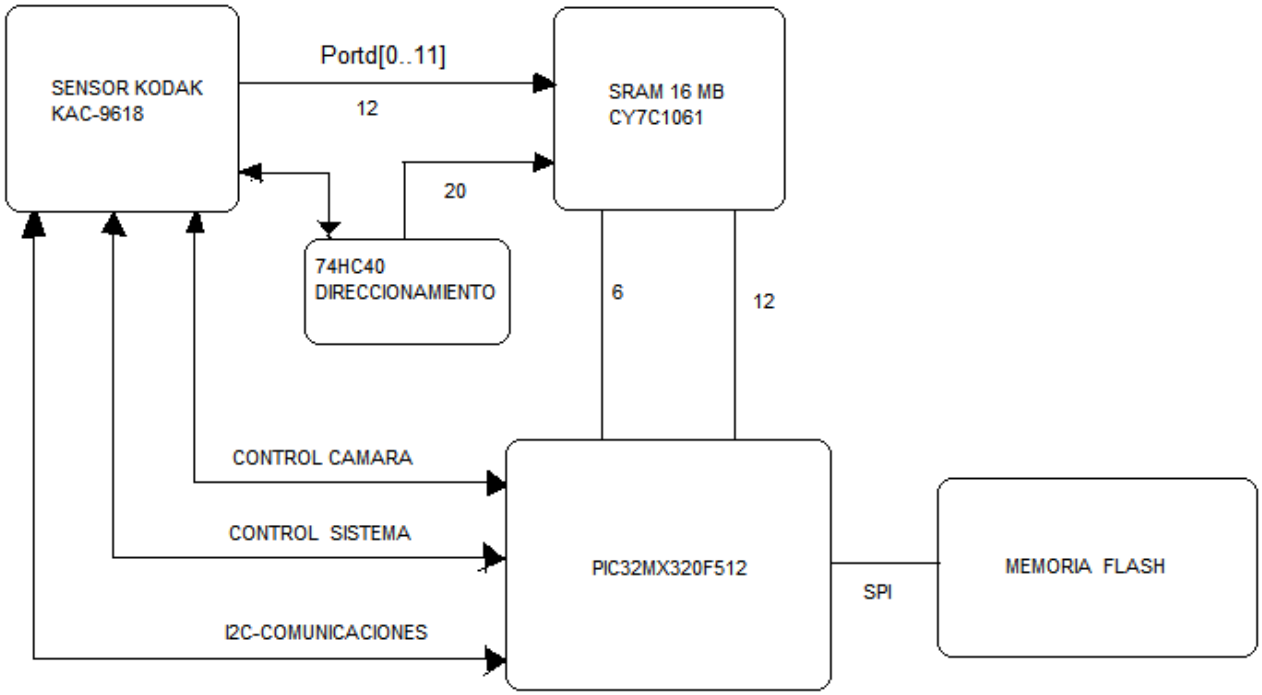


Figura 4.1. Diagrama de bloques del sensor estelar

Con este esquema, el microprocesador PIC32 configura el sensor de la cámara a través del bus asíncrono I2C y mediante señales de control da la orden al sensor para que tome una foto, que es almacenada en la memoria SRAM con la ayuda del contador MM74HC40, que direcciona la memoria. Una vez tomada la imagen, el PIC32 implementará el algoritmo de identificación de estrellas mostrado en los capítulos anteriores, leyendo los datos de la imagen de la SRAM y los datos del catálogo que previamente al lanzamiento se han introducido en la memoria flash, por medio del bus síncrono SPI. Vayamos en detalle con cada uno de los elementos que conforman el sistema.

4.1.1.- SENSOR DE IMAGEN

Es una de las partes más importantes del sistema, pues es el encargado de tomar la imagen que se utilizará para calcular la actitud del Cubesat. El sensor seleccionado se trata de un sensor de tipo CMOS, selección derivada de las ventajas que López Suesma extrae de los estudios de Sferco [12] y Litwiller [13]. El sensor se trata del KODAK KC-9618, cuyas principales características las podemos encontrar en la tabla 4.1.

Array Format	Total: 664H x 504V Active: 648H x 488V
Effective Image Area	Total: 4.98mm x 3.78 mm Active: 4.86 mm x 3.66 mm
Optical Format	1/3“
Pixel Size	7.5μm x 7.5μm
Video Outputs	8,10 & 12 Bit Digital
Frame Rate	30 frames per second
Dynamic Range	62dB in linear mode 110dB in non linear mode
Electronic Shutter	Rolling reset
FPN	0.1%
PRNU	1.5%
Sensitivity	5 V/lux.s
Quantum Efficiency	27%
Fill Factor	47%
Package	48 CLCC
Single Supply	3.3 V +/-10%
Power Consumption	168 mW
Operating Temp	-40 to 85^{O}C

Tabla 4.1. Características del sensor KODAK KC-9618

El diagrama de bloques del sensor viene representado en la figura 4.2. La imagen llega al sensor a través de la lente, que veremos en el siguiente apartado, y es enviada al procesador por medio de 12 señales digitales. El bus I2C se utiliza para configurar las características del sensor. En nuestro caso, la imagen de salida del sensor a través de las 12 señales digitales pasará a la SRAM sin pasar por el procesador, con ayuda de un contador, de tal forma que el procesador lee la imagen de la SRAM directamente. Más información sobre este sensor puede encontrarse en la documentación adjunta.

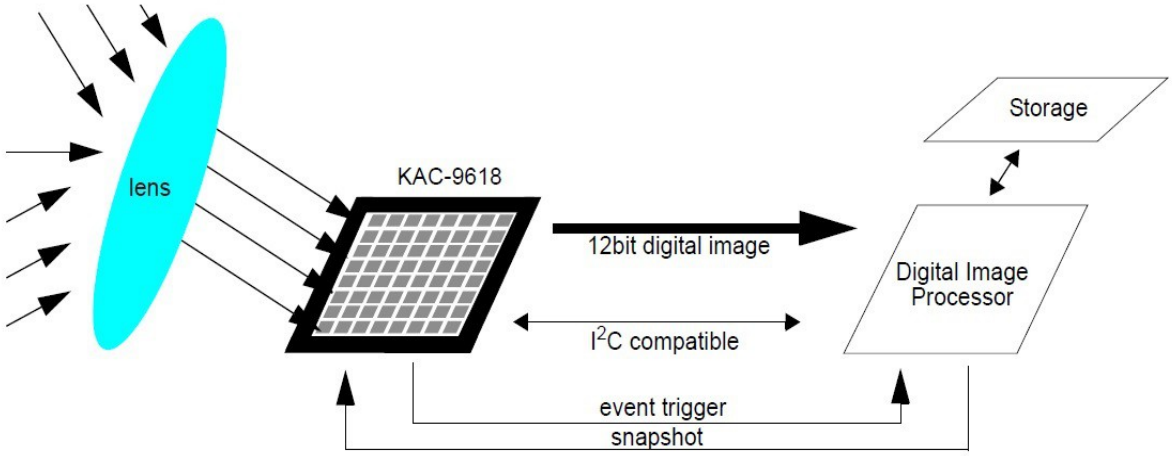


Figura 4.2. Diagrama de bloques del sensor KODAK KC-9618

4.1.2.- LENTE

Aunque López Suesma no selecciona una lente, para poder probar nuestro algoritmo del sensor estelar deberemos de seleccionar una, puesto que es necesario conocer el campo de visión de la lente o *FOV* y la distancia focal *f*, que determinará las estrellas que el sensor es capaz de ver en una sola instantánea. En las características del sensor, vemos que el formato de la óptica asociado es de 1/3". Buscando las soluciones que KODAK como constructor del sensor nos proporciona, encontramos el KODAK KAC-96-1/3" LENS KIT, que se trata de un kit de 4 lentes de 1/3" para la familia de sensores KODAK96xx. La figura 4.3 muestra el aspecto de las 4 lentes.

Figura 4.3. Lentes del KODAK KAC-96-1/3" LENS KIT

Las principales características de las lentes que forman este kit vienen representadas en las tablas 4.2, 4.3, 4.4 y 4.5.

Sensor Size	6.1 mm diagonal
Construction	Single element, no diffractive
Effective Focal Length	6.0 mm
Angle Of View	27º
F/#	F/2.5
MTF	40% at 35 cy/mm on axis 20% at 35 cy/mm at 1.53 mm image height 5% at 35 cy/mm at 2.14 mm image height

Tabla 4.2. Lente 1

Sensor Size	6.1 mm diagonal
Construction	Two element plastic
Effective Focal Length	5.8 mm
Angle Of View	27º
F/#	F/2.5
MTF	50% at 35 cy/mm on axis 40% at 35 cy/mm at 1.53 mm image height 30% at 35 cy/mm at 2.14 mm image height

Tabla 4.3. Lente 2

Sensor Size	6.1 mm diagonal
Construction	Three element plastic
Effective Focal Length	6.4 mm
Angle Of View	55º
F/#	F/2.2
MTF	70% at 35 cy/mm on axis

Tabla 4.4. Lente 3

Sensor Size	6.0 mm diagonal
Construction	Four element Glass
Effective Focal Length	6.0 mm
Angle Of View	60º
F/#	F/2.8
MTF	80% at 35 cy/mm on axis

Tabla 4.5. Lente 4

De las cuatro lentes del kit, decidimos seleccionar la lente 4, con un FOV de 60º, pues cuanto mayor *FOV* mayor cantidad de estrellas tendremos en nuestra imagen, y como veremos más adelante, esto beneficia a nuestro algoritmo de identificación de estrellas. Como datos que serán de

nuestro interés más adelante cabe señalar que esta lente tiene una distancia focal *f* de 6 mm y el tamaño de la diagonal del sensor es de 6 mm.

4.1.3.- MICROPROCESADOR

El microprocesador encargado de controlar el sistema y realizar la algoritmia del sensor estelar es el PIC32MX320F128H, de la familia de procesadores de propósito general de 32 bits del fabricante Microchip. El diagrama de bloques de este procesador se muestra en la figura 4.4.

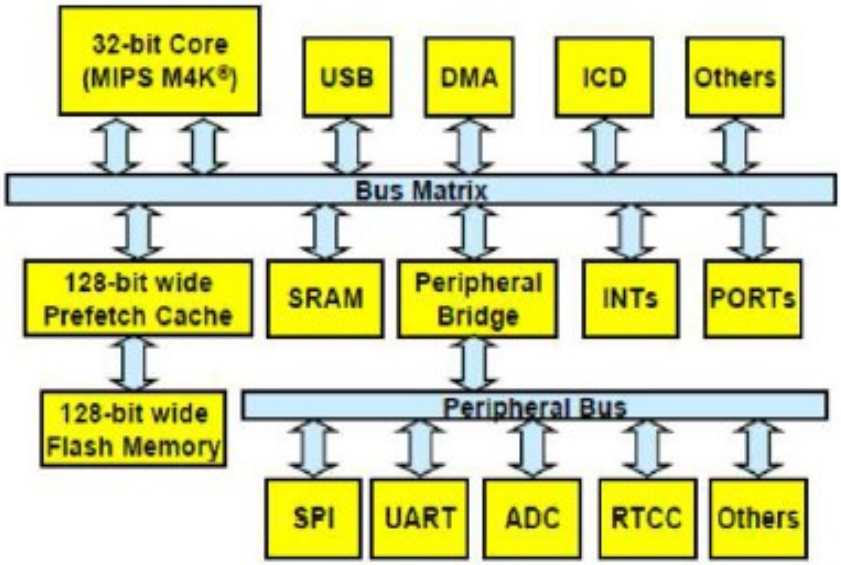


Figura 4.4. Diagrama de bloques del PIC32

Las principales características de este microprocesador son:

- PIC32MX320F128H: 80 MHz 32-bit MIPS, 128K Flash, 16K SRAM.
- 42 I/O pins disponibles.
- 12 Entradas analógicas.
- 3.3V - Voltaje de operación.
- 80Mhz - Frecuencia de trabajo.
- 75mA - Corriente típica de trabajo.
- 7V a 15V - Voltajes de alimentación admisibles (20V máximo)
- +/-18mA DC corriente por pin

A las características anteriores hay que añadir la conectividad UART, SPI, I2C, salidas PWM y entradas analógicas. Este procesador se encuentra también disponible en plataformas de desarrollo tipo Arduino, concretamente la plataforma CHIPKIT, que nos permitirá un prototipado con menor complejidad y coste y un debugeo más sencillo de los algoritmos. Podremos utilizar para su programación la plataforma MPID tipo Arduino, con varias librerías disponibles para el control de sus funcionalidades, o utilizar la herramienta profesional proporcionada por el fabricante Microchip, el entrono de programación MPLAB para la programación en MIPS o en C/C++ con su compilador, y el programador PICkit 3, con toda su potencia.

El núcleo del procesador es un núcleo de 32 bits MIPS M4K basado en arquitectura Harvard, ya que posee buses separados de datos y de instrucciones conectados al *Bus Matrix*. Además de todas las características de la arquitectura PIC32, el verdadero poder del procesador radica en su núcleo, lo cual, junto con su bajo coste, ha hecho que nos decidamos por él para el control del ACS.

4.1.4.- CATÁLOGO DE ESTRELLAS Y MEMORIA FLASH

Para el funcionamiento de nuestro algoritmo de identificación de estrellas del sensor estelar, deberemos de usar información de un catálogo de estrellas, información que modificaremos para presentarla de la forma que más nos convenga, y almacenaremos en el satélite para su posterior uso. Existen varios catálogos de estrellas con información sobre las estrellas que se pueden observar en el firmamento. Utilizaremos el catálogo HYG [15] que utiliza Senabre en [14].

La base de datos HYG es un subconjunto de los datos en tres grandes catálogos: el catálogo Hipparcos, el catálogo de Yale Bright Star (quinta edición), y el catálogo Gliese de estrellas cercanas (tercera edición). Cada uno de estos catálogos contiene información útil para los astrónomos aficionados. El nombre de la base de datos proviene de los tres catálogos que contienen sus datos: Hipparcos, Yale, y Gliese. En total, esta base de datos contiene todas las estrellas que son o más brillante que un cierto corte magnitud (magnitud 7,5 a 9,0) o dentro de 50 parsecs (unos 160 años luz) del Sol. La versión actual, v3.0, no tiene límite de magnitud: cualquier estrella en Hipparcos, Yale o Gliese está representada. La base de datos es un archivo de valores separados por comas (CSV) que se puede importar a la mayoría de los programas de bases de datos y hojas de cálculo. Los campos de cada estrella de la base datos son los siguientes:

1. StarID: Es la clave principal de la base de datos, a partir de una "base de datos máster" más grande de las estrellas. Usaremos este campo para identificar las estrellas localizadas por el sensor estelar.
2. HD: Identificación en el catálogo de Henry Draper de la estrella, si se conoce.
3. HR: Identificación en el catálogo de Harvard Revisado, que es el mismo que su número en el catálogo de Yale Bright Star de la estrella.
4. Gliese: Identificación en la tercera edición del Catálogo Gliese de Estrellas, se encuentra cerca de la estrella.
5. BayerFlamsteed: Designación Bayer / Flamsteed, de la quinta edición del Catálogo de Yale Bright Star. Esta es una combinación de las dos designaciones. El número Flamsteed, si está presente, se da primero; luego una abreviatura de tres letras de la letra griega Bayer; el número superíndice Bayer, si está presente; y finalmente, la constelación de abreviatura de tres letras. Así Alpheratz tiene el valor del campo "21Alp Y", y kappa1 Sculptoris (sin número Flamsteed) tiene "Kap1Scl".
6. RA, DEC: ascensión recta de la estrella y declinación, para la época 2000.0. Estrellas presentes sólo en el Catálogo Gliese, que utiliza coordenadas 1950.0, han tenido estas coordenadas precesión a 2.000.
7. ProperName: Nombre común para la estrella, como "Estrella de Barnard" o "Sirius". Estos nombres están principalmente tomados del sitio web del proyecto Hipparcos, que enumera los nombres representativos de las 150 estrellas más brillantes y muchas de las 150 estrellas más cercanas.
8. Distancia: La distancia de la estrella en parsecs, la unidad más común en la astronomía. Para convertir parsecs a años luz, se multiplica por 3.262.
9. Mag: magnitud visual aparente de la estrella.
10. AbsMag: magnitud visual absoluta de la estrella (su magnitud aparente desde una distancia de 10 parsecs).
11. Spectrum: tipo espectral de la estrella, si se conoce.
12. ColorIndex: índice de color de la estrella (magnitud azul - magnitud visual), cuando sea conocido.

Como hemos visto en el apartado 2.3.2.1, para la generación del catálogo que guardaremos en nuestro sensor estelar, sólo necesitaremos la información del ID de la estrella, su magnitud, y la ascensión recta y la declinación. Nuestro catálogo, derivado del HYG, constará de los pares de estrellas con una magnitud superior a un umbral, así como su distancia angular relativa, que

calcularemos en función de su ascensión recta y declinación. Por tanto, nuestra función *catinit*, tomará los campos StarID, Mag, RA y DEC de la base de datos HYG para crear una nueva base de datos con un listado de pares de estrellas con sus distancias relativas y los StarID de las estrellas cuya magnitud sea superior al umbral.

Como describe Sidi [16] en su anexo 4, para determinar la posición de una estrella en el firmamento en coordenadas ecuatoriales, cuyo esquema podemos observar en la figura 4.5, se usan dos ángulos:

1. La ascensión recta RA, medida desde el equinoccio de primavera o punto Aries en el plano ecuatorial de la esfera celeste. Este dato en el catálogo HYG se mide en horas, minutos y segundos.
2. La declinación DEC, está comprendida entre ±90º medidos desde el plano ecuatorial, siendo positivo el ángulo en dirección norte.

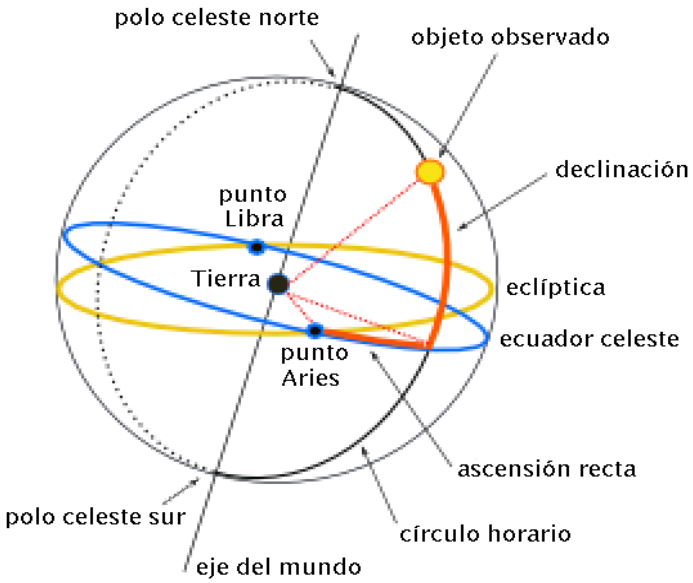


Figura 4.5. Sistema de coordenadas ecuatoriales

En astronomía, una época es un momento específico en el tiempo para el que se especifican las coordenadas celestes o elementos orbitales, y a partir de las cuales otros parámetros orbitales se calculan con el fin de predecir la posición futura. La época actual estándar se llama *"J2000.0"* y corresponde aproximadamente al mediodía del 1 de enero 2000, en el calendario gregoriano, en el Observatorio Real de Greenwich, en Londres (Inglaterra). Esto es equivalente a:

1. La fecha juliana 2.451.545,0 (Tiempo Terrestre).
2. 1 de enero 2000, 11: 59:. 27.816 TAI (Tiempo Atómico Internacional)
3. 1 de enero 2000, 11: 58: 55,816 UTC (Tiempo Universal Coordinado).

Cuando fechas u horas se expresan en años con una fracción decimal de *J2000*, los años son de exactamente 365,25 días, que es la duración media de un año en el calendario juliano. La ascensión recta y la declinación se toman con respecto *J2000*. Convertiremos la ascensión recta dada en horas, minutos y segundos en grados sabiendo que el mapa celestial se divide en 24 horas, una hora se divide en 60 minutos y un minuto en 60 segundos, y una hora equivale a 15º.

La información del catálogo la guardaremos antes del lanzamiento en una memoria flash. López Suesma selecciona una memoria AT45DB081D de ADESTO Technologies. La memoria flash es un chip semiconductor, que permite guardar datos conservando su contenido incluso cuando la batería se apaga, y que ademas, permite ser borrado aplicándole una carga eléctrica. Las dos principales características desde el punto de vista de datos son:

- Número de grabaciones de datos/byte: 100.000 ciclos (típico).
- Mas de 100 años de retención de datos

Las memorias Flash se utilizan en múltiples dispositivos, tales como ordenadores portátiles, teléfonos móviles, cámaras fotográficas digitales, organizadores personales, receptores GPS, equipos de red, y son el medio mas adecuado para transportar información ya que su estado sólido las hace resistentes a golpes y vibraciones. La memoria utilizada tiene un interface SPI que permite utilizar pocos elementos de interconexión. El tipo de dispositivo utilizado permite almacenar hasta 1 Mbyte de datos. Si para almacenar el catálogo fuese necesario un mayor tamaño, podremos colocar varias de estas memorias flash en el bus SPI, seleccionando en cada caso cual queremos utilizar mediante su entrada CS (Chip selector) de selección de dispositivo en el bus SPI. Los datos de esta memoria los podemos encontrar en el ANEXO III y en la documentación adjunta.

4.1.5.- MEMORIA RAM

La imagen del firmamento que toma el sensor se guardará temporalmente en una memoria SRAM para su posterior uso en el algoritmo de identificación de estrellas. La memoria SRAM que se utilizará será la CY7C1061AV33 de tecnología CMOS, que se organiza en 1.048.576 palabras de 16 bits. La SRAM dispone de 20 pines para la codificación de la dirección de memoria que se desea leer o escribir, con lo que necesitaríamos una mayor cantidad de pines de I/O en nuestro procesador. Para evitar esto, usaremos dos contadores de 12 bits que seleccionarán todas las direcciones de la memoria. Estos contadores usarán como señal de reloj de entrada el pin *pclk* del sensor KODAK-9618 El resto de pines de control de la SRAM serán controlados directamente por el microprocesador. El esquema de conexión de la memoria se presenta en la figura 4.6.

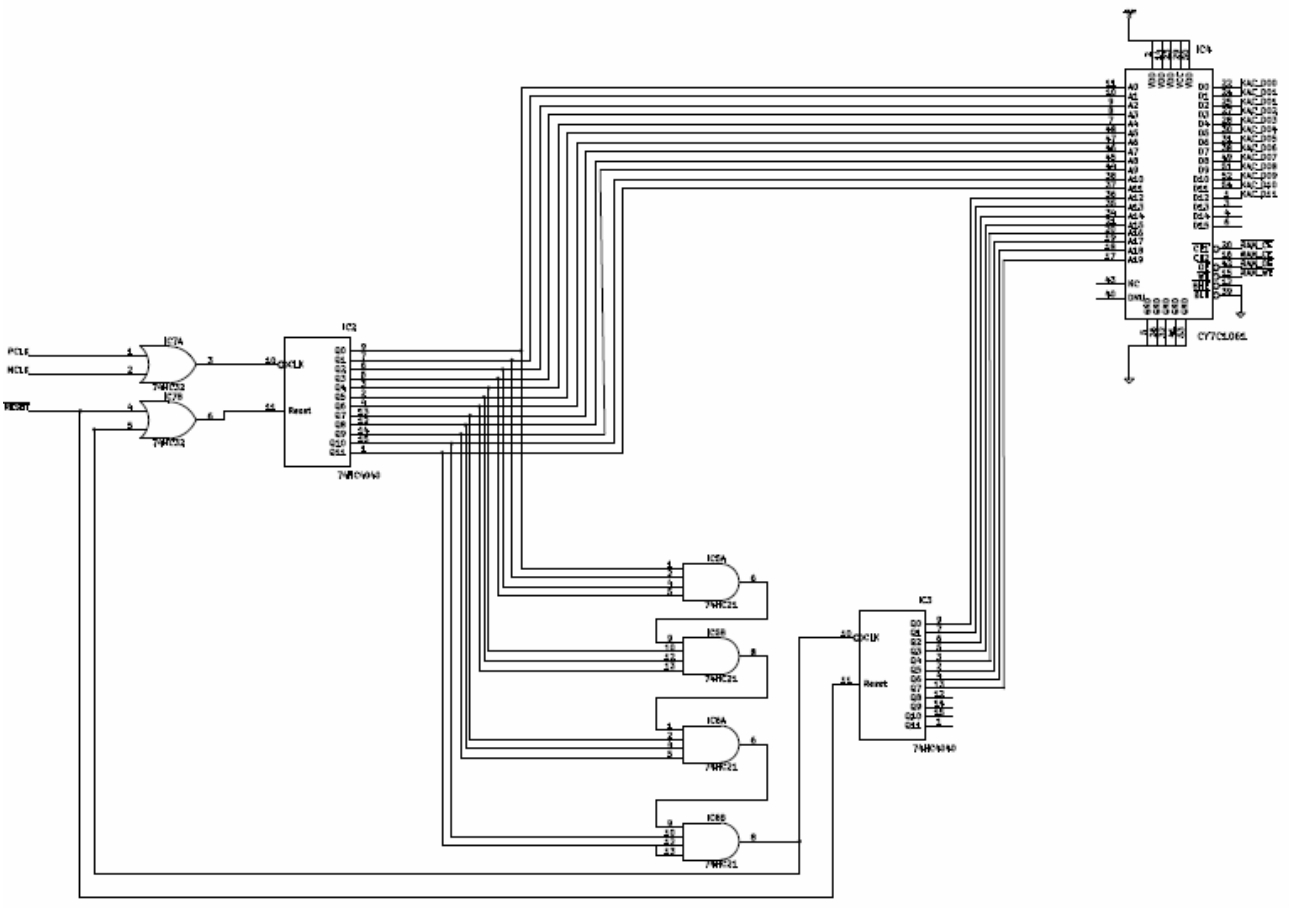


Figura 4.6. Acceso a memoria SRAM CY7C1061

Las características de la SRAM y los contadores pueden ser encontradas en el ANEXO IV y en la documentación adjunta.

4.2.- SIMULACIONES

Una vez tenemos implementado nuestro algoritmo de identificación de estrellas y cálculo de actitud en Matlab, vamos a comprobar su correcto funcionamiento para la configuración hardware propuesta anteriormente. Introduciremos en todas las funciones del algoritmo los parámetros del sensor y lente seleccionados. Generaremos un catálogo de estrellas formado por pares de estrellas y distancias relativas a partir del catálogo HYG general. A partir de este catálogo y de la configuración de la cámara y lente generaremos una imagen sintética con una actitud aleatoria. Pasaremos esta imagen por nuestro algoritmo y compararemos el resultado con los datos que ya conocemos sobre la imagen que hemos generado, y haremos un análisis de la precisión del algoritmo. Veamos en detalle este procedimiento.

4.2.1.- RANDOM ATTITUD

El primer paso para generar una imagen sintética es generar una actitud aleatoria. Lo haremos por medio de tres ángulos aleatorios entre -180 y 180 grados. El generador de números aleatorios de Matlab devuelve valores comprendidos entre 0 y 1, por lo que utilizaremos la función 4.1 para convertir esos valores α_1, α_2 y α_3 en tres ángulos aleatorios θ_1, θ_2 y θ_3 en el rango deseado.

$$[\theta_1 \ \theta_2 \ \theta_3] = ([\alpha_1 \ \alpha_2 \ \alpha_3] - [0.5 \ 0.5 \ 0.5]) * 180^{\circ} \tag{4.1}$$

La codificación de la fórmula 4.1 en Matlab tiene el siguiente aspecto:

```
% Random Attitude
alpha = [rand rand rand];
theta = (alpha - [0.5 0.5 0.5]) * (180);
theta = deg2rad(theta);
```

A partir de estos ángulos, calcularemos la matriz directriz coseno realizando una rotación 3-2-1 sobre los ángulos θ_1, θ_2 y θ_3, resultando una matriz de rotación desde el sistema inercial al sistema de coordenadas de la cámara, utilizando la fórmula 4.2.

$$R_I^C = \begin{bmatrix} cos\theta_1 & sin\theta_1 & 0 \\ -sin\theta_1 & cos\theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos\theta_2 & 0 & -sin\theta_2 \\ 0 & 1 & 0 \\ sin\theta_2 & 0 & -cos\theta_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\theta_3 & sin\theta_3 \\ 0 & -sin\theta_3 & cos\theta_3 \end{bmatrix} \tag{4.2}$$

La codificación de la fórmula 4.2 en Matlab tiene el siguiente aspecto:

```
% Direction cosine matrix
R_1 = [ cos(theta(1))    sin(theta(1))    0;
       -sin(theta(1))    cos(theta(1))    0;
        0                0                1];
R_2 = [ cos(theta(2))    0               -sin(theta(2));
        0                1                0;
        sin(theta(2))    0                cos(theta(2))];
R_3 = [ 1                0                0;
        0                cos(theta(3))    sin(theta(3));
        0                -sin(theta(3))   cos(theta(3))];
R_I_C = R_1 * R_2 * R_3;
```

Esta matriz determina la actitud de la imagen generada y la compararemos con la matriz A dada en la formula 2.27 y que nuestro algoritmo genera. Si nuestro algoritmo funciona correctamente, el error entre la matriz A calculada y la matriz generadora de la imagen deberá ser inferior a una tolerancia que posteriormente fijaremos, para dar por bueno nuestro algoritmo.

4.2.2.- IMAGE GENERATION

Una vez se ha seleccionado la actitud de la imagen a generar según el procedimiento anterior, definiremos los parámetros de la cámara, con su sensor y lente necesarios. A partir de las características del sensor KODAK KC-618 y de la lente seleccionada en el capítulo anterior, obtenemos la tabla 4.6 con los parámetros que necesitaremos para la generación de la imagen.

LENS	Aperture diameter, d(mm)	6
	Field-of-view FOV (º)	60
	Focal length, f(mm)	6
	Transmittance, τ	1
SENSOR	Resolution (pixels)	648H x 488V
	Pixel size (μm)	7,5
	Quantum efficiency	0,27
	Dynamic range(dB)	62
	Spectral range, λ(μm)	0,6
	Saturation limit, l*sat* (photons)	13500

Tabla 4.6. Características del sensor KODAK KC-9618 y su lente

Como veremos a continuación, para poder generar la imagen sintética, deberemos de tener un catálogo con las estrellas que tengan una magnitud superior a un valor dado. Cuanto mayor sea la magnitud de una estrella menos brillará en el firmamento, por tanto, cuanto mayor sea nuestra magnitud límite, mayor será la cantidad de estrellas de nuestro catálogo. Cuantas más estrellas tengamos, más estrellas seremos capaces de generar y posteriormente identificar, mejorando la precisión del algoritmo, pero aumentando el procesamiento y el tamaño de la memoria de almacenamiento. Por tanto, buscaremos un tamaño que sea lo más eficiente posible. Para estas simulaciones hemos elegido un valor de magnitud límite de 3,75, que nos da como resultado un total de 385 estrellas. La figura 4.7 muestra la distribución de estas estrellas en el plano RA-DEC junto con el StarID que identifica la estrella en el catálogo.

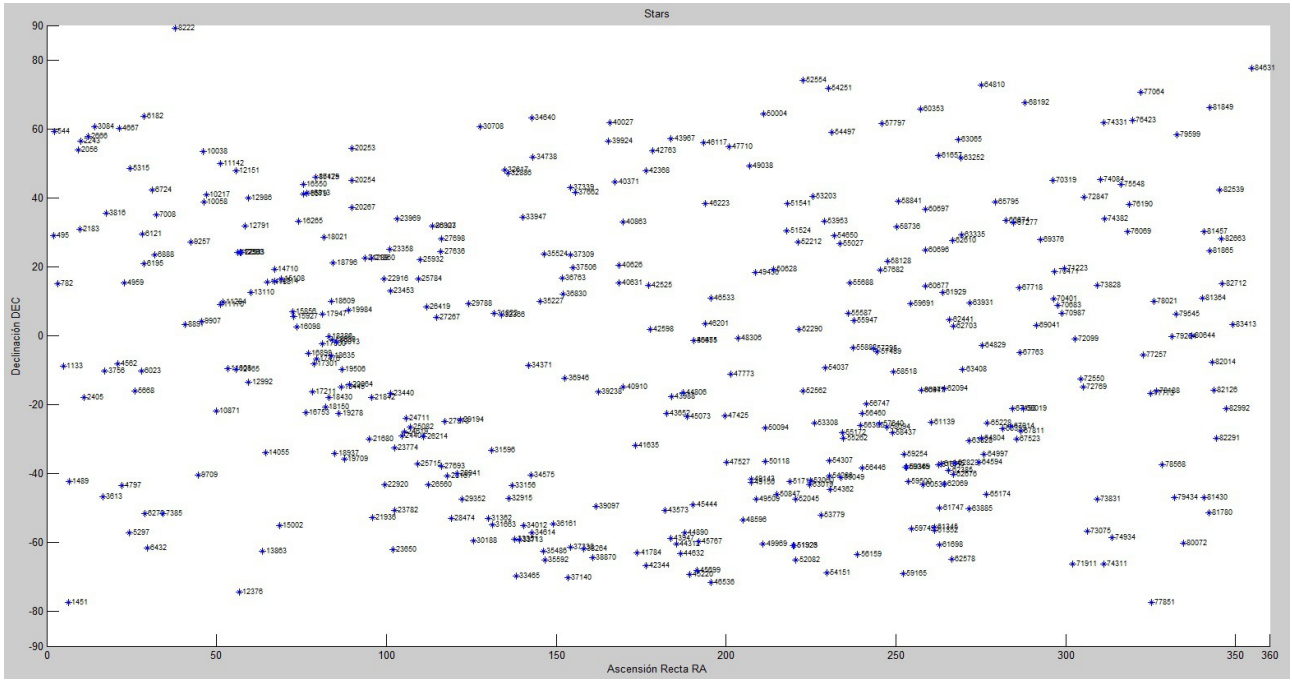


Figura 4.7. DEC vs. RA para límite magnitud = 3,75

Podemos observar como hay algunas zonas del firmamento que no tienen muchas estrellas, sobre todo las correspondientes a declinaciones cercanas a ±90º. Las imágenes con una actitud en estas zonas tendrán pocas estrella y es posible que el error en el cálculo de la actitud sea elevado, o que incluso no se pueda determinar, pero decidimos fijar este límite de magnitud que hace que el catálogo tenga un tamaño manejable, pues con 395 estrellas y un FOV de 60º, los posibles pares de estrellas cuya distancia sea inferior a ese FOV son 19.177, que ya es una cantidad considerable, y con un crecimiento exponencial al aumentar el número de estrellas, como podemos observar en la figura 4.8 para un límite en la magnitud de 5, donde tendríamos 1.637 estrellas y 342.664 pares de estrellas.

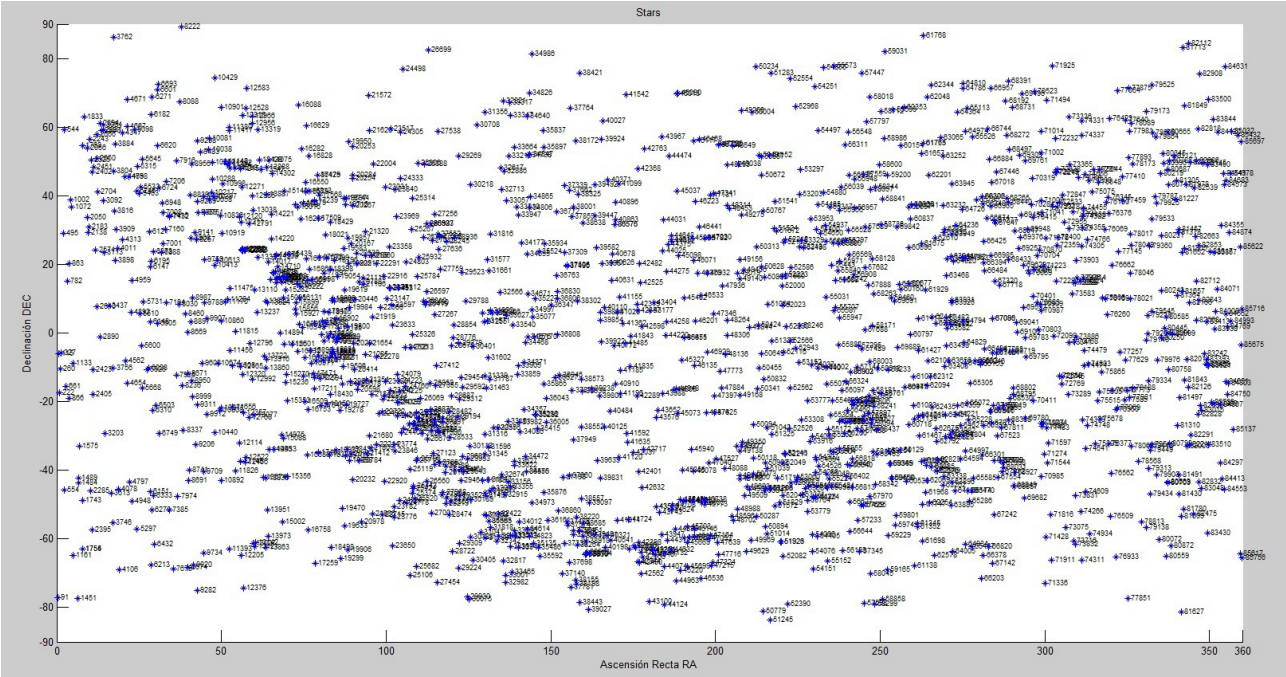


Figura 4.8. DEC vs. RA para límite magnitud = 5

La generación de la imagen consta de los siguientes pasos, cuya codificación en Matlab se representa en los cuadros que acompañan a cada fórmula:

1. Para cada actitud generada según el método del capítulo anterior, calcularemos el vector unitario u_{bore} a lo largo de la dirección que apunta al centro de la tierra o *boresight*, en el sistema de coordenadas inercial, dado por la ecuación 4.3.

$$u_{bore} = (R_I^C)^T [0\ 0\ 1]^T \tag{4.3}$$

```
u_bore = R_I_C' * [0 0 1]';
```

2. Se verifica cada estrella del catálogo para comprobar si cae dentro del campo de vista *FOV* de la cámara, usando la condición expresada en la ecuación 4.4.

$$u_{bore}^T u_{star} > \cos(\frac{\theta_{FOV}}{2}) \tag{4.4}$$

```
dist = u_bore' * u_hyg(k,:)';
dist_deg = acos(dist)*180/pi;
if ( dist_deg < (FOV/2) )...
```

3. Si la estrella cae dentro del campo de vista FOV, su vector unitario se convierte al sistema de coordenadas de la cámara mediante la ecuación 4.5

$$u_{star}, c = (R_I^C) u_{star} \tag{4.5}$$

```
u_star_C  = R_I_C * u_hyg(k,:)';
```

4. Calcularemos las coordenadas de la estrella en el plano de la imagen *(u,v)* mediante las ecuaciones 4.6 y 4.7. $(u_{star}, c)_x$ es el *x*-valor del vector unitario de la estrella en el plano de la cámara, y de forma similar obtenemos $(u_{star}, c)_y$ y $(u_{star}, c)_z$.

$$u = -f\left(\frac{(u_{star}, c)_x}{(u_{star}, c)_z}\right) \tag{4.6}$$

$$v = -f\left(\frac{(u_{star}, c)_y}{(u_{star}, c)_z}\right) \tag{4.7}$$

```
u(cont) = focal*(u_star_C(1)/u_star_C(3));
v(cont) = focal*(u_star_C(2)/u_star_C(3));
```

5. La función de generación de imágenes requiere conocer por cuantos puntos n_p deberemos dividir los fotones de la estrella, así como el factor de desenfoque *b*. Tras varias pruebas, para esta simulación tomaremos unos valores de n_p = 50 y b = 0.015. El tiempo de exposición t_{exp} también será necesario. Fijaremos este valor en 0.15 segundos.

6. El número de fotones S que la estrella genera se puede calcular usando los parámetros del sensor y de la lente así como la magnitud *m* de la estrella, usando la ecuación 4.8.

$$S = (t_{\exp})\left(\frac{\pi}{4} d^2\right)(\tau)(\lambda)\left(10^{-0.4\mathrm{m}}\right) \tag{4.8}$$

```
S = t_exp * (pi*(Aperture_diameter^2)/4) * tau * lammda * (10^(-0.4*Mag(k)));
```

7. La estrella es ahora desenfocada asignando a cada uno de los puntos n_p una posición aleatoria *(u',v')* cerca de la localización de la estrella en la imagen *(u,v)*.

$$[u'\ v']^T = [ub \cdot rand(-.5,.5)\ vb \cdot rand(-.5,.5)]^T \tag{4.9}$$

```
for j=1:np
    [u_prima(j),v_prima(j)] = rand_circ(1,u(cont),v(cont),b);
    x = round((u_prima(j)/pixel_size_x)+(Resolution_x/2));
    y = round((v_prima(j)/pixel_size_y)+(Resolution_y/2));
```

8. Para cada pixel de la matriz de intensidad L, si un punto *(u',v')* cae dentro de la imagen, añadimos S/n_p fotones a ese punto.

```
if (x>0 && x<Resolution_x && y>0 && y<Resolution_y)
        IMG(x,y) = IMG(x,y) + (S/np);
end
```

9. Finalmente, normalizamos la matriz de intensidad $\tilde{L}$ por el límite de saturación l_{sat}.

$$\begin{cases} \tilde{L}(i,j) = \dfrac{L(i,j)}{l_{sat}}, & \dfrac{L(i,j)}{l_{sat}} \leqslant 1 \\ \tilde{L}(i,j) = 1, & \dfrac{L(i,j)}{l_{sat}} > 1 \end{cases} \tag{4.10}$$

```
for i=1:Resolution_x
      for j=1:Resolution_y
            if((IMG(i,j)/l_sat) <= 1)
                  IMG(i,j) = IMG(i,j)/l_sat;
            end
            if((IMG(i,j)/l_sat) > 1)
                  IMG(i,j) = 1;
            end
      end
end
```

En $\tilde{L}$ y en IMG para nuestro algoritmo en Matlab, tendremos la matriz de intensidad con las estrellas que la cámara vería si apuntase en la actitud aleatoria generada previamente. Las imágenes así generadas tienen el aspecto de la figura 4.9, y serán las que pasemos a nuestro algoritmo, el cual deberá de ser capaz de identificar las estrellas que en ella están contenidas, y calcular la actitud, que deberá coincidir con la utilizada para generar la imagen.

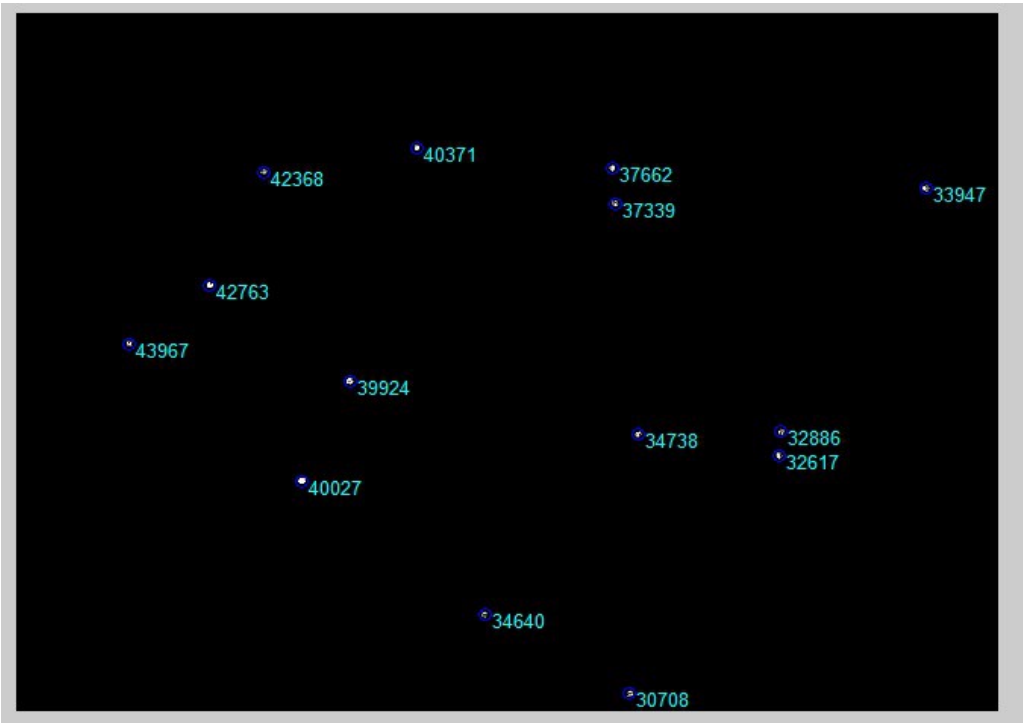


Figura 4.9. Ejemplo de imagen generada con los IDs de las estrellas

4.2.3.- EJECUCIÓN DEL ALGORITMO DEL STAR TRACKER

Ya tenemos creadas las funciones de nuestro algoritmo de *Star Tracker*, y tenemos generada una imagen con una determinada actitud. Pasaremos ahora esta imagen por nuestro algoritmo y compararemos la actitud calculada con la original de la imagen. Para ello será necesario llamar a las funciones con los correspondientes parámetros para el sensor KODAK KC-9618 y su lente. En el caso de la memoria SRAM y la flash con el catálogo no es necesario modelarlo pues estamos comprobando la precisión del algoritmo en sí, y estos dos aspectos no influyen en la funcionalidad del algoritmo, en todo caso en el tiempo de procesamiento, y vamos a suponer que ambas memorias son accesibles a través de variables de programa. En el caso real, estas variables se deberán de sustituir por llamadas a funciones que recuperen el valor de la memoria SRAM a través de las líneas de control y de la memoria flash a través del bus SPI.

Para el posterior análisis, repetiremos este proceso un número n de veces y guardaremos la información necesaria en ficheros. El código Matlab que gestiona el algoritmo del *Star Tracker*, y que corresponde con el diagrama de flujo de la figura 3,1 viene recogido en ANEXO II. Veamos más a fondo los puntos más importantes de este algoritmo, representado el código Matlab asociado a cada paso:

1. Definición de parámetros del sensor KODAK KC-9618 y la lente necesarios para el funcionamiento de las funciones posteriores.

```
FOV = 60;            % deg
Mag_limit = 3.75;
focal = 6;           % mm
pixel_size =0.0075; % mm
Resolution_y = 648; % pixels
Resolution_x = 488; % pixels
d = 6000;            % Aperture diameter (6 mm)
```

2. Llamada a *catinit.* Mediante la variable *GenerateCatalog* controlaremos si vamos a generar el catálogo, para lo que será necesario pasar a la función el catálogo HYG completo así como la información del FOV y el límite de magnitud de las estrellas, o vamos a cargarlo de un fichero generado previamente. El primer caso es necesario si no se ha generado nunca o si cambiamos algún parámetro y se realizaría antes del lanzamiento del satélite.

```
% Read Catalog
if (GenerateCatalog == 1)
   [Cat_starPairs,Cat_Uvect,Cat_StarsID] = catinit('hygfull.csv',FOV,Mag_limit);
else
   [Cat_starPairs,Cat_Uvect,Cat_StarsID] = catinit('cat_hyg.mat');
end
```

3. Generar y cargar imagen. Mediante la variable GenerateImage controlaremos si generamos o no la imagen. La imagen generada se guarda en la carpeta *'Fotos'* con el nombre de *Imagen_1.bmp*. Y posteriormente se cargará en la variable *Image* con la función *loading*. Si deseamos analizar una imagen sin necesidad de generarla sintéticamente, colocaremos el nombre correspondiente en la función loading.

```
% Generate Image
if (GenerateImage == 1)
  [IMG,RIC,Original_StarsID,h_1]  = Generate_Image( Resolution_x, Resolution_y,
                                                    pixel_size, FOV, focal, d );
end
% Load Image
Image = loading( 'Fotos/Imagen_1.bmp' );
```

4. Calcular centroides de la imagen. Para ello definiremos los valores de I_{thresh} y a_{Roi} necesarios, que en este caso tomarán el valor de 100 y 11 respectivamente, tras varias pruebas para determinar empíricamente el valor más adecuado. La función *centroid* nos devolverá un array con las posiciones de los centroides de las estrellas localizadas en la imagen.

```
Ithresh = 100;
aRoi = 11; % Must to be odd values
% Calculate Centroides
Star_Centers = centroid( Image, Ithresh, aRoi );
```

5. Calcular vectores unitarios de las estrellas de la imagen con la función *uvec*

```
% Calculate Unit Vectors of image stars
Image_Unit_Vectors = uvec( Star_Centers, focal, pixel_size );
```

6. Identificación de estrellas con el método *Voting* mediante la función *starid*

```
% Star Matching: Voting method
diagonal = sqrt(Resolution_x^2 + Resolution_y^2);
tol = FOV / diagonal;
[Candidate_Uvectors,Catalog_Uvectors,Catalog_Star_IDs ] =
              starid( Image_Unit_Vectors, Cat_starPairs, Cat_Uvect, Cat_StarsID,
                      FOV, pixel_size, tol );
```

7. Determinación de la actitud con la función *adet*

```
% Attitude Determination
A = adet( Candidate_Uvectors, Catalog_Uvectors );
```

4.3.- ANÁLISIS Y CONCLUSIONES

Nuestro algoritmo nos da como resultado la actitud del satélite expresada mediante la matriz directriz coseno $\boldsymbol{A}$. Vamos a calcular el error entre la actitud medida que nos devuelve el algoritmo y la actitud inicial con la que se ha generado la imagen sintética. De esta manera podremos comprobar cuanto de bueno es nuestro algoritmo. El error para cada uno de los ejes se calcula según las ecuaciones 4.11 a 4.14.

$$\Theta = R_{calc}^{-1} R_{actual} = \begin{bmatrix} \approx 1 & -\phi_z & -\phi_y \\ \phi_z & \approx 1 & -\phi_x \\ \phi_y & \phi_x & \approx 1 \end{bmatrix} \tag{4.11}$$

$$\varepsilon_x = 90^{\circ} - \cos^{-1}\phi_x \tag{4.12}$$

$$\varepsilon_y = 90^{\circ} - \cos^{-1}\phi_y \tag{4.13}$$

$$\varepsilon_z = 90^{\circ} - \cos^{-1}\phi_z \tag{4.14}$$

El código de Matlab para el cálculo del error cometido en el cálculo de la actitud se puede ver a continuación. Multiplicamos el valor obtenido en las fórmulas 4.12, 4.13 y 4.14 por 3.600 para obtener el resultado en arco-segundos.

```
% Error Calculation
Attitude_Error = inv(A)*RIC;
Err_x = abs(3600*(90 - rad2deg(acos(Attitude_Error(3,2))))); % arcseconds
Err_y = abs(3600*(90 - rad2deg(acos(Attitude_Error(3,1))))); % arcseconds
Err_z = abs(3600*(90 - rad2deg(acos(Attitude_Error(2,1))))); % arcseconds
```

Para probar nuestro algoritmo, generaremos una imagen y la pasaremos por nuestros procedimientos para calcular el error, en un bucle de 100 iteraciones. Los resultados obtenidos de este proceso los guardamos en una carpeta que contendrá, ordenados en sub-carpetas, los resultados de cada una de las imágenes, para un posterior análisis más detallado del proceso. Por cada imagen guardaremos la imagen sintética generada y otra con los centroides localizados. También guardaremos un fichero por cada imagen con los Ids de las estrellas generadas y los Ids de las estrellas identificadas, así como el error en el cálculo de la actitud. También generaremos un fichero con todos los valores de error en los tres ejes, para un mejor tratamiento posterior, y un resumen de la simulación al completo.

Consideraremos que nuestro algoritmo falla si el error en cualquiera de los tres ejes es superior a los 100 arco-segundos. Este límite es considerado por McBride [2] como razonable, teniendo en cuenta que todos los errores provienen del algoritmo en sí, ya que no se han incluido errores físicos en la cámara.

Tras realizar una simulación, obtenemos los siguientes resultados generales:

```
Estrellas Generadas:   1167
Estrellas Localizadas: 1070
Error Medio X: 2.598057e+001
Error Medio Y: 1.594060e+001
Error Medio Z: 1.891790e+001
Error Medio TOTAL: 2.027969e+001
TOTAL Imágenes OK: 95
```

Vemos como de un total de 1.167 estrellas generadas en las 100 imágenes sintéticas, hemos sido capaces de identificar 1.070, con un error medio total de 20.28 arco-segundos,

significativamente inferior al límite de 100 arco-segundo que nos hemos impuesto. Nuestro algoritmo es capaz de encontrar la actitud en un 95% de los casos. Analicemos más en detalle estos resultados.

Mediante el script *calculos.m* de Matlab generamos una serie de gráficas que nos van a ayudar a interpretar los resultados de la simulación. En primer lugar, vemos en la figura 4.10 como la mayoría de las veces, la cantidad de estrellas identificadas en cada imagen, se encuentra en unos valores de entre 7 y 15 estrellas. Como mínimo, nuestro algoritmo ha sido capaz de encontrar dos estrellas en dos imágenes distintas, y un máximo de 24 estrellas en una única imagen.

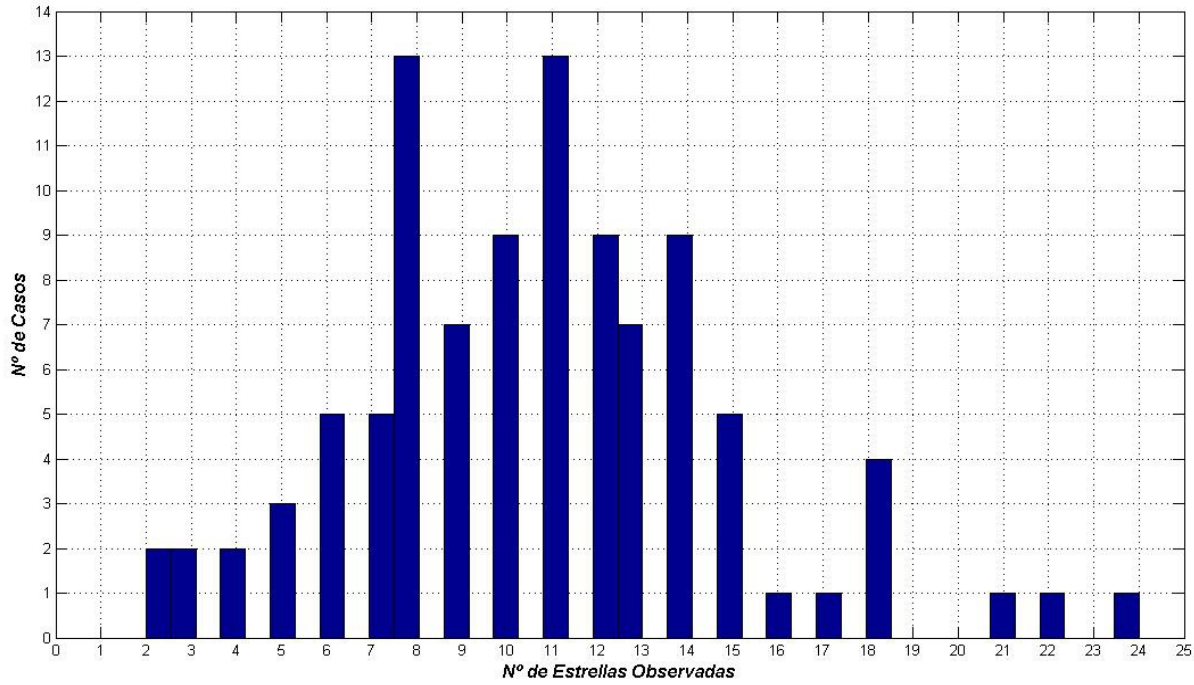


Figura 4.10. Número de casos por cada número de estrellas identificadas

Por otro lado, veamos como se distribuye el error en los tres ejes. Si observamos la figura 4.11, observamos como la gran mayoría de los errores se encuentran por debajo de los 30 arco-segundos, lo que da evidencia de la precisión del algoritmo. Tan sólo podemos ver algunos errores dispersos, sobre todo en el eje X, que en un caso llegan a superar los 200 arco-segundos de error.

Con la información de estas dos gráficas, vamos a estudiar la influencia que tiene el número de estrellas identificadas con el error en el cálculo de la actitud. Si observamos la figura 4.12, podemos observar como a mayor cantidad de estrellas identificadas, menor es el error en el cálculo de la actitud, algo que era de esperar.

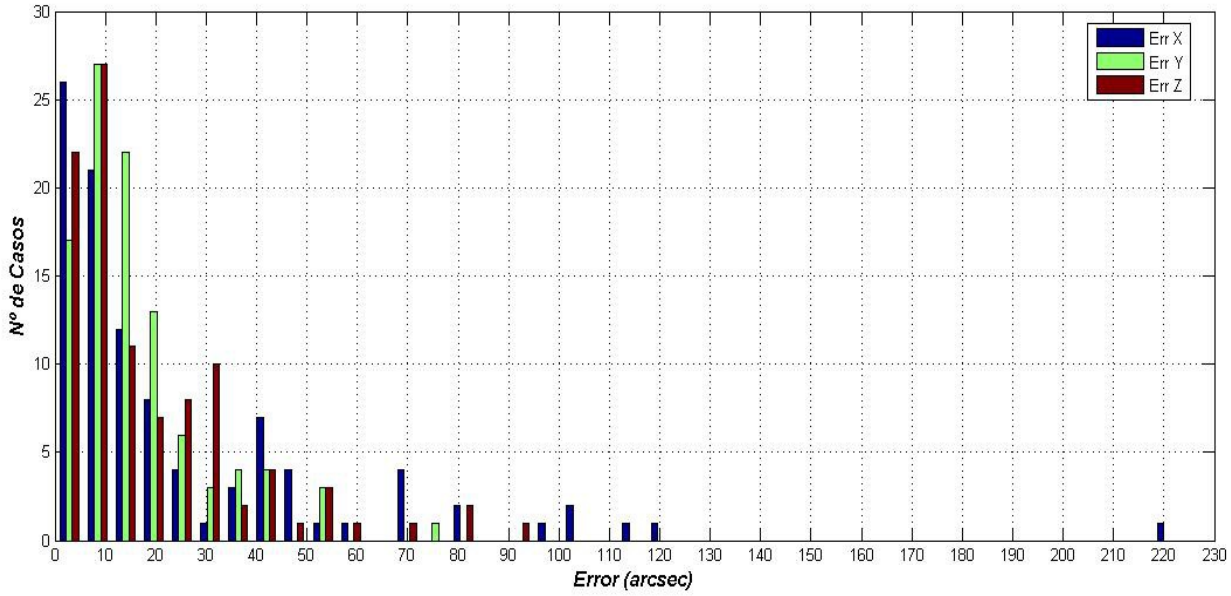


Figura 4.11. Número de casos por cada error en el cálculo de la actitud.

La figura 4.13 presenta un valor promediado de los errores de todas aquellas imágenes que tienen el mismo número de estrellas identificadas, es por eso que no observamos un pico de 200 arco-segundos, como hemos podido observar en el eje X en la figura 4.11.

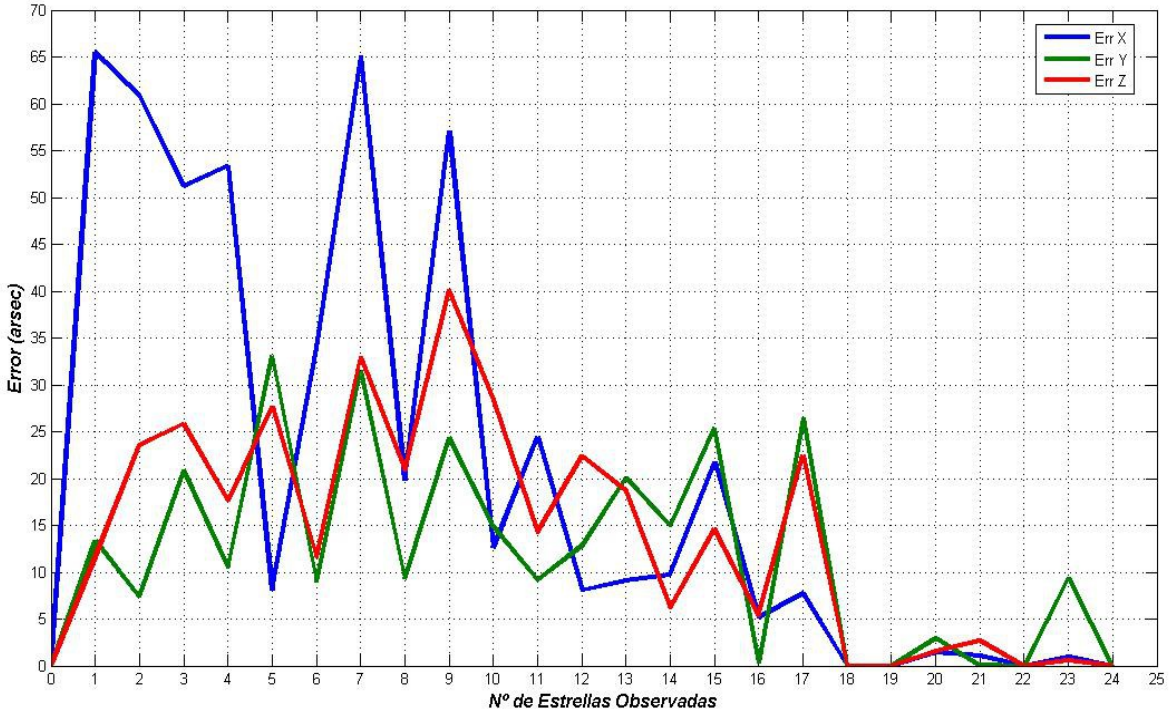


Figura 4.12. Error en la actitud por número de estrellas identificadas y eje.

La figura 4.13 muestra el promedio de error en los tres ejes distribuido en la cantidad de estrellas observadas por cada imagen. El promedio nunca supera los 45 arco-segundos, y se reduce considerablemente a partir de la identificación de al menos 10 estrellas en la imagen, reduciéndose el error medio por debajo de los 20 arco-segundos.

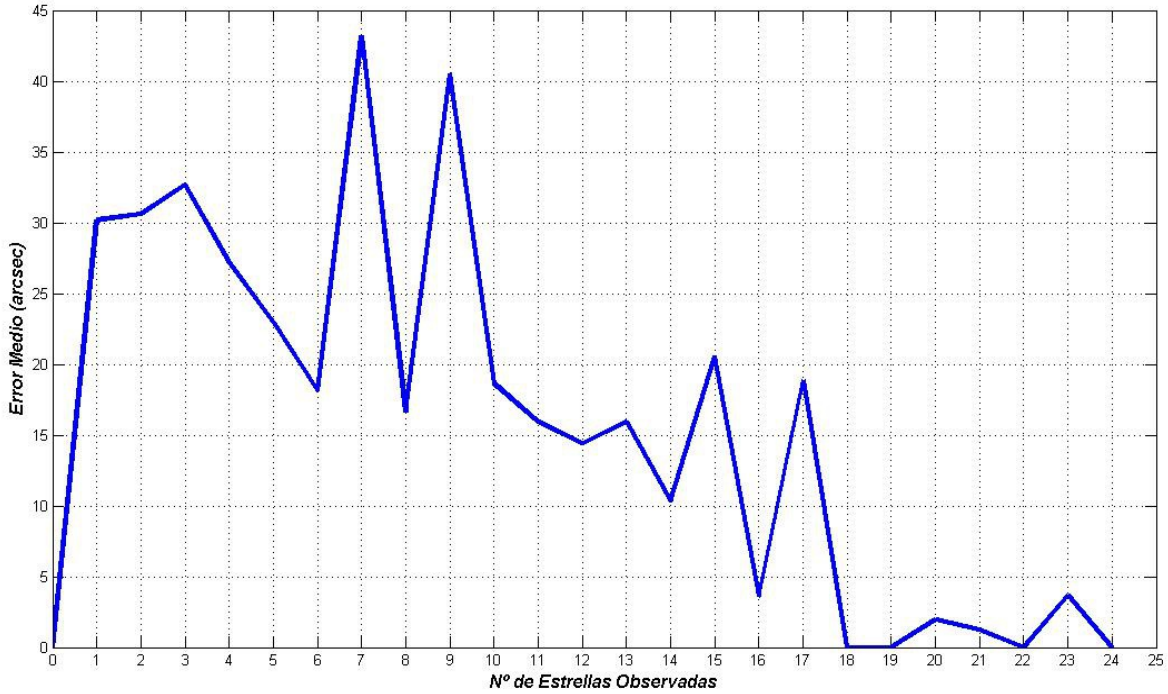


Figura 4.13. Error promedio en la actitud por número de estrellas identificadas.

Como ya hemos mencionado anteriormente, a mayor número de estrellas identificadas, más preciso es el algoritmo. Podemos conseguir un mayor número de estrellas aumentando el FOV de la cámara, o también podemos aumentar la magnitud máxima de las estrellas que queremos identificar. En contraposición, un mayor número de estrellas a identificar, supone un mayor tamaño en catálogo de estrellas a almacenar y una mayor cantidad de cálculos a realizar.

Veamos por qué nuestro algoritmo tiene un error en algún eje por encima de 100 arco-segundos. Localizamos la imagen cuyo error en el eje X supera los 200 arco-segundos y que hemos visto en la figura 4.10. Se trata de la imagen número 20 cuyos resultados particulares podemos obtener en la carpeta *Results_20*. Veamos los datos reflejados en su fichero *Result_20.tx*t:

```
Original:      495    782    2183   81364  81865  82014  82712  83413
Calculated:    495    782    83413
Error X-Y-Z:   2.234293e+002        8.755041e+000        3.028698e+001
```

Vemos como de las ocho estrellas que estaban presentes en la imagen sintética original, hemos sido capaces de identificar tan sólo tres, lo que hace que el error en el cálculo de la actitud se incremente considerablemente. Si comparamos estos resultados con los de la imagen 22, que presenta un error muy bajo, vemos como la cantidad de estrellas identificadas es altísima, alcanzándose las 15 estrellas, lo que confirma nuestra tesis de mayor número de estrellas identificadas, menor error en el cálculo de la actitud.

Original:	32617	32886	34640	34738	37339	37662	39924	40027	40371	40863	42368
	42763	43967	46117	47710							
Calculated:	32617	32886	34640	34738	37339	37662	39924	40027	40371	40863	42368
	42763	43967	46117	47710							
Error X-Y-Z:	8.161565e+000			1.369921e+001			1.399651e+001				

La figura 4.14 muestra la imagen generada con las estrellas que el algoritmo de cetroiding ha sido capaz de encontrar, y que sus vectores unitarios posteriormente se pasan al algoritmo de reconocimiento de estrellas, para las imágenes 20 y 22.

Claramente se ve como las estrellas generadas son menores en el caso de la imagen 20, y además el algoritmo de identificación sólo es capaz de reconocer 3, por lo que el error se incrementa. Hay zonas de la esfera celeste que presentan menor densidad de estrellas, como se pudo ver en la figura 4.7.

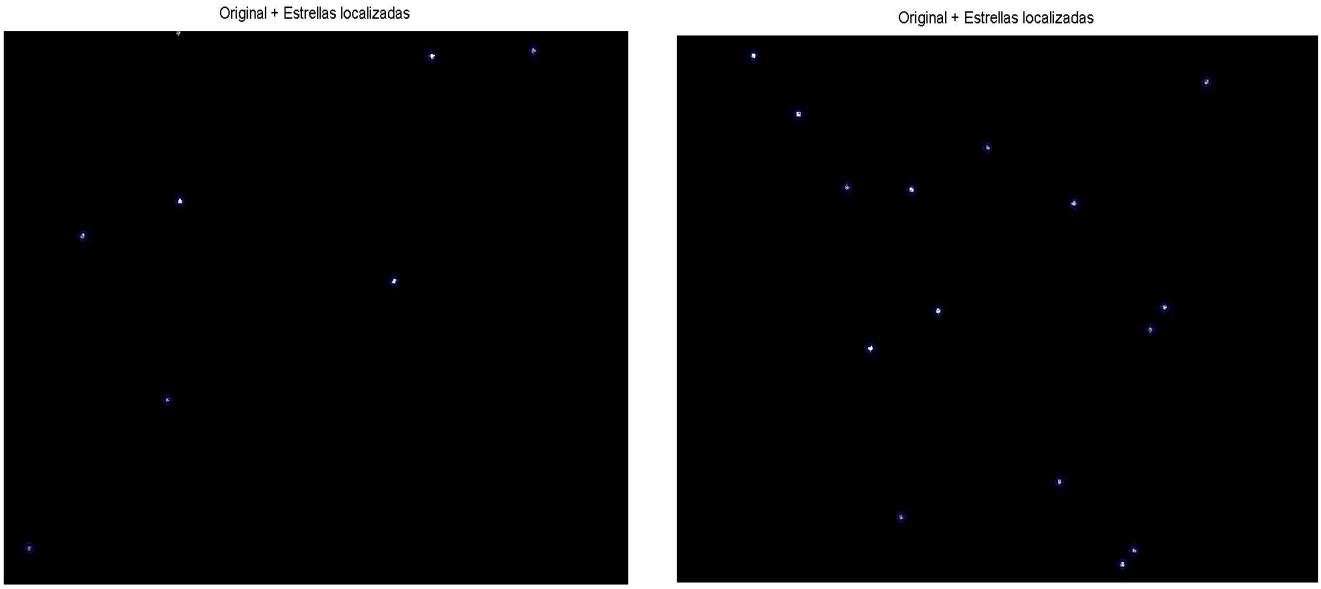


Figura 4.14. Izda: Imagen 20; Dcha: Imagen 22.

5.- SISTEMA DE PRUEBAS

En los capítulos anteriores hemos visto como implementar el software de nuestro sensor estelar y hemos simulado y demostrado su funcionalidad para una configuración hardware mediante el uso de Matlab. Vamos ahora a probar nuestro sistema con una configuración hardware real. En los siguientes apartados detallaremos el sistema hardware de pruebas utilizado.

5.1.- INTRODUCCIÓN

El sistema hardware que hemos utilizado para simular y probar nuestros algoritmos de identificación de estrellas es el propuesto por López Suesma [11] en su proyecto. En este proyecto se presentan los esquemas de fabricación de la PCB con los componentes seleccionados. No vamos a crear una PCB prototipo basada en este diseño, pues el coste de su fabricación y posibles re-fabricaciones hasta dar con el diseño definitivo no son objetivo de este trabajo. Sin embargo, vamos a crear un sistema similar y equivalente utilizando elementos *low cost* basados en la filosofía open-hardware, open-software y la tendencia a hacer uno mismo lo que necesita, do-it-yoursef DIY, que hoy en día tanta aceptación tiene, como las plataformas Arduino, Raspberri-Pi, Beaglebone...

Vamos a ver primero los elementos hardware seleccionados, como se conectan entre sí para crear un sistema similar y equivalente al propuesto por López Suesma, para posteriormente desarrollar el software que los controla así como el software propio de nuestro sensor estelar.

5.2.- HARDWARE

Como hemos comentado, el objetivo de este sistema de pruebas es crear un sistema equivalente al utilizado anteriormente, pero con elementos *low cost*, sin crear ninguna placa prototipo PCB. Para ello, nos vamos a basar en la plataforma de prototipado CHIPKIT UNO32, que consiste en una placa de desarrollo compatible con Arduino, pero con la potencia de un procesador de 32 bits de Microchip, concretamente el PIC32MX320F128, que es exactamente el procesador elegido por López Suesma. Al tratarse del mismo procesador, podremos establecer un paralelismo en el funcionamiento de ambos sistemas. Su programación la haremos con el entorno MPIDE basado en el entorno de propagación de Arduino, que gracias a sus librerías nos va a facilitar considerablemente el trabajo.

Una vez seleccionado el procesador, montado en la plataforma CHIPKIT UNO32, repasamos las características principales para observar que posee 16 KBytes de memoria RAM para datos y 128 KBytes de flash para programa. Ambas no son suficientes para almacenar una imagen ni el catálogo de estrellas, por lo que vamos a tener que añadir una RAM y una EEPROM externas para almacenar la imagen tomada por la cámara y el catálogo de estrellas. La memoria RAM la conectaremos al procesador a través del bus SPI y la EEPROM a través del bus I2C. Por su parte, elegiremos la cámara OV7670+FIFO, que conectaremos al Uno32, encargado de su configuración vía I2C y de almacenar la imagen en la RAM. Podemos ver cómo están conectados todos estos elementos en el esquema representado en la figura 5.1.

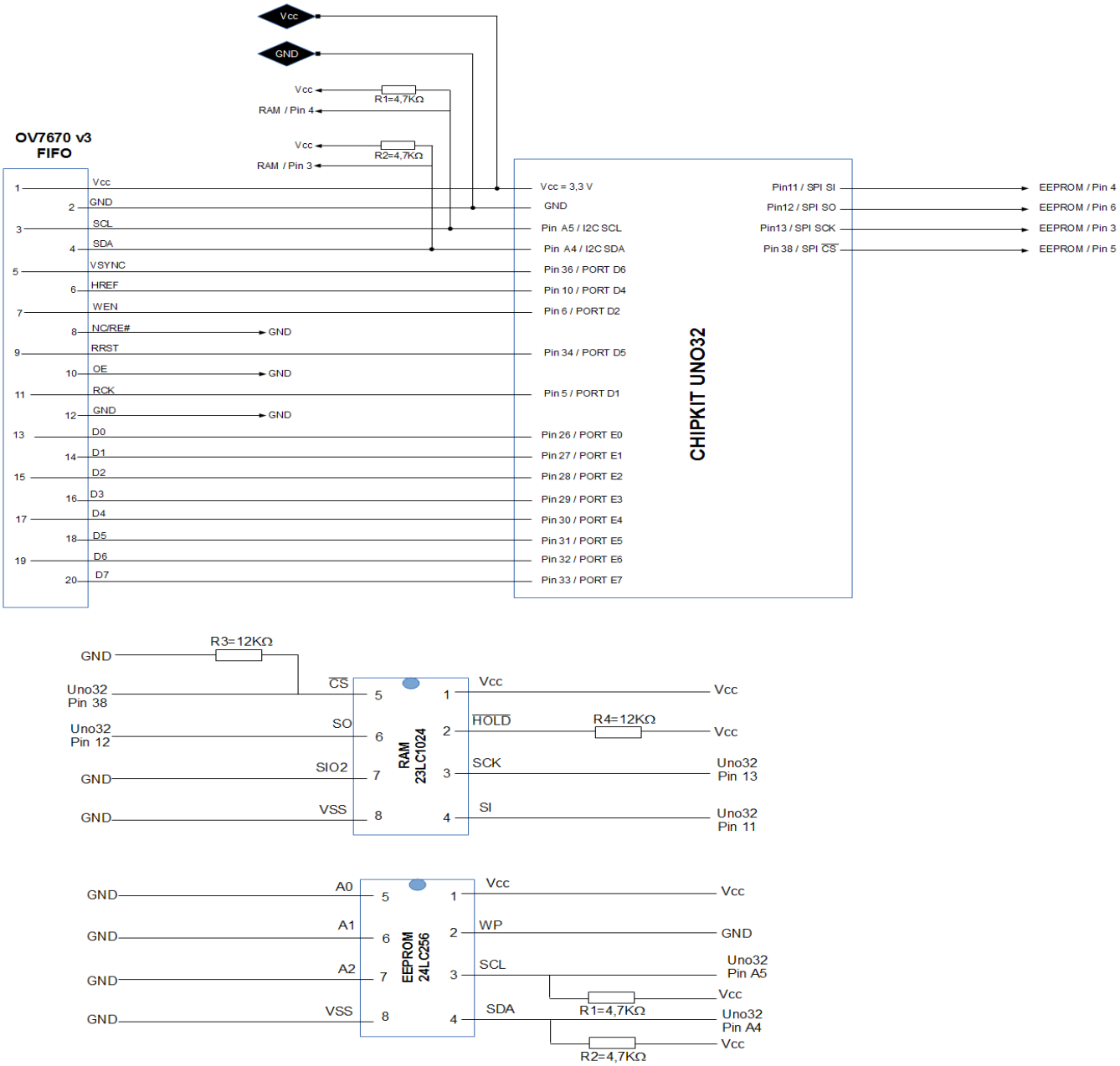


Figura 5.1. Esquema General del Hardware de Pruebas

En los siguientes apartados vamos a detallar cada uno de los componentes del sistema.

5.2.1.- PROCESADOR

Vamos a utilizar la plataforma CHIPKIT UNO32 que se basa en la popular plataforma Arduino de hardware abierto para prototipos, pero añade el rendimiento del microcontrolador Microchip PIC32. El Uno32 presenta el mismo factor de forma que la placa Arduino Uno. Cuenta con una interfaz de puerto serie USB para la conexión con el IDE y puede ser alimentado a través de USB o una fuente de alimentación externa.

Development Board	 chipKIT™ UNO32
Code Development and Programming	
MPIDE (Arduino™-based)	✓
MPLAB IDE (MIPS Assembly and C)	Requires PICkit™ 3 (PG164130)
Hardware Connectivity	
Arduino™ Form Factor	✓ (Arduino UNO)
Pmod™ Connectors	Expansion Shield Available
Available I/O Pins	42
Analog Pins	12
USB (Host/Device,OTG)	
Ethernet	
CAN	
R/C Servo Connectors	
UART/SPI/I²C™	2/2/2
	All platforms feature
Operational Characteristics	
Performance	80 MHz (1.56 DMIPS/MHz)
Program Flash (KB)	128
SRAM Data (KB)	16

- Microchip® PIC32MX320F128H microcontroller (80 Mhz 32-bit MIPS, 128K Flash, 16K SRAM)
- compatible with many existing Arduino code samples and other resources
- Arduino Uno form factor
- compatible with many Arduino shields
- 42 available I/O pins
- two user LEDs
- PC connection uses a USB A > mini B cable
- 12 analog inputs
- 3.3V operating voltage
- 80Mhz operating frequency
- 75mA typical operating current
- 7V to 15V input voltage (recommended)
- 20V input voltage (maximum)
- 0V to 3.3V analog input voltage range
- +/-18mA DC current per pin

Figura 5.2.- Aspecto del CHIPKIT UNO32 y sus características principales

La placa Uno32 aprovecha la potencia del microcontrolador PIC32MX320F128. Este microcontrolador utiliza instrucciones de 32 bits MIPS con un núcleo del procesador corriendo a 80Mhz, 128K de memoria de programa Flash y 16K de memoria de datos SRAM.

El Uno32 se puede programar utilizando un entorno basado en el original de Arduino IDE modificado para dar soporte PIC32. Además, el Uno32 es totalmente compatible con el avanzado Microchip MPLAB® IDE y el programador / depurador PICkit3.

El Uno32 ofrece 42 pins de E/S que admiten una número de funciones periféricas, tales como UART, SPI, I2C y salidas de ancho de pulso modulado PWM. Doce de los pines de E/S se puede utilizar como entradas analógicas o como entradas y salidas digitales. La figura 5.3 muestra los pines del UNO32.

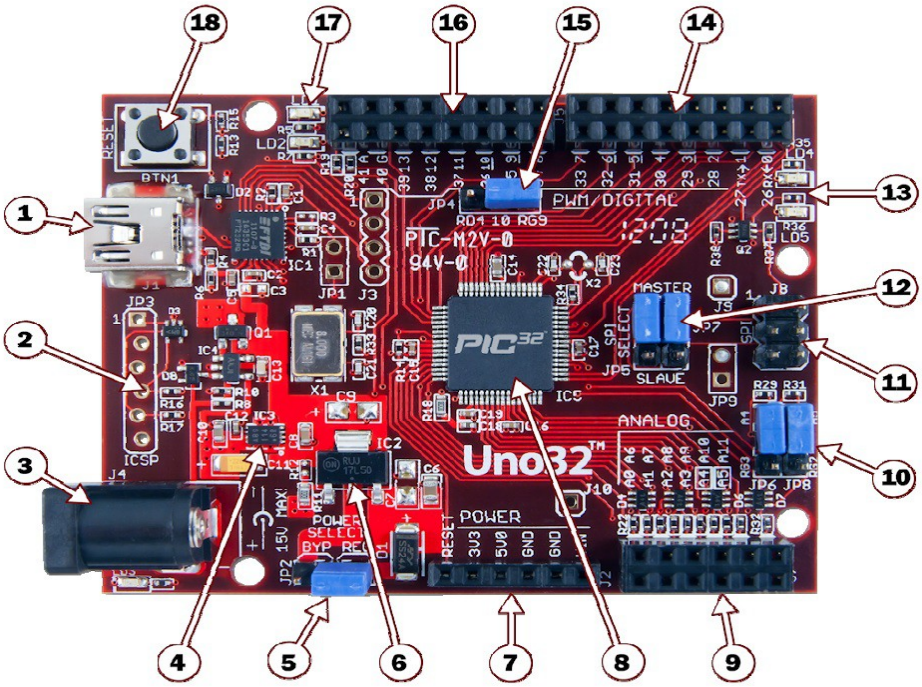


Figura 5.3.- Pines del CHIPKIT UNO32

Repasemos para qué sirven estos pines:

1. ***Conector USB para el conversor USB-Serie***. Se conecta a un puerto USB del PC para proporcionar el puerto de comunicaciones que el MPIDE utiliza para comunicarse con el Uno32. También se puede utilizar para alimentar la placa cuando está conectada a la PC.
2. ***JP3 – Conector para la herramienta de depuración de Microchip***. Este conector se utiliza para conectar herramientas programador / depurador de Microchip, como el PICkit 3. Esto permite que Uno32 pueda ser utilizado como un microcontrolador tradicional mediante el IDE de microchip MPLAB.
3. ***J4 - Conector de alimentación externa***. Se trata de un conector de 5,5 mm x 2,1 mm tipo barril utilizado para alimentar la placa Uno32 mediante una fuente de alimentación externa. Está conectado con el terminal central como la tensión de alimentación positiva. La tensión de alimentación debe estar en el rango de 7V a 15V.
4. ***Fuente de alimentación - Regulador 3.3V***. Regulador de tensión para la alimentación de 3.3V. Esta fuente de alimentación puede proporcionar hasta 500 mA de corriente.

5. ***JP2 – Jumper de selección de potencia***. Este puente se utiliza para enrutar potencia desde el conector de alimentación externa hacia el regulador de voltaje 5V de la placa o para evitar el regulador de 5V. La posición REG lleva tensión a través del regulador de 5V. La posición BYP no pasa por el regulador de 5V de la placa. Con el puente en la posición BYP la tensión de entrada máxima que se puede aplicar en el exterior conector de alimentación es de 6V.
6. ***Fuente de alimentación - Regulador 5V.*** Este regulador de voltaje de 5V regula la tensión de entrada aplicada en el conector exterior de alimentación de 5V. Se utiliza para alimentar el regulador de 3.3V y proporcionar 5V a las placas (shields) de expansión. Este regulador puede proporcionar hasta 800mA de corriente.
7. ***J2 - Conector de energía a placas de expansión***. Este conector proporciona energía a las placas de expansión de E / S conectados a la tarjeta.
8. ***Microcontrolador PIC32***. El microcontrolador PIC32MX320F128H es el procesador principal de la placa.
9. ***J7 - Conector de señales analógicas***. Este conector proporciona acceso a los pines analógicos / digitales de E / S en el microcontrolador.
10. ***JP6 / JP7 - A4 / A5 Jumpers de selección de señal***. Estos puentes se utilizan para cambiar los pines 9 y 11 en el conector J7 entre las entradas analógicas A4 y A5 o la señales I2C SDA y SCL.
11. ***J8 - Conector de señal SPI***. Este conector proporciona acceso alternativo a las señales SPI. Es utilizado por algunas placas de expansión para tener acceso al bus SPI.
12. ***JP5 / JP7 – Jumpers de selección Master / Slave SPI***. Estos puentes se utilizan para cambiar las señales SPI para el uso de la tarjeta Uno32 como dispositivo maestro o como un dispositivo esclavo SPI. Ambos puentes se deben cambiar juntos. Colocaremos los puentes en la posición de MASTER para el funcionamiento como maestro y en la posición SLAVE para el funcionamiento como esclavo. Normalmente, estos puentes están en la posición MASTER.
13. ***LEDs de usuario***. Dos LEDs conectados a los pines de señales digitales 13 y 43.
14. ***J6 - Conector de señales digitales***. Este conector proporciona acceso a los pines de E / S digitales del microcontrolador.
15. ***JP4 - Pin 10. Jumper de selección de señal***. Este puente se utiliza para cambiar el pin 5 del conector J5 (señal digital 10) entre PWM y modo SPI. El puente se coloca en la posición RD4 para la salida de PWM y en la posición RG9 para el funcionamiento como esclavo SPI. Estará normalmente en la posición RD4.

16. ***J5 - Conector de señal digital***. Este conector proporciona acceso a los pines de E / S digitales en el microcontrolador.
17. ***LEDs de estado de Comunicaciones***. Estos LED indican la actividad en la interfaz serie USB.
18. ***Botón de reinicio.*** Este botón se puede utilizar para reiniciar el microcontrolador, reiniciando la operación desde el bootloader.

Más información sobre el Uno32 se puede encontrar en el ANEXO V o en la documentación adjunta a este proyecto.

5.2.2.- CÁMARA Y LENTE

Como sensor estelar para nuestro Star Tracker vamos a seleccionar el OV7670 de OMNIVION. El chip OV7670 es un sensor de imagen de bajo voltaje que dispone de todas las funciones de una cámara VGA de un solo chip, y un procesador de imagen en un mismo circuito impreso. El OV7670 ofrece imágenes a fotograma completo, submuestreadas o en ventanas de 8 bits, en una amplia gama de formatos, controlado a través del interfaz SCCB, Serial Camera Control Bus, basado en I2C. Este producto es capaz de funcionar hasta 30 fotogramas por segundo (*fps*) en VGA con un completo control del usuario sobre la calidad de imagen, formato y de la transferencia de los datos de salida. Todas las funciones de procesamiento de imágenes necesarias, incluyendo el control de la exposición, gamma, balance de blancos, saturación de color, etc, también son programables a través de la interfaz de SCCB. La tabla 5.1 muestra las principales características del sensor OV7670.

Array Element (VGA)		640 x 480
Power Supply	**Digital Core**	1.8VDC ±10%
	Analog	2.45V to 3.0V
	I/O	1.7V to 3.0V
Power Requirements	**Active**	TBD
	Standby	< 20 µA
Temperature Range	**Operation**	-30°C to 70°C
	Stable Image	0°C to 50°C
Output Formats (8-bit)		• YUV/YCbCr 4:2:2 • RGB565/555 • GRB 4:2:2 • Raw RGB Data
Lens Size		1/6"
Chief Ray Angle		24°
Maximum Image Transfer Rate		30 fps for VGA
Sensitivity		1.1 V/Lux-sec
S/N Ratio		40 dB
Dynamic Range		TBD
Scan Mode		Progressive
Electronics Exposure		Up to 510:1 (for selected fps)
Pixel Size		3.6 µm x 3.6 µm
Dark Current		12 mV/s at 60°C
Well Capacity		17 K e
Image Area		2.36 mm x 1.76 mm
Package Dimensions		3785 µm x 4235 µm

Tabla 5.1.- características del sensor OV7670

Más información en ANEXO VI y documentación adjunta a este proyecto.

Para la configuración del tamaño de la imagen que deseamos tomar, así como el formato de salida de los datos, o características como ganancia, balance de blancos, filtros... usaremos el puerto SCCB del sensor, que consiste en un interface maestro esclavo, en la que el Uno32 es el maestro, unidos a través de dos hilos, y totalmente compatible con el bus I2C de nuestro Uno32. Para que la comunicación entre el OV7670 y el Uno32 a través del bus I2C llegue a buen termino, deberemos de colocar sendas resistencias de pull-up de 4,7 KΩ entre los terminales de datos y reloj del bus y la alimentación V_{DD}. Esta conexión queda reflejada en la figura 5.4.

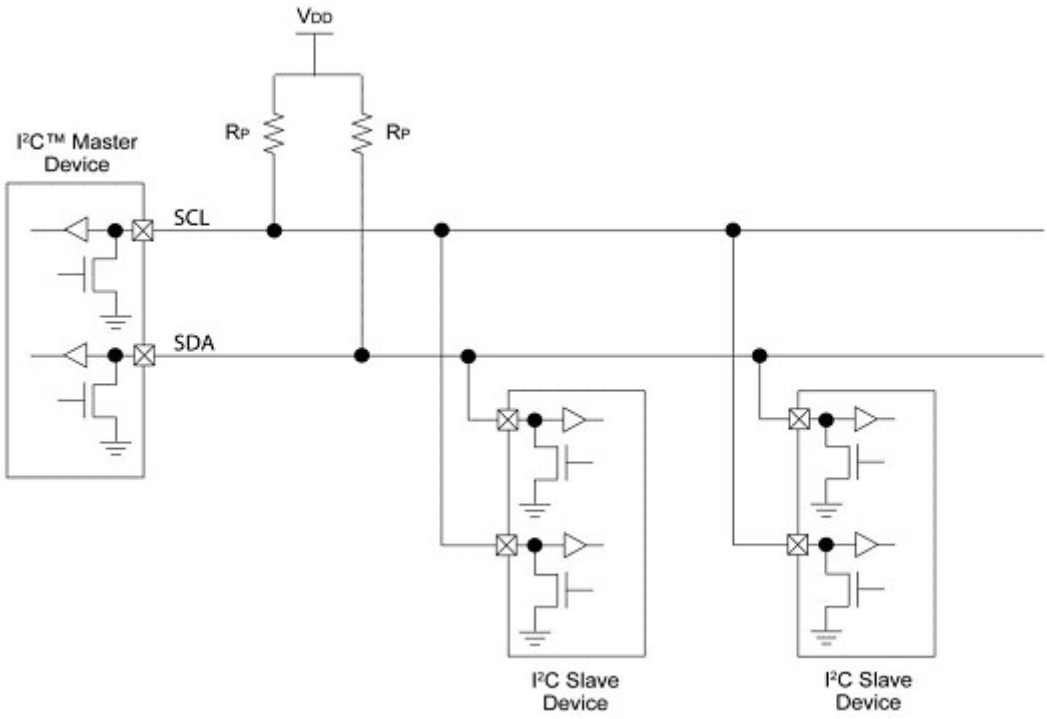


Figura 5.4.- Conexión maestro-esclavos en bus I2C. R_p = 4,7 KΩ

Una vez tenemos configurado el sensor OV7670 con el tamaño, formato y funcionalidades deseadas, veamos como capturar una imagen. El OV7670 dispone de los siguientes pines para el control y adquisición de una imagen.

VSYNC	Output	Vertical synchronization
HREF	Output	Horizontal synchronization
PCLK	Output	Pixel clock
XCLK	Input	System clock
D0-D7	Output	Video parallel output
RESET	Input	Reset (Active low)
PWDN	Input	Power down (Active high)

Tabla 5.2.- Pines de control de imagen del sensor OV7670

El OV7670 envía los datos en paralelo y en formato síncrono. En primer lugar, para obtener los datos de la OV7670, es necesario suministrar una señal de reloj en el pin XCLK. De acuerdo con la hoja de datos, este reloj debe tener una frecuencia de entre 10 y 48 MHz, con una frecuencia típica recomendada de 24 MHz. Después de que la señal de reloj ha sido aplicada al pin XCLK, el

OV7670 empezará a producir datos en los pines VSYNC, HREF y D0-D7. Vamos a echar un vistazo a estas señales.

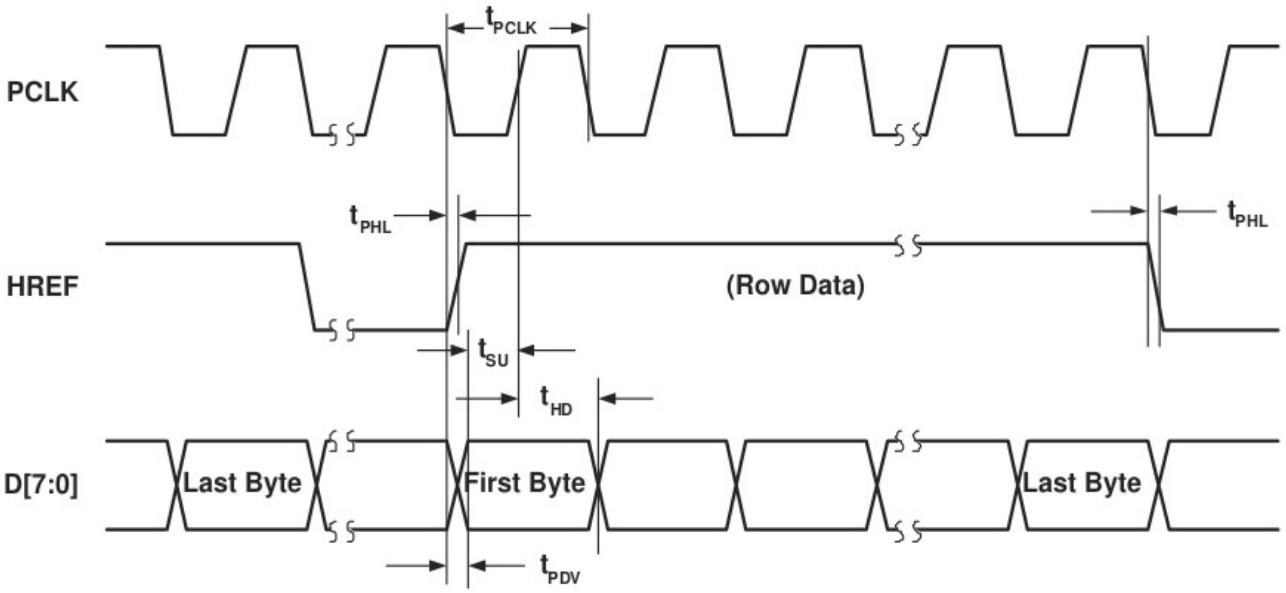


Figura 5.5.- Sincronización horizontal

Lo primero a destacar es que el byte D0-D7 debe ser muestreado en el flanco ascendente de la señal PCLK. Segundo, D0-D7 debe ser muestreado sólo cuando HREF está en estado alto. Además, el flanco ascendente de HREF señala el comienzo de una línea, y el flanco descendente de HREF señala el final de la línea. Todos estos bytes muestreados cuando HREF estaba en estado alto, corresponden a los pixels de una línea.

Hay que tener en cuenta que un byte no es un pixel, pues depende del formato elegido. Por defecto el formato es YCbCr42 (formato que elegiremos). YCbCr es un formato en el que un color RGB puede ser codificado. El componente de luminancia, o Y, es la cantidad de luz blanca de un color, y el Cb y Cr son los componentes de color, que respectivamente codifican los niveles de azul y rojo en relación con la componente de luminancia. El canal Y codifica la escala de niveles de de grises de la imagen. Por lo tanto, la forma más fácil de obtener una imagen monocroma de la OV7670 es extraer el canal Y del formato YcbCr.

Como el formato RGB, el formato YCbCr también almacena cada canal como 8 bits (de 0 a 255) y podemos convertir de YCbCr a RGB utilizando la siguiente expresión.

$$
\begin{aligned}
R &= Y + 1.402 \cdot (C_R - 128) \\
G &= Y - 0.34414 \cdot (C_B - 128) - 0.71414 \cdot (C_R - 128) \\
B &= Y + 1.772 \cdot (C_B - 128)
\end{aligned}
$$

El sensorOV7670 utiliza el formato YCbCr422, este formato se almacena como muestra la tabla 5.3:

	Byte 0	Byte 1	Byte 2	Byte 3
Word 0	Cb0	Y0	Cr0	Y1
Word 1	Cb2	Y2	Cr2	Y3
Word 2	Cb4	Y4	Cr4	Y5

Tabla 5.3.- Datos guardados como palabras de 4 bytes

O equivalentemente, los datos llegan en el siguiente orden:

N	Byte
1st	Cb0
2nd	Y0
3rd	Cr0
4th	Y1
5th	Cb2
6th	Y2
7th	Cr2
8th	Y3
...	...

Tabla 5.4.- Orden de llegada de datos en YCbCr422

Y los pixels actuales son de la siguiente manera:

Pixel 0	Y0 Cb0 Cr0
Pixel 1	Y1 Cb0 Cr0
Pixel 2	Y2 Cb2 Cr2
Pixel 3	Y3 Cb2 Cr2
Pixel 4	Y4 Cb4 Cr4
Pixel 5	Y5 Cb4 Cr4

Tabla 5.5.- Pixels en YCbCr422

Se puede observar que cada pixel es de 3 bytes de longitud (por ejemplo, Y0, Cb0 y Cr0), como en el formato RGB. Pero en el formato YCbCr422, los canales Cb y Cr se comparten entre dos pixels consecutivos (por ejemplo los pixels 0 y 1 comparten CB0 y Cr0). Por tanto, dos pixels se han "comprimido" en 4 bytes o 32 bits, esto significa que en promedio cada pixel se almacena como 2 bytes o 16 bits. En el ejemplo anterior, 3 palabras (12 bytes) almacenan 6 pixels. La ventaja adicional de YCbCr es que el canal Y es la imagen en escala de grises, mientras que en RGB tendríamos que promediar los 3 canales para obtener la imagen en escala de grises.

La Figura 5.6 muestra las señales de en modo VGA (640 x 480). Mientras HSYNC está en el estado alto, debemos capturar 640 pixels, lo que equivale a una línea. Las 480 líneas, lo que equivale a una imagen, son capturados durante el estado bajo de VSYNC. Esto significa que el flanco descendente de VSYNC señala el comienzo de una imagen, y su flanco ascendente indica el final de una imagen. Eso cubre todo el proceso de obtención de una imagen, la cuestión es cómo de rápido se envían las imágenes. Por defecto, el PCLK tendrá la misma frecuencia de XCLK, sin embargo se pueden configurar pre-escaladores y PPLs mediante el SCCB, para producir un PCLK de diferente frecuencia. Un PCLK de 24 MHz producirá 30 fps, un PCLK de 12 MHz producirá 15 fps y así sucesivamente. Todo esto es independiente del formato de la imagen: VGA, CIF, QCIF, etc.

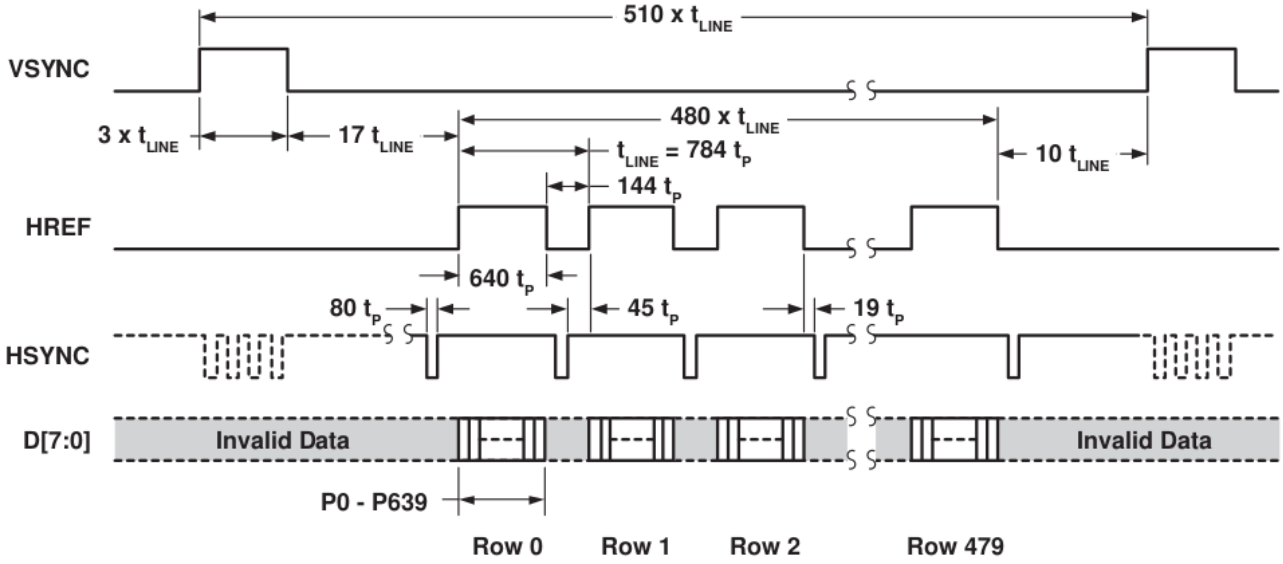


Figura 5.6.- Timing VGA

Como veremos más adelante, elegiremos una tamaño de imagen de QVGA, que corresponde a un tamaño de 320x240 pixels. La figura 576 muestra el timing para este tamaño de imagen.

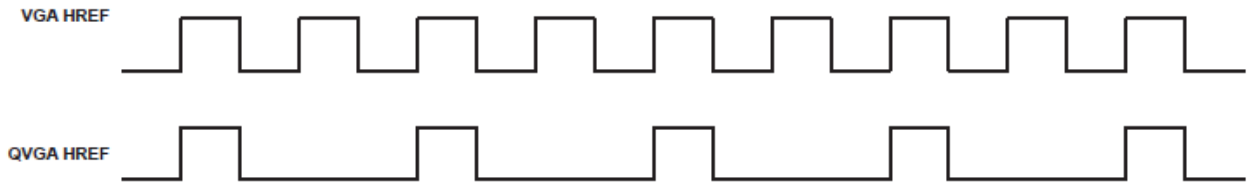


Figura 5.7.- Timing QVGA

Vemos como el periodo de la señal HREF se reduce a la mitad, aunque en cada región con valor alto de HREF se lean los datos a la misma frecuencia.

Encontramos dos versiones del chip OV7670 ya ensambladas en placas PCB con conectores de 20 pines, listos para ser usados utilizando cables de conexión, sin necesidad de soldar el chip ni ningún otro componente a una PCB. Veamos los dos casos:

1. ***OV7670 sin FIFO***: El sensor OV7670 exporta directamente todas las señales digitales de control de imagen y conecta el resto de sus pines a un conector de 18 pines en la PCB de la siguiente manera:

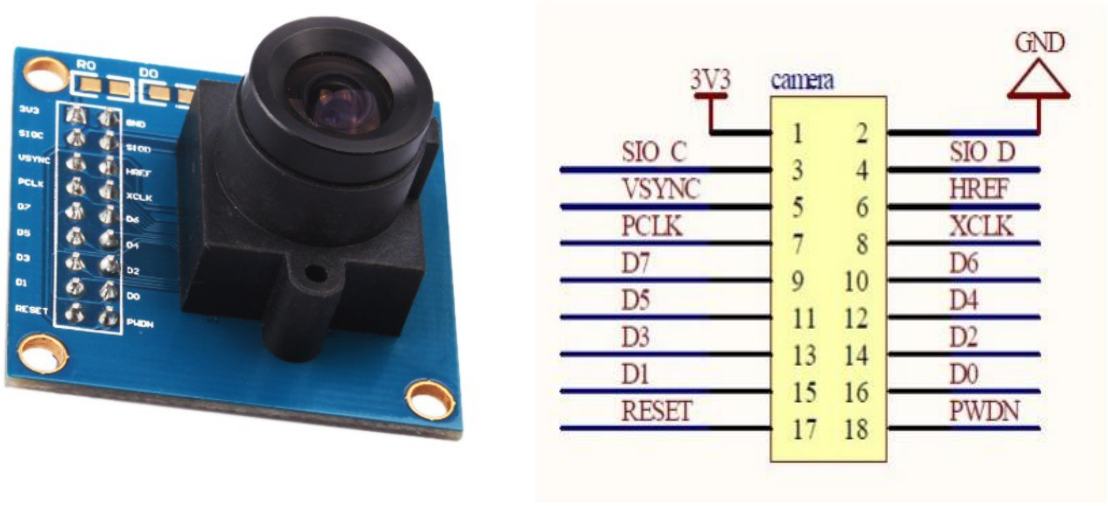


Figura 5.8.- OV7670 sin FIFO

Será el propio microcontrolador el encargado de controlar todas las señales de salida de la cámara para poder componer la imagen que la cámara da según el timing mostrado anteriormente.

En un primer momento, decidimos hacer pruebas con esta cámara. Para darle la señal de reloj utilizamos una salida en PWM del Uno32, modificando la librería *wiring_analog.c* que el entorno de programación MPIDE proporciona para generar señales PWM, de tal manera que podamos controlar el duty-cycle y la frecuencia de la señal PWM. Generamos una señal PWM con duty-cycle 50% y una frecuencia de 10 MHz, la mínima que soporta la cámara OV7670 según especificaciones.

Según la figura 5.5, deberemos de leer un byte de datos de la imagen en las señales D0-D7 a una frecuencia de 10 MHZ, y antes de leer el siguiente byte, deberemos de almacenar ese byte en RAM, y así sucesivamente hasta completar toda la imagen.

Posteriormente, durante la fase de programa que ejecuta el algoritmo de identificación de estrellas, accederemos a la imagen almacenada en RAM. El problema que pudimos observar, es que, como nuestro procesador ejecuta instrucciones a 80 MHz, y la lectura de bytes de datos se debe hacer a 10 MHz, y entre dos lecturas consecutivas debemos guardar ese byte en RAM a través del protocolo SPI, el procesador no era capaz de hacer todo esto, perdiendo la sincronización de los datos que componen la imagen. Nuestro procesador no es los suficientemente rápido como para llevar a cabo la tarea de lectura y guardado de bytes a la menor frecuencia de trabajo de la cámara (10 MHZ).

Para realizar estas pruebas y llegar a esta conclusión, probamos primero con las funciones que el entorno de desarrollo MPIDE proporciona para acceso a sus señales digitales. Se trata de las funciones *digitalRead* y *digitalWrite*. Estas funciones realizan varias operaciones más además de acceder a los puertos del PIC32, y por tanto, empeoraban la situación. Decidimos acceder directamente a los puertos del PIC32, con lo que mejoró la velocidad de acceso, pero no fue suficiente, así que tuvimos que probar con otra configuración, como veremos en el siguiente apartado. En la figura 5.9 vemos como configurar pin del Uno 32 como entrada y salida, y como acceder a ellos para su lectura y escritura respectivamente.

```
pinMode(9, OUTPUT);
pinMode(10, INPUT);
digitalWrite(9,HIGH);
if (digitalRead(10) == HIGH) ...
```

FIGURA 5.9.- Acceso a entradas y salidas digitales mediante funciones MPIDE

La figura 5.10 muestra cómo configurar los puertos E y D como entrada y salida y cómo acceder a ellos para su lectura y escritura, mediante el acceso directo a los puertos del PIC32 y sin hacer uso de las funciones de MPIDE, que como ya hemos comentado, realizan más operaciones y son mas ineficientes desde el punto de vista del procesamiento.

```
TRISE = 0xFF;  // Pins 26 - 33 as INPUT for D0-D7 Camera Data
TRISD = B11011001;
// READ PORTS
#define  RCLK   (int) (PORTD & B00000010)  //5  // RD1
// WRITE PORTS
#define RCLK_HIGH  PORTD |= (int) (1<<RCLK_bit);
#define RCLK_LOW   PORTD &= (int) (~(1<<RCLK_bit));

RCLK_HIGH;
result = (uint8_t) PORTE;
RCLK_LOW;
```

FIGURA 5.10.- Acceso a entradas y salidas digitales mediante acceso directo a puertos PIC32

2. ***OV7670v3 con FIFO:*** Este componente integra en la PCB el sensor OV7670, un oscilador a 24 MHZ y una memoria FIFO, de tal manera que mediante la electrónica del sistema, los bytes de la imagen adquirida por el sensor van directamente a la memoria FIFO, para posteriormente, y a la velocidad que deseemos y que nuestro procesador sea capaz de alcanzar, leer esos datos de la FIFO. El aspecto que presenta este componente lo podemos ver en la figura 5.11.

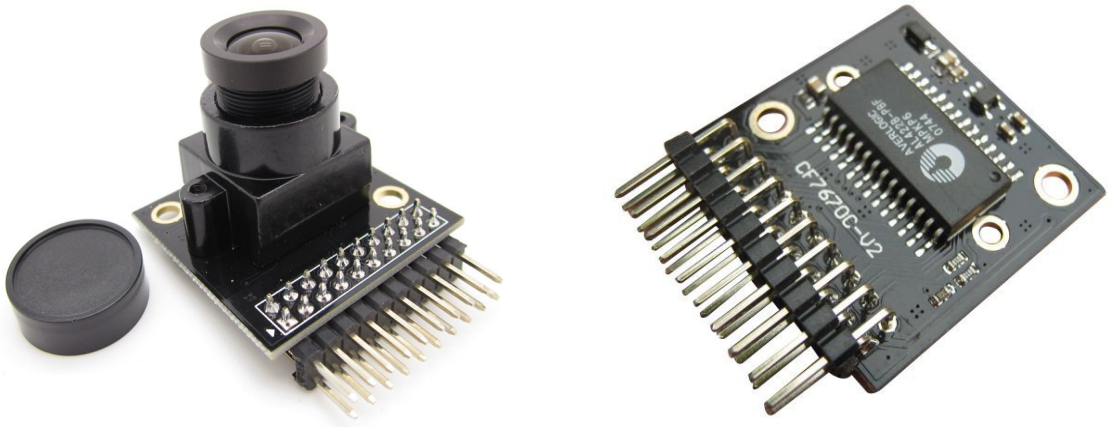


FIGURA 5.11.- OV7670v3 + FIFO

Podemos observar como en el reverso de la PCB, al otro lado del sensor y la lente, se encuentra la FIFO. Se trata de una memoria AL422B, consistente en un DRAM de 3M-bits y configurada como una FIFO (first in, first out) de 393.216 palabras de 8 bits. Sus principales características las podemos ver en la figura 5.12.

- 384K (393,216) x 8 bits FIFO organization
- Support VGA, CCIR, NTSC, PAL and HDTV resolutions
- Independent read/write operations (different I/O data rates acceptable)
- High speed asynchronous serial access
- Read/write cycle time: 20ns
- Access time: 15ns
- Output enable control (data skipping)
- Self refresh
- 5V or 3.3V power supply
- Standard 28-pin SOP package

FIGURA 5.12.- Características AL422B

Como podemos ver, la velocidad de escritura y lectura en la memoria son independientes, por lo que podremos escribir la imagen a la frecuencia que la cámara necesite, y posteriormente leer esa imagen de la memoria a la velocidad que nuestro procesador sea capaz de alcanzar. La descripción de los pines de la AL422B, que nos ayudará a comprender como funciona el sistema al completo, se muestra en la tabla 5.6.

Pin name	Pin #	I/O type	Function
DI0~DI7	1~4, 11~14	input	Data input
WCK	9	Input	Write clock
/WE	5	Input (active low)	Write enable
/WRST	8	Input (active low)	Write reset
DO0~DO7	15~18, 25~28	Output (tristate)	Data output
RCK	20	Input	Read clock
/RE	24	Input (active low)	Read enable
/RRST	21	Input (active low)	Read reset
/OE	22	Input (active low)	Output enable
TST	7	Input	Test pin (pulled-down)
VDD	10		5V or 3.3V
DEC/VDD	19		Decoupling cap input
GND	6, 23		Ground

Tabla 5.6.- Descripción pines AL422B

El esquema que presenta esta cámara con memoria incorporada lo podemos observar en la figura 5.13.

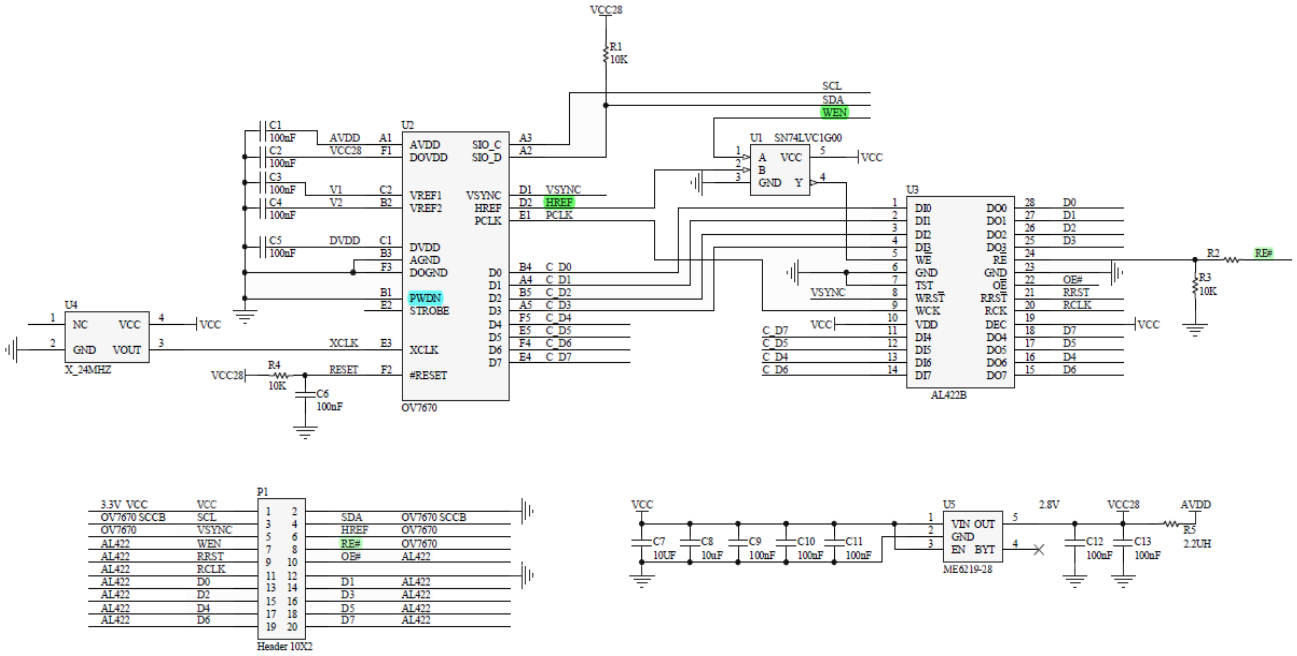


FIGURA 5.13.- Esquema OV7670v3 + FIFO

Vemos como la señal de reloj XCLK para el sensor la proporciona un oscilador de 24 MHz y como la señal de salida PCLK para lectura de los datos de la cámara va conectada a la señal WCK correspondiente al reloj de escritura de la memoria AL422B.

Para la escritura de datos en la FIFO, vemos como el reset del puntero de escritura se realiza con el flanco de bajada de la señal de sincronismo vertical de la cámara, debido a la conexión entre las señales VSYNC de la cámara y la señal $\overline{WRST}$ de la FIFO. Pero como vemos en la figura 5.5 el reset del puntero de escritura, que debiera de producirse al leer una nueva imagen, se realiza en el flanco positivo, por tanto, configuramos la cámara a través de la comunicación SCCB para invertir la señal de VSYNC. También vemos que mediante la salida el chip SN74LVC1G00, que realiza un NAND de las dos entradas que dispone, damos permiso de escritura a la FIFO a través de su pin $\overline{WE}$. Las dos entradas de la puerta NAND corresponden a la señal HREF de sincronización horizontal de la cámara y la señal WEN del conector, que irá a nuestro procesador, de tal manera que siempre que desde el PIC32 activemos WEN y la cámara esté transmitiendo una línea de la imagen, la FIFO leerá los datos de la cámara a través de la conexión de los pines D0-D7 de la cámara a los pines de entrada de la FIFO, a la velocidad marcada por el oscilador y por la señal PCLK que exporta la cámara y que recibe la FIFO como reloj de escritura. Posteriormente veremos como controlamos la adquisición de una imagen en la FIFO mediante el PIC32.

Para la lectura de los datos de la imagen almacenada en la FIFO deberemos primero de resetear el puntero de lectura de la FIFO a través de la señal $\overline{RRST}$ conectada al PIC32. Posteriormente, y a la velocidad que proporcione el programa que corre en nuestro PIC, iremos leyendo los datos de la FIFO activando y desactivando la señal RCLK que corresponde al reloj de lectura de la FIFO. En cada ciclo de RCLK, la FIFO proporcionará a través de los bits DO0-DO7 cada uno de los bits que forman la imagen, y que previamente hemos escrito. Sin realizar una nueva lectura de una imagen de la cámara, tan sólo reseteando de nuevo el puntero de lectura, podremos volver a leer la imagen almacenada en la FIFO.

El tamaño de la FIFO es de 393.216 bytes. Si configuramos la cámara para tomar imágenes YUV, para posteriormente tomar sólo el canal Y correspondiente al brillo, la cámara nos devolverá, como veremos más adelante, una media de 2 bytes por pixel. Una imagen del sensor en formato VGA presenta un tamaño de 640x480 = 307.200 pixels. Como la cámara proporciona 2 bytes por pixel, tendremos 614.400 bytes, que no caben en la FIFO. Deberemos pues de reducir el formato de nuestra imagen a QVGA que presenta un tamaño de 320x240 = 76.800 pixels, que nos da como resultado un tamaño en bytes para la FIFO de 153.600, tamaño que sí es factible de almacenar en la FIFO. Usaremos pues la comunicación SCCB para configurar nuestra cámara en modo YUV y formato QVGA, para poder adquirir correctamente la información de la cámara que deseamos. Posteriormente, de los bytes YUV con la imagen, tan sólo tomaremos para utilizar en nuestro algoritmo de reconocimiento de estrellas el canal Y, con información sobre el brillo de la imagen en formato de 8 bits, es decir, en escala de grises de valores compendios entre 0 y 255.

La figura 5.14 presenta el esquema de la conexión de la cámara OV7670 con FIFO al Uno 32, donde se pueden ver las conexiones para el bus I2C de configuración de la cámara, y las señales digitales de control de la imagen capturada y los datos de salida de la cámara para componer la imagen.

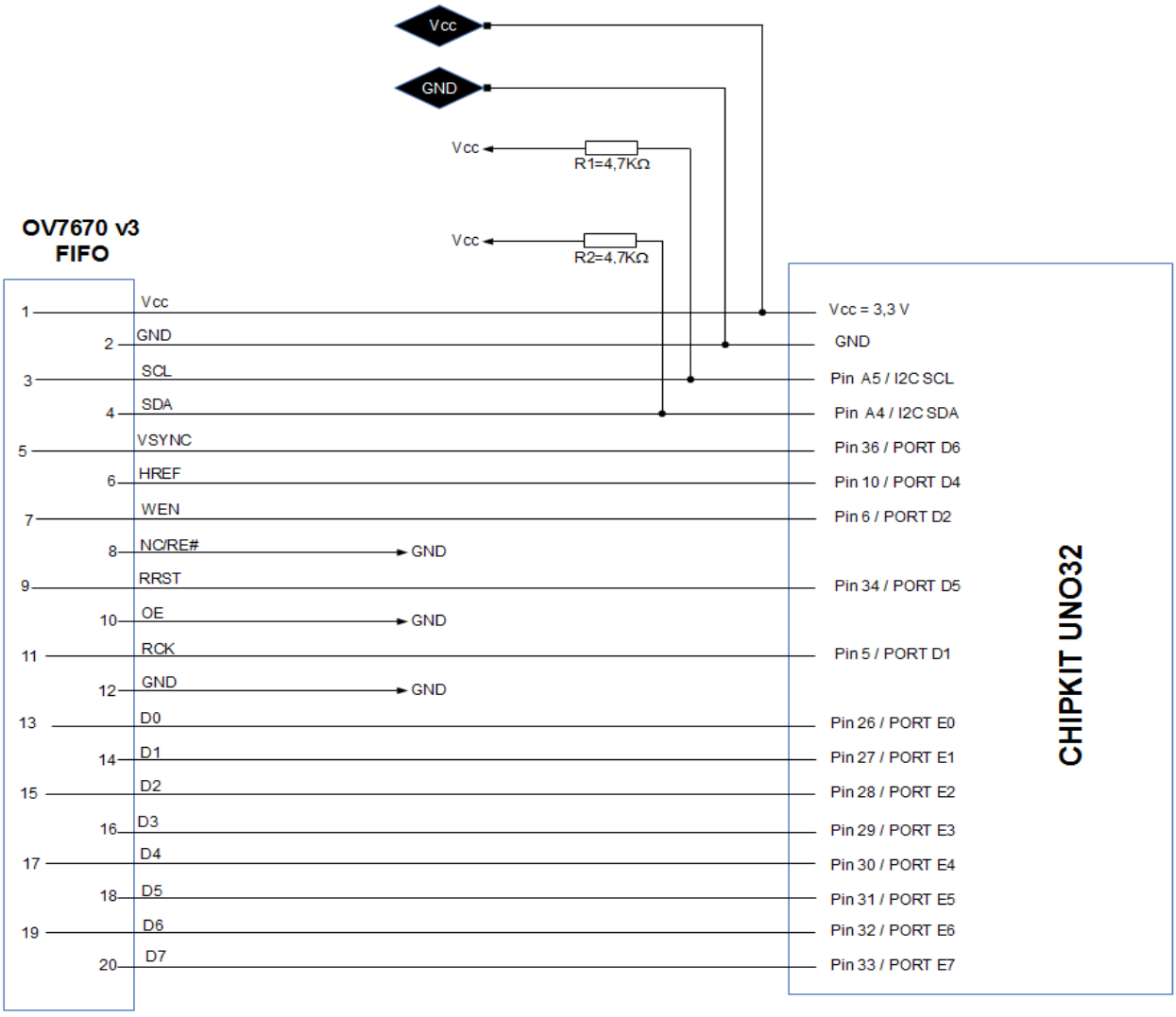


FIGURA 5.14.- Esquema conexión OV7670v3 + FIFO a Uno32

En cuanto a la lente a utilizar, disponemos del kit de lentes de Cognex para sensores de 1/6" con unas distancias focales de 3.6 mm, 5.8 mm, 8 mm, 16 mm y 25 mm. La figura 5.15 muestra el aspecto de las lentes del kit y la figura 5.16 su FOV en función de la distancia de trabajo.

FIGURA 5.15.- Juego de lentes Cognex para sensores de 1/6"

Para nuestra aplicación, elegiremos la lente de 3.6 mm, ya que presenta un mayor campo de visión FOV, lo que nos permitirá aumentar el número de estrellas visualizadas en una sola imagen. El FOV equivalente en grados de esta lente es de 45º, datos necesario para el funcionamiento de nuestro algoritmo.

FIGURA 5.16.- FOV vs distancia de trabajo para las lentes Cognex

5.2.3.- RAM

Tras obtener una imagen de la cámara y tenerla almacenada en la FIFO, vamos a proceder a leerla de la FIFO y pasarla a una memoria RAM. La pasaremos a una RAM para poder acceder a la información de los pixels de la imagen de la manera que el algoritmo de reconocimiento de estrellas necesite, y no de la manera secuencial que tendríamos si leyésemos directamente de la FFIO.

Vamos a seleccionar una memoria RAM que se comunicarán con el Uno32 a través del bus síncrono SPI. Se trata de la memoria 23LC1024 del fabricante Microchip, que tiene una capacidad de 1024 Mbits, es decir, 128 KBytes, o lo que es lo mismo, 128 Kpixels = 131.072 pixels. Como hemos visto anteriormente, en la memoria FIFO vamos a almacenar la imagen en formato YUV y QVGA. Nos quedaremos sólo con la información del canal Y, es decir, un byte por pixel, que para una imagen QVGA darán un total de 320x240 = 76.800 pixels, con lo que nuestra memoria RAM será más que suficiente para almacenar una imagen.

El esquema con los pines y su significado se muestra en la figura 5.17.

Name	Function
$\overline{CS}$	Chip Select Input
SO/SIO1	Serial Output/SDI/SQI Pin
SIO2	SQI Pin
Vss	Ground
SI/SIO0	Serial Input/SDI/SQI Pin
SCK	Serial Clock
$\overline{HOLD}$/SIO3	Hold/SQI Pin
Vcc	Power Supply

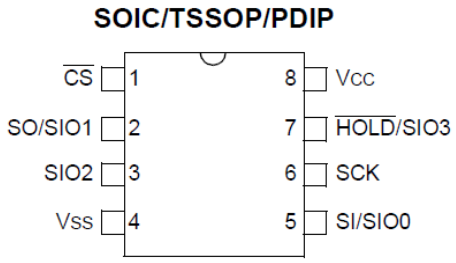


FIGURA 5.17.- Pines RAM 23LC1024

El esquema de conexión de la RAM a nuestro Uno32 se muestra en la figura 5.18

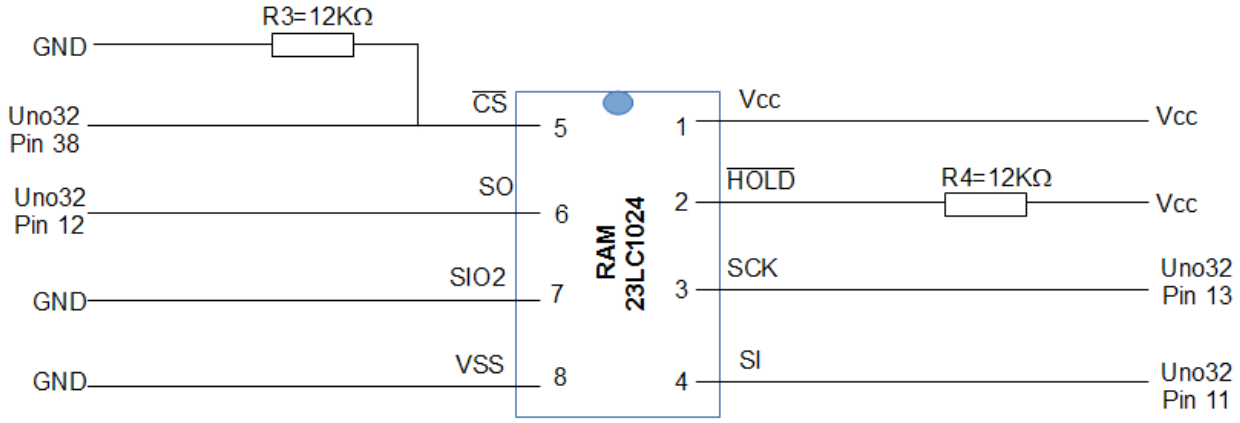


FIGURA 5.18.- Esquema conexión RAM a Uno32

En la figura anterior podemos ver como los tres pines que tiene para la comunicación SPI, el SI, el SO y el SCK, se conectan respectivamente a los pines MOSI, MISO y SCK del Uno32, que ejerce como maestro del bus. El pin $\overline{CS}$ conectado al pin 38 del Uno32 (entrada digital) a través de una resistencia de12 KΩ sirve para seleccionar y habilitar la lectura y escritura sobre esta RAM. De esta manera, en el bus SPI, podremos tener más dispositivos, incluso más memorias idénticas a ésta, y con este pin seleccionar sobre cual de los esclavos se desea leer o escribir. El esquema de comunicaciones en un bus SPI para más de un esclavo lo podemos observar en la figura 5.19.

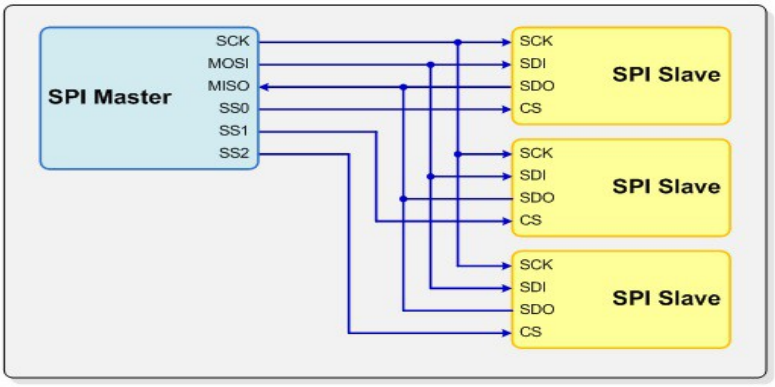
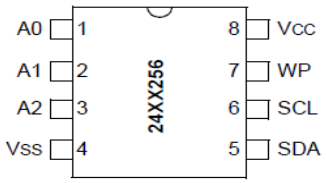


FIGURA 5.19.- Esquema bus SPI multi-esclavo

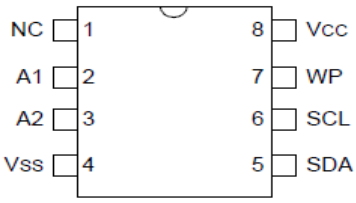
5.2.4.- EEPROM

Necesitaremos una memoria que mantenga sus datos a pesar de no disponer de tensión de alimentación, donde almacenar datos de configuración del sistema, y sobre todo el catálogo de estrellas para nuestro algoritmo. Para ello seleccionamos dos EEPROM de Microchip con diferente tamaño, y usaremos la que mejor se ajuste al tamaño de los datos que deseemos almacenar en ella. Las dos EEPROM seccionadas se tratan de las 24LC256 y 24FC1024 del fabricante Microchip. Ambas son memorias EEPROM con acceso de lectura y escritura a través del bus de comunicaciones asíncrono I2C. La diferencia está en que la primera tiene un tamaño de 256 Kbits, es decir 32 KBytes, y la segunda tiene un tamaño de 1Mbit, es decir, 128 KBytes. Las figuras 5.20 y 5.21 recogen los pines y su significado para las EEPROM 24LC256 y 24FC1024 respectivamente.



Name	PIN	Function
A0	1	User Configurable Chip Select
A1	2	User Configurable Chip Select
A2	3	User Configurable Chip Select
Vss	4	Ground
SDA	5	Serial Data
SCL	6	Serial Clock
WP	7	Write-Protect Input
Vcc	8	+2.5V to 5.5V

FIGURA 5.20.- Pines EEPROM 24LC256



Name	PIN	Function
NC	1	Not Connected
A1	2	User Configurable Chip Select
A2	3	User Configurable Chip Select
Vss	4	Ground
SDA	5	Serial Data
SCL	6	Serial Clock
WP	7	Write-Protect Input
Vcc	8	+1.8 to 5.5V (24FC1026)

FIGURA 5.21.- Pines EEPROM 24FC1024

En la figura 5.22 podemos observar el esquema de conexión de ambas EEPROM al Uno 32. Para ambas es el mismo esquema, pues en el primer caso los pines A0, A1 y A2, y en el segundo caso los pines NC, A1 y A2, se conectan a GND. Los pines SCL y SDA se conectan a los pines A5 y A4 del Uno32, pines dedicados a desarrollar las comunicaciones I2C, a través de las resistencias de pull-up que ya tenemos instaladas para las comunicaciones SCCB de la cámara. La selección de a que dispositivo esclavo se refiere el Uno32 se hace por medio de software, como veremos más adelante.

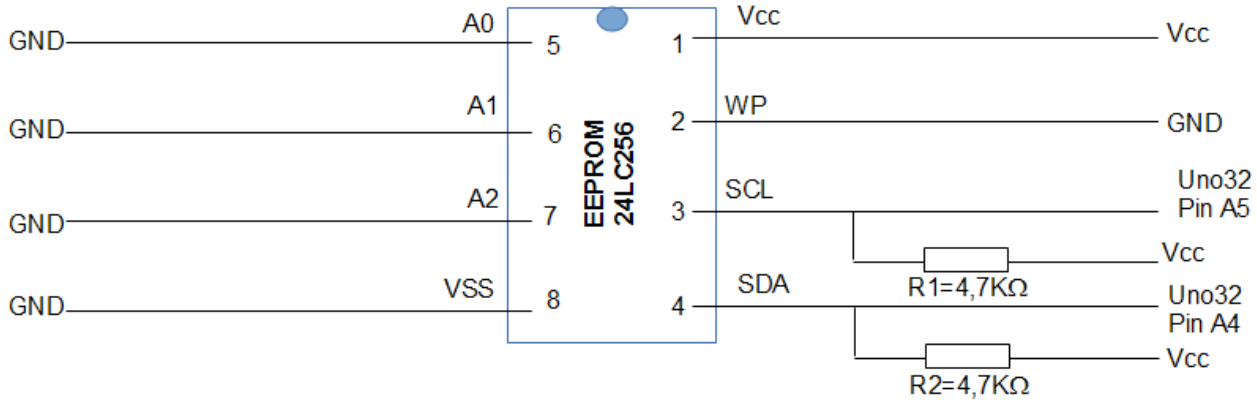


FIGURA 5.22.- Esquema conexión EEPROM a Uno32

5.3.- SOFTWARE

En el apartado anterior hemos definido y explicado el sistema hardware, cada uno de los componentes que van a formar el sistema, y como se conectan entre ellos. Vamos ahora a definir el funcionamiento del sistema a nivel de software. Empezaremos viendo como configurar el Uno32 para que pueda desarrollar todas las funcionalidades que deseamos y controlar los dispositivos que a

él hay conectados: cámara, RAM y EEPROM, y continuaremos viendo como implementar el algoritmo de identificación de estrellas de nuestro *Star Tracker*, que va a utilizar todo lo anterior para su funcionamiento. También veremos como el Uno32 se comunica con el sistema de control, en este caso un PC con una aplicación de control, comunicado con el Uno32 a través del puerto USB/serie.

Desarrollaremos el código firmware que llevará nuestro procesador utilizando el IDE de desarrollo basado en Arduino conocido como MPIDE. Este software dispone de gran variedad de bibliotecas que nos van a facilitar nuestro trabajo. El IDE presenta el aspecto que podemos ver en la figura 5.23.

FIGURA 5.23.- Aspecto del MPIDE

El MPIDE utiliza un puerto de comunicaciones serie para comunicarse con el loader de arranque que se ejecuta en el Uno32. El puerto serie de la placa Uno32 se implementa mediante un conversor serie-USB FT232R FTDI. El Uno32 utiliza un conector estándar mini-USB para la conexión al puerto USB del PC. Cuando el MPIDE necesita comunicarse con el Uno32, la tarjeta se resetea y comienza su sistema de arranque. El MPIDE entonces establece comunicaciones con el cargador o *bootloader* y descarga el programa a la tarjeta. Por tanto no es necesario el uso de un programador/debugger externo, como pueda ser el PICkit 3.

El programa desarrollado para el Uno32 a nivel de flujo consta de una función *setup* que se ejecuta en el arranque del PIC32 y donde configuraremos todos los dispositivos hardware del sistema, y una función *loop* que realiza la tarea principal del procesador de manera cíclica. Esta función se encarga de leer las ordenes que le llegan del PC de control por el puerto serie, y en función de las órdenes recibidas realiza una cosa u otra.

El procesador se puede encontrar en modo MANUAL o modo STAR_TRACKER. En modo STAR_TRACKER realizará el proceso de identificación de estrellas, es decir, captura una foto en la FIFO de la cámara, la pasa a la RAM vía SPI, y a partir de aquí ejecuta el algoritmo de centroiding para localizar las estrellas en la imagen tomada, y a través del algoritmo de Voting identifica las estrellas comparando con el catálogo que se encuentra almacenado en la EEPROM, al cual se accede por el bus I2C. En cada ciclo del algoritmo del Star Tracker, comprueba si recibe la orden del PC de pasar a modo MANUAL.

En modo MANUAL, el procesador lee el puerto serie esperando uno de los siguientes comando que le puede enviar el PC:

```
READ_CAMERA_CONFIG,
CAPTURE_IMAGE_FIFO,
READ_IMAGE_FIFO,
STORE_CATALOG_HEADER,
READ_CATALOG_HEADER,
STORE_STAR_PAIR,
READ_STAR_PAIR,
STORE_STAR_ID,
READ_STAR_ID,
STORE_UVECTOR,
READ_UVECTOR,
MANUAL_MODE,
STAR_TRACKER_MODE
```

El PC puede dar orden al procesador que capture una nueva imagen en la FIFO de la cámara y la envíe a la RAM, para luego leerla de la RAM y enviarla al PC por el puerto serie. También vemos toda una serie de funciones para almacenar en la EEPROM el catálogo de estrellas, y para su posterior lectura y verificación. Así mismo, se verifica si hay orden del PC de pasar a modo STAR_TRACKER.

A nivel de jerarquía de clases, el programa se encuentra encapsulado en varias clases que darán acceso a las funcionalidades hardware del sistema. A continuación pasaremos a revisar estas clases y como es controlado el hardware por parte del procesador.

5.3.1.- CONTROL DE LA CÁMARA

Como hemos comentado con anterioridad, todas las funciones para la configuración de la cámara se encuentran dentro de la clase OV7670. Las funciones de control de lectura de imágenes se encuentran en el archivo principal del proyecto. Además de la configuración en QVGA y YUV que hemos seleccionado para trabajar con nuestro *Star Tracker*, vamos a crear las funciones necesarias para configurar otros formatos, para poder realizar otras pruebas u otras posibles aplicaciones.

La clase OV7670 está implementada en tres archivos: *ov7670.cpp*, *ov7670.h* y *ov7670reg.h*. El último archivo contiene las definiciones de los tamaños de imagen y las definiciones de los registros y sus valores para cada una de las posibles configuraciones de la cámara. El fichero ov7670 presenta los prototipos de las funciones de la clase y sus miembros, y el fichero *ov7670.cpp* implementa estas funciones. La siguiente figura muestra el aspecto del fichero *ov7670.h*.

```
#include "ov7670reg.h"
#define OV7670_REGMAX (201)
#define OV7670_ADDDR  (int) 0x21
class ov7670
{
public:
   ov7670(TwoWire& s);
   void init();
   void PrintRegister(void);
   void setSerial(HardwareSerial *s);
   void Reset(void);
   void InitForFIFOWriteReset(void);
   void InitSetColorbar(void);
   void InitDefaultReg(void);
   void InitRGB444(void);
   void InitRGB555(void);
   void InitRGB565(void);
   void InitYUV(void);
   void InitBayerRGB(void);
   void InitVGA(void);
   void InitFIFO_2bytes_color_nealy_limit_size(void);
   void InitVGA_3_4(void);
   void InitQVGA(void);
   void InitQQVGA(void);
 private:
   uint8_t initialized;
   HardwareSerial *serial;
   TwoWire& I2C;
   void sccb_read(uint8_t address, uint8_t *data);
   void sccb_write(uint8_t address, uint8_t data);
};
```

FIGURA 5.24.- Fichero ov7670.h

Las funciones de esta clase tienen como misión principal configurar la cámara OV7670 de la manera que queramos, utilizando el protocolo SCCB. Este protocolo utiliza el bus I2C de nuestro Uno32 para leer y escribir en los registros de la cámara. Mediante la escritura de los adecuados valores en los registros de la cámara, configuraremos la cámara de la manera que deseemos. En la documentación adjunta podemos ver toda la serie de registros del sensor OV7670 así como los valores que pueden tomar y los efectos que estos tiene sobre la configuración de la cámara.

Las funciones que controlan la lectura y escritura básica del protocolo SCCB son *sccb_read* y *sccb_write*. Ambas utilizan la librería de MPIDE *Wire.h*, que controla las comunicaciones I2C de nuestro Uno32. Ambas funciones necesitan conocer la dirección del esclavo con el cual se quieren comunicar. Según las especificaciones del fabricante de la cámara, la dirección de lectura es 0x42 y la de escritura es 0x43. Sin embargo, las funciones de lectura y escritura de la librería *Wire.h* sólo presentan una dirección. Veamos qué significa esto y cómo utilizar esta información para calcular la dirección a colocar en las funciones de la librería *Wire.h*. En la comunicación I2C, tanto para leer como para escribir, se envía un primer byte de control en el que se codifica la dirección del esclavo y si se solicita una lectura o una escritura, seguida de la dirección de los datos a leer o escribir. Este byte de control presenta la siguiente estructura:

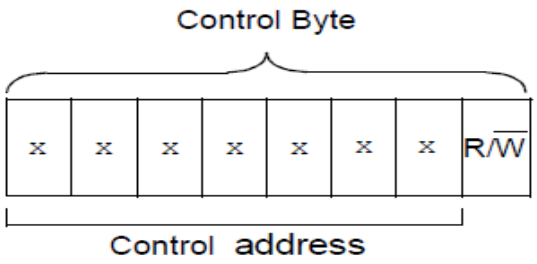


FIGURA 5.25.- Byte de control I2C

Vemos como los 7 primeros bits forman la dirección I2C del dispositivo esclavo y el último bit indica si se desea escribir o leer. Un 1 en este bit significa una lectura y un 0 una escritura. El fabricante de lOV760 dice que la dirección de escritura es 0x42 y la de lectura se 0x43. En binario estas direcciones son 1000010 y 1000011. Vemos como el último bit de ambas direcciones coincide con el hecho de que se trate de escritura y lectura. Como raíz, ambas direcciones poseen 100001, que en hexadecimal será 0x21, que corresponderá con la dirección que la librería *Wire.h* espera recibir. Este valor queda recogido en #define OV7670_ADDDR (int) 0x21.

En las especificaciones de la cámara, observamos que la frecuencia para el reloj en las comunicaciones puede alcanzar un valor a un máximo de 400 Khz. Como la EEPROM también va a compartir bus, miramos también sus características para comprobar que también soportan esa frecuencia. Para aumentar la velocidad en las comunicaciones, decidimos modificar el archivo *twi.h*, de las librerías core de MPIDE para el Uno32, y cambiamos el valor del define TWI_FREQ de 100.000 a 400.000, es decir, de 100KHz a 400 Khz.

Crearemos un objeto *Camera*, a partir de la clase ov7670, que utilizaremos a lo largo del programa para controlar la configuración de la cámara. En el constructor de la clase pasaremos como parámetro el objeto I2C, derivado de la librería *Wire.h* de MPIDE que controla el interfaz I2C. En la función *setup* del Uno32, iniciaremos el objeto Camera y le pasaremos una referencia al puerto serie del Uno32, para que el objeto Camera pueda enviar al PC información de depuración. También configuraremos la cámara para que nos devuelva la imagen en formato YUV de escala de grises y tamaño QVGA (320x240). La figura 5.26 muestra un resumen del código que realiza toda la configuración de la cámara.

```
TwoWire I2C;
ov7670  Camera(I2C);
Camera.init();
Camera.setSerial(&Serial);
Set_Camera_Config(YUV_GreyScale, QVGA);
```

FIGURA 5.26.- Código para iniciar y configurar cámara OV7670

La función *Set_Camera_Config* es la encargada de configurar la cámara a través del protocolo SCCB para que ésta nos proporcione una imagen en formato YUV y con una resolución QVGA. Además se encargará de resetear los registros de la cámara y de invertir la señal de sincronización vertical para realizar reset del puntero de escritura de la FIFO en el flanco positivo, como vimos en el apartado anterior. Un resumen del código que ejecuta esta función se puede ver en la figura 5.27.

```
Camera.Reset();
Camera.InitYUV();
Camera.InitQVGA();
Camera.InitForFIFOWriteReset();
Camera.InitDefaultReg();
```

FIGURA 5.27.- Resumen de código función Set_Camera_Config

Todas estas funciones establecen los bits de los registros de la cámara de tal manera que ésta se ponga en el modo que deseamos. Uno de los registros más importantes que modifican estas funciones se trata del registro COM7. La siguiente figura muestra la dirección y el significado de sus bits según la documentación del fabricante.

Address (Hex)	Register Name	Default (Hex)	R/W	Description
12	COM7	00	RW	Common Control 7 Bit[7]: SCCB Register Reset 0: No change 1: Resets all registers to default values Bit[6]: Reserved Bit[5]: Output format - CIF selection Bit[4]: Output format - QVGA selection Bit[3]: Output format - QCIF selection Bit[2]: Output format - RGB selection (see below) Bit[1]: Color bar 0: Disable 1: Enable Bit[0]: Output format - Raw RGB (see below) COM7[2] COM7[0] YUV 0 0 RGB 1 0 Bayer RAW 0 1 Processed Bayer RAW 1 1

FIGURA 5.28- Registro COM7 de la cámara OV7670

La función *Camera_Reset* establece a 1 el Bit[7] de este registro, que establece todos los registros de la cámara a sus valores por defecto. La función *Camera_YUV* establece a 0 los Bit[2] y Bit [0], y la función *Camera_InitQVGA* establece a 1 el Bit[4], además de otros bits en otros registros para ajustar otras propiedades secundarias.

Otro registro importante es el COM10, cuya dirección y significado de sus bits se muestra en la figura 5.29.

Address (Hex)	Register Name	Default (Hex)	R/W	Description
15	COM10	00	RW	Common Control 10 Bit[7]: Reserved Bit[6]: HREF changes to HSYNC Bit[5]: PCLK output option 0: Free running PCLK 1: PCLK does not toggle during horizontal blank Bit[4]: PCLK reverse Bit[3]: HREF reverse Bit[2]: VSYNC option 0: VSYNC changes on falling edge of PCLK 1: VSYNC changes on rising edge of PCLK Bit[1]: VSYNC negative Bit[0]: HSYNC negative

FIGURA 5.29- Registro COM10 de la cámara OV7670

La función *Camera_InitForFIFOWriteReset* pone a 1 el Bit[1] que invierte la señal VSYN de sincronización vertical de la cámara para, como hemos visto anteriormente, permitir el reset del puntero de escritura de la la memoria FIFO.

La función *Camera_InitDefaultReg* establece el valor de otros registros para la correcta configuración de la imagen que tome la cámara, como por ejemplo los factores gamma de corrección de color. Para una descripción mas detallada de todos los registros de la cámara, véase la documentación adjunta.

Una vez tenemos configurada la cámara, estaremos dispuestos para capturar una imagen. En un primer paso, capturaremos la imagen y la guardaremos en la FIFO de la cámara. Una vez tengamos la imagen en la FIFO, podemos enviarla vía serie al PC para mostrarla en el software de control del sistema, o guardarla en la memoria RAM. En ambos casos, será necesaria la lectura de los datos almacenados en la FIFO. Para efectos de depuración, crearemos funciones que envíen al PC por el puerto serie la imagen contenida en la RAM.

Para capturar una imagen y guardarla en la FIFO, utilizaremos la función *CaptureImage_to_FIFO*. Esta función activa el bit WEN de la cámara, que habilita la escritura de datos en la FIFO por hardware siempre que HREF indique que se están recibiendo datos de una línea, mientras la señal de sincronización vertical VSYNC permanece a 1 durante la transmisión de una imagen, tal y como hemos visto anteriormente. El reset del puntero de escritura se hace al inicio de la transmisión de la imagen al activarse automáticamente por hardware, como vimos en el apartado anterior, la señal $\overline{WRST}$. El código de esta función lo podemos ver en la figura 5.30.

```
void CaptureImage_to_FIFO(void)
{
  while(VSYNC!=0) {}; //While VSYNC_HIGH
  while(VSYNC==0){}; //While VSYNC_LOW

  WEN_HIGH;
  while(VSYNC!=0)
  {
    while(HREF==0 && VSYNC!=0){};
    while(HREF!=0 && VSYNC!=0){};
  }
  while(VSYNC==0){};
  WEN_LOW;
}
```

FIGURA 5.30- Código función CaptureImage_to_FIFO

La figura 5.31 muestra los timings de las señales de la FIFO implicadas en la escritura de los datos de la imagen y que respaldan el procedimiento de lectura de imágenes desarrollado

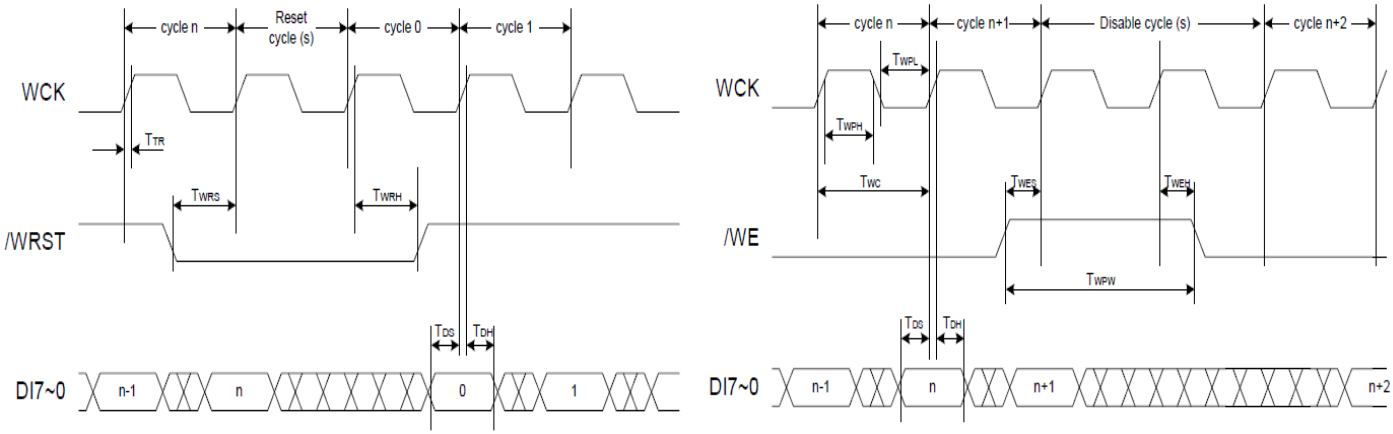


FIGURA 5.31- Timing de señales para escritura en FIFO

Tras ejecutar la función *CaptureImage_to_FIFO* tendremos una imagen en formato YUV en la FIFO de la cámara. Ahora vamos a leer esa imagen y enviarla vía serie al PC o guardarla en la RAM. En la ejecución del programa en modo AUTO, la imagen la guardaremos en la RAM para poder ser utilizada posteriormente en el algoritmo de identificación de estrellas. Disponemos de las siguientes funciones para leer la imagen de la FIFO:

- GetImage_FIFO_to_Serial_byte
- GetImage_FIFO_to_Serial_Line
- GetImage_FIFO_to_RAM

Estas tres funciones tienen el denominador común de leer la imagen de la FIFO. Las dos primeras la envían al PC a través del puerto serie, ya sea byte a byte leído, o línea a línea. La tercera función almacena la imagen leída en la FIFO en la RAM, escribiendo línea a línea. La siguiente figura muestra la función *GetImage_FIFO_to_RAM*. Las otras dos comparten la forma de leer la imagen de la FIFO, por lo que nos las representaremos aquí.

```
void GetImage_FIFO_to_RAM(int ImageNumber)
{
  int ii =0;
  int jj =0;
  uint8_t Line[Image_Columns];
  uint8_t val;
   ReadStart();
  // for IMAGE_COLUMNS*IMAGE_FILES*2 times, pulse the RCLK and read the digital pins
  // for image data
  for (jj=0; jj<Image_Files; jj++)
  {
            for (ii=0; ii<Image_Columns; ii++)
            {
                     val = ReadOneByte();
                    val = ReadOneByte();
                     Line[ii] = val;
            }
   SpiRam.write_camera_line(ImageNumber,(jj+1),Line);
   }
   ReadStop();}
```

FIGURA 5.32- Código función GetImage_FIFO_to_RAM

Podemos ver como, tras llamar a la función *ReadStar*, mostrada en al figura 5.35, y que comentaremos a continuación, realizamos una lectura de todos los pixels de la imagen mediante el barrido del número de columnas y filas en sendos bucles *for*. Para cada pixel, como el formato elegido es el YUV, y como hemos visto en el capítulo anterior y mostrado en la tabla 5.4, nos quedaremos sólo con la información de brillo, es decir los bytes Y. Es por eso que leemos los dos bytes con la función *ReadOneByte* pero tan sólo almacenamos en el array con los bytes de la línea el segundo de ellos. La lectura de un byte se muestra en la figura 5.33.

```
// Data Read
uint8_t ReadOneByte(void)
{
    uint8_t result;
    RCLK_HIGH;
    result = (uint8_t) PORTE;
   RCLK_LOW;
   return result;
}
```

FIGURA 5.33.- Código función ReadOneByte

Vemos como para leer un byte de salida de la FIFO, conectado al puerto E del Uno32, activamos la señal de reloj de lectura de la FIFO RCLK, leemos el puerto de datos E, y desactivamos RCLK.

Antes de la lectura de todos los bytes de la imagen, deberemos de resetear el puntero de lectura de la FIFO mediante la señal RRST. La figura 5.34 muestra el timing para el reseteo de este puntero y las señales involucradas en él.

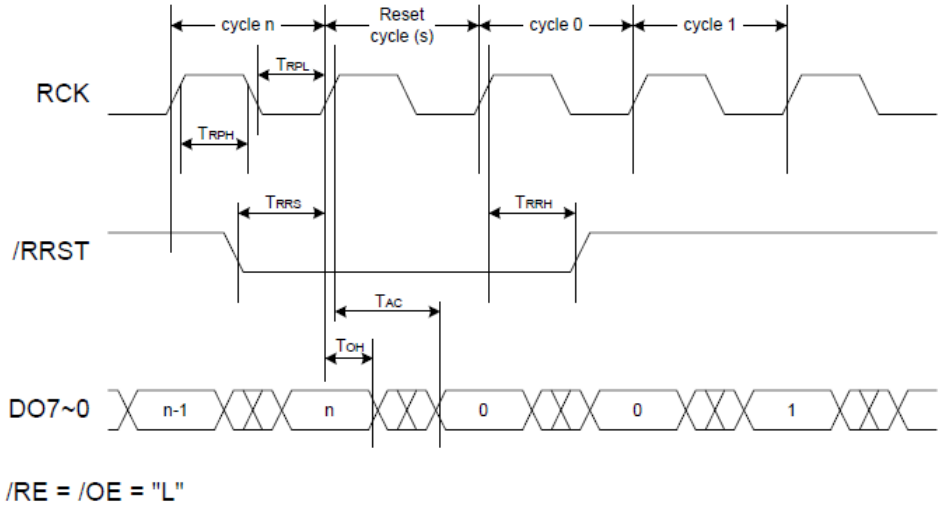


FIGURA 5.34.- Timing de señales para lectura en FIFO

Vemos como deberemos de mantener la señal $\overline{RRST}$ de reset del puntero de lectura activa durante al menos dos ciclos de la señal de reloj de lectura RCLK. La siguiente figura muestra la implementación de este procedimiento.

```
// Data Start
void ReadStart(void)
{
   // Cycle n
   RCLK_HIGH;
   RRST_LOW;
   RCLK_LOW;

   // Reset cycle
   RCLK_HIGH;
   RCLK_LOW;
   RCLK_HIGH;

   // Cycle 0
   RCLK_LOW;
   RRST_HIGH;
}
```

FIGURA 5.35.- Código función ReadStart

Una vez hayamos leído una línea de la imagen, almacenada en el array Line[], la enviaremos vía protocolo SPI al la memoria RAM mediante la línea de código:

```
SpiRam.write_camera_line(ImageNumber,(jj+1),Line);
```

que comentaremos en el siguiente apartado.

Finalmente, llamamos a la función ReadStop, que lee el último byte de la imagen, y que no almacenamos pues no corresponde al canal Y, y colocamos el reloj RCLK a 1, dejando las señales preparadas para una nueva adquisición de imagen. La siguiente figura muestra el aspecto de esta función.

```
// Data Stop
void ReadStop(void)
{
   ReadOneByte();
   RCLK_HIGH;
}
```

FIGURA 5.36.- Código función ReadStart

Al finalizar la llamada a esta función, o cualquiera de las otra dos, habremos leído la imagen de la memoria FIFO de la cámara, y la habremos almacenado en la RAM, o enviado vía serie al PC en los otros dos casos.

5.3.2.- CONTROL DE LA RAM

Anteriormente hemos visto como vamos a necesitar de una memoria RAM conectada al Uno32 a través del bus SPI para poder almacenar la imagen captada por la cámara y posteriormente utilizarla en nuestro algoritmo de reconocimiento de estrellas. Como también hemos visto anteriormente, en el bus SPI, el Uno32 realizará las funciones de maestro y la RAM de esclavo. Para que el Uno 32 se comporte como maestro del bus SPI, deberemos de colocar los jumpers JP5 y JP7 en posición de maestro. Como hemos visto en la figura 5.18, conectaremos entre el Uno32 y la RAM 4 pines. Los dos primeros con los canales de entrada y salida, y otro con el reloj del bus. Estas tres señales irán conectadas a los pines 11, 12 y 13 correspondientes a los pines que el Uno32 tiene reservados para el bus SPI. El cuarto pin corresponde a una salida digital del Uno32, concretamente la 38, que dará la señal de habilitación de la RAM y que nos permitiría la conexión de más elementos en el bus, cada uno con su señal de control controlada por el microprocesador.

La figura 5.37 muestra la configuración de estos pines en el programa:

```
//  SPI-RAM SETTINGS
#define DATAOUT         11  //MOSI
#define DATAIN          12  //MISO
#define SPICLOCK        13  //sck
#define SLAVESELECT     38  //ss
```

FIGURA 5.37- Configuración SPI maestro en Uno32

El control de la memoria RAM 23LC1024 lo vamos a encapsular dentro de la clase SpiRam, implementada en los archivos *SpiRam.h* y *SpiRam.cp*p, con los encabezados y la implementación de las funciones de la clase. La figura 5.38 muestra el aspecto del archivo con los prototipos de la clase SpiRam.

```
#include <SPI.h>
// SRAM opcodes
#define WREN  6
#define WRDI  4
#define RDSR  5
#define WRSR  1
#define READ  3
#define WRITE 2
// SRAM Hold line override
#define HOLD 1

// SRAM modes
#define BYTE_MODE   (0x00 | HOLD)
#define PAGE_MODE   (0x80 | HOLD)
#define STREAM_MODE (0x40 | HOLD)

// Clock Divider for 20 MHz
#define CLOCK_DIVIDER_20_MHZ 1

#define IMAGE_RAM_RESERVED 77000   // MAX  VGA=307200 pixels
                                   // MAX QVGA= 76800 pixels
class SpiRAM
{
public:
   SpiRAM(byte ssPin);
   uint8_t read_byte(uint32_t address);
   uint8_t write_byte(uint32_t address, uint8_t data_byte);
   void read_page(uint32_t address, uint8_t *buffer);
   void write_page(uint32_t address, uint8_t *buffer);
   void read_stream(uint32_t address, uint8_t *buffer, int length);
   void write_stream(uint32_t address, uint8_t *buffer, int length);

   //Camera
   void write_camera_line(int Image, int Line, uint8_t *buffer);
   void read_camera_line(int Image, int Line, uint8_t *buffer);
   void write_camera_pixel(int Image, int Line, int Column, uint8_t Pixel_value);
   uint8_t read_camera_pixel(int Image, int Line, int Column);
   void Set_Image_Columns(int ImgColumns);
```

```
  private:
    int Image_Columns;
    char _current_mode;
    byte _ssPin;
    void enable();
    void disable();
    void _set_mode(char mode);
};
```

FIGURA 5.38- Fichero SpiRam.h

Analicemos el contenido de este fichero. El fichero hace una llamada a la librería ***SPI.h*** de MPIDE, que contiene todas las funciones básicas que controla el acceso del Uno32 al bus SPI.

Vemos como al constructor de la clase debemos de pasar el número de pin del Uno32 al que le conectaremos el pin SS de la RAM que permite su selección y habilitación para la lectura y escritura. En la implementación del constructor en el fichero SpiRam.cpp también fijamos la frecuencia del reloj que el Uno32 como maestro del bus SPI y la RAM como esclavo van a utilizar para la recepción y transmisión de información. Fijamos esta frecuencia en 20 Mhz por medio del divisor de frecuencia que asignamos a la clase SPI base de la librería SPI.h de MPIDE. Esta asignación se realiza mediante la siguiente instrucción:

```
SPI.setClockDivider(CLOCK_DIVIDER_20_MHZ);
```

A partir de aquí, las funciones públicas que exporta la clase realizan la lectura y escritura de bytes, páginas de memoria o bloques, acciones que controlaremos mediante los comandos y los modos definidos en el cabecero del fichero. Las funciones que realmente utilizaremos para nuestro programa, sin embargo, son las que leen y escriben líneas y pixels de nuestra imagen. Las funciones de lectura de pixels se apoyan en la de lectura y escritura de bytes anterior, y la de lectura y escritura de líneas se apoyan en las de bloques. Se trata de una particularizan de las funciones para nuestro caso de una imagen. A todas estas funciones les pasamos un primer parámetro con la imagen seleccionada. Esto es pensando en la posibilidad de que seamos capaces de almacenar más de una imagen en la RAM. Utilizando el tamaño de una imagen en RAM, definido por la constante IMAGE_RAM_RESERVED, accederemos a la imagen que deseemos. Para que las funciones de acceso a pixels y líneas de la imagen funcionen correctamente, deberemos primero establecer la cantidad de columnas de la imagen mediante la función:

```
SpiRam.Set_Image_Columns(Image_Columns);
```

Con la información de la imagen seleccionada, la línea de la imagen, la columna y la cantidad de columnas por línea, seremos capaces de acceder a cada pixel de la imagen y cada línea de esta, ya sea para su lectura o para su escritura.

En las funciones privadas de la clase podemos observar las funciones *enable* y *disable*, que servirán para habilitar y deshabilitar la memoria RAM a través de la activación o no de su pin SS, permitiendo de esta manera el acceso a más esclavos conectados al bus.

Para una descripción mas detalla del funcionamiento de la memoria RAM 23LC1024 de Microchip, véase el ANEXO VII o la documentación adjunta.

5.3.3.- CONTROL DE LA EEPROM

Para que nuestro Star Tracker sea capaz de identificar las estrellas que contiene la imagen tomada por la cámara y posteriormente calcular la actitud del Cubesat, deberemos de tener almacenado el catálogo de estrellas que hemos generado anteriormente en Matlab con la función *catint*, como se ha explicado en el capítulo 4.3. El catálogo lo almacenaremos en una memoria EEPROM 24LC256 o 24FC1024, como se ha detallado en el capítulo anterior, de un tamaño acorde al tamaño que seleccionemos para nuestro catálogo. También almacenaremos cierta información de configuración del sistema que deseamos que se mantenga en memoria tras una pérdida de alimentación y que a su vez pueda ser modificada si es necesario.

El acceso a esta memoria se realiza mediante el bus asíncrono I2C, en el que el Uno32 toma las funciones de maestro y la EEPROM de esclavo. En este bus ya se encuentra conectada la cámara OV7670 y las resistencias de pull-up, por tanto, conectaremos estos puntos comunes a los pines SCL y SDA del Uno32, como muestra la figura 5.22.

Encapsularemos todo el control de la EEPROM que va a contener el catálogo de estrellas y la información de configuración en la clase *Catalog_EEPROM*. El código asociado a esta clase se encuentra en los ficheros *EEPROM.h*, *EEPROM.cpp* y *EEPROM_Defines.h*. El fichero *EEPROM.h* presenta los prototipos de las funciones de la clase, así como sus variables miembro. El fichero *EEPROM.cpp* contiene la implementación de las funciones de la clase. El último fichero contiene

ciertos *defines*, como la dirección I2C de la RAM en el bus, y la definición de estructuras que utilizaremos para guardar de una manera ordenada la información en la EEPROM. La clase hará uso de la librería *Wire.h* disponible en MPIDE y que controla el acceso al bus I2C.

En el fichero *EEPROM_Defines.h* tenemos definidas las estructuras mostradas en la figura 5.39. Con estas estructuras, mantendremos la información en la EEPROM de una manera ordenada.

```
typedef struct {
    uint32_t StarNum_1;
    uint32_t StarNum_2;
    float dist;
} StarPair;

typedef struct {
    float X;
    float Y;
    float Z;
} unit_vector;

typedef struct {
    float focal;
    float FOV;
    float pixel_size;
    uint32_t Resolution_X;
    uint32_t Resolution_Y;
    uint32_t Number_of_StarPairs;
    uint32_t Number_of_Stars;
} HEADER;

typedef struct {
    HEADER header;
    StarPair StarPairs[];
    uint32_t StarID[];
    unit_vector U_Vectors[];
} EEPROM_MAP;
```

FIGURA 5.39- Estructuras para almacenar datos en EEPROM

El mapeado y la correspondencia entre el contenido de la EEPROM y las variables de trabajo viene reflejado perfectamente en la estructura *EEPROM_MAP*. Vemos como los primeros bytes de la EEPROM corresponden a un cabecero o HEADER, de tamaño fijo, que contiene los datos de configuración de nuestra cámara y lente, como son la distancia focal, el FOV y el tamaño de pixel del sensor, así como el tamaño de la imagen en pixels. También almacenaremos el número de pares de estrellas almacenados y número de estrellas que conforman el catálogo. Estas dos últimas variables nos van a servir para dimensionar los arrays de *StarPairs*, *StarID* y *vectorU_Vectors*, así como para calcular los punteros a las direcciones de memoria EEPROM donde

leer cada unos de los elementos de los arrays. La estructura *StarPair* contiene la información relacionada con un par de estrellas del catálogo, es decir, sus Ids y la distancia angular entre ellas. La estructura *unit_vector* contiene el valor en los tres ejes del vector unitario de la estrella del catálogo en el sistema de coordenadas inercial.

El tamaño del HEADER es siempre constante, mientras el tamaño de los arrays es variable. El tamaño de cada uno de los tres arrays de la EEPROM viene codificado en el cabecero. Se trata de una manera dinámica de direccionar la memoria, y que nos permitirá modificar el contenido del catálogo si fuese necesario, sin necesidad de modificar el programa de nuestro *Star Tracker*. Tan sólo modificaremos los tamaños en el HEADER e iremos introduciendo los elementos de los arrays secuencialmente. Como veremos a continuación, estableceremos un procedimiento y protocolo para descargar toda la información desde nuestra aplicación PC de control.

Nuestro objetivo es tener un catálogo de estrellas lo suficientemente grande como para que el algoritmo sea preciso, pero a su vez lo suficientemente pequeño como para que quepa en nuestra EEPROM. El tamaño de la EEPROM es bastante limitado en este aspecto, así que intentaremos afinar al máximo en la selección de los tipos básicos de datos que conforman las variables del catálogo. Por un lado elegiremos todas las variables en formato real con precisión *float*, que representa un número en coma flotante de 4 bytes. Y por otro, elegiremos un tipo de datos uint32 para almacenar los identificadores de estrellas en el catálogo. En el catálogo HYG general usado hay 87.475 estrellas, por tanto deberemos de ser capaces de almacenar al menos ese número. Con un entero sin signo de 16 bits, tenemos el rango (0, 65.535) que no es suficiente, en cambio con 32 bits sin signo alcanzamos un valor máximo de 4.294.967.296, que cumple de sobra con nuestras expectativas.

Una posibilidad para reducir el tamaño consiste en no utilizar el ID de la estrella en el catálogo, sino utilizar un índice. Como hemos visto en el apartado 4.2.2 con un catálogo con 385 estrellas, nuestro algoritmo podemos considerarlo lo suficientemente preciso, por tanto podemos reducir los tipos uint32 a uint16 (hasta 65.535 índices), con lo que reducimos en 4 bytes cada identificador de estrella a almacenar. A la hora de depurar el sistema, deberemos de tener a mano la conversión de esos índices a los IDs reales de las estrellas que representan, para así saber exactamente qué estrellas han sido identificadas por nuestro algoritmo.

Hagamos algunos cálculos del tamaño que tendría nuestro catálogo para así poder seleccionar cual de las dos EEPROM presentadas con anterioridad se ajusta más a nuestras necesidades. El tamaño del cabecero va a ser siempre de 32 bytes. A partir de aquí, en la tabla 5.7 vemos el tamaño que tendría nuestro catálogo de estrellas para nuestro caso de un FOV de 45º dado por la lente de la OV7670. Vamos a contemplar dos posibilidades, una primera con una magnitud de 3.75, y una segunda con una magnitud máxima de 3.5. Para ambos casos veremos el tamaño si consideramos IDs de 32 bits o de 16 bits, como anteriormente hemos explicado.

Cabecero	32 Bytes					
	Tamaño Unidad		**Mag = 3.75**		**Mag = 3.5**	
			385 Estrellas	11.682 Pares	291 Estrellas	6.812 Pares
	Uint32	Uint16	Uint32	Uint16	Uint32	uint16
Pares Estrellas	12 Bytes	8 Bytes	140.184 Bytes	93.456 Bytes	81.744 Bytes	54.496 Bytes
Star IDs	4 Bytes	2 Bytes	1.540 Bytes	770 Bytes	1.164 Bytes	582 Bytes
U-vectors	12 Bytes		4.620 Bytes		3.492 Bytes	
TOTAL			**146.384 Bytes**	**98.878 Bytes**	**86.432 Bytes**	**58.602 Bytes**

TABLA 5.7- Tamaño del catálogo de estrellas en la EEPROM

Vemos como en la EEPROM 24LC256 de 32 KBytes no seremos capaces de almacenar los datos del catálogo, a no ser que disminuyamos el número de estrellas o realicemos algún otro tipo de método para reducir el tamaño de los datos en memoria. Por tanto elegiremos la EEPRFC1026 con 128 Kbytes y utilizaremos los Ids en formato de 16 bits, para así tener un mayor número de estrellas y mejorar la precisión del algoritmo.

Veamos ahora más a fondo la clase *Catalog_EEPROM*. La figura 5.40 muestra los prototipos de las funciones y los miembros que componen la clase.

```
#include "EEPROM_defines.h"

class Catalog_EEPROM {
public:
        Catalog_EEPROM(TwoWire& s);
        HEADER Header;
        void init();
        // Write Functions
        void write_Header(HEADER header);
        void write_Header(uint8_t* buffer);
        void write_StarPair(int num, StarPair Pair);
        void write_StarPair(int num, uint8_t* buffer);
```

```
        void write_StarID(int num, uint32_t StarID);
        void write_StarID(int num, uint8_t* buffer);
        void write_Uvector(int num, unit_vector u_vector);
        void write_Uvector(int num, uint8_t* buffer);
        // Read Functions
        void read_Header();
        void read_Header(uint8_t *buffer);
        void read_StarPair(int num, StarPair *Pair);
        void read_StarPair(int num, uint8_t *buffer);
        void read_StarID(int num, uint32_t *StarID);
        uint32_t read_StarID(int num);
        void read_StarID(int num, uint8_t *buffer);
        void read_Uvector(int num, unit_vector *u_vector);
        unit_vector read_Uvector(int num);
        void read_Uvector(int num, uint8_t *buffer);
private:
        uint8_t initialized;
        TwoWire& I2C;
        uint32_t _lastWrite; // for waitEEReady

        void write_Byte(uint16_t memoryAddress, uint8_t data );
        byte read_Byte (uint16_t memoryAddress );
        void writeBlock(uint16_t memoryAddress, uint8_t* buffer, int length);
        void readBlock(uint16_t memoryAddress, uint8_t* buffer, int length);
        void waitEEReady();
};
```

FIGURA 5.40- Contenido del fichero EEPROM.h

El constructor de la clase, como el caso de la cámara, toma como parámetro una referencia al objeto I2C, derivado de la librería *Wire.h* de MPIDE que controla el interfaz I2C. En las funciones privadas, se encuentran las rutinas necesarias para la lectura y escritura de bytes y bloques de datos en la EEPROM. Se trata de funciones generales de acceso a posiciones de memoria de la EEPROM. Estas funciones utilizan la dirección de la memoria EEPROM en el bus I2C, que el fabricante establece, y que se encuentra definida en el fichero *EEPROM_defines.h* como "*#define EEPROM_ADDDR (int) 0x50* ". Las funciones públicas echan mano de las privadas para proporcionar un interface de acceso más cómodo a la estructura de datos del catalogo de estrellas almacenado en la memoria EEPROM. Podemos ver como tenemos funciones para la lectura y escritura del cabecero, de los pares de estrellas, de los IDs de las estrellas y de sus vectores unitarios. El cabecero, además lo tenemos guardado en la propia memoria RAM del Uno32 en la variable *Header*, para una mayor comodidad, de tal manera que cada vez que modifiquemos el valor en el catálogo, lo guardaremos en la EEPROM y los cargaremos en la RAM para su posterior uso en el resto del programa sin necesidad de una lectura de la EEPROM. Estas funciones de lectura y escritura se encuentran sobrecargadas, para ser utilizadas de la manera que mejor convenga en el programa.

A partir de aquí, sólo queda hacer uso de la clase Catalog_EEPROM en el programa para acceder a la EEPROM. Utilizaremos la variable *Eeprom*, definida en el archivo *Program.h* y que definiremos de la siguiente manera:

```
TwoWire I2C;
Catalog_EEPROM     Eeprom(I2C);
```

Un primer paso consistirá en guardar los datos del catálogo en la EEPROM. Para ello, definimos una serie de comandos con el PC de control a través de los cuales almacenaremos el cabecero, los pares de estrellas, los IDs de las estrellas y sus vectores unitarios. Estos comandos son enviados por el PC a través del puerto serie, y una vez recibidos, el Uno32 pasa a almacenar en la EEPROM vía el puerto I2C ya sea el catálogo, o el resto de la información. Establecemos los comandos que en la figura 5.41 podemos ver representados.

```
<STORE_CATALOG_HEADER>
<READ_CATALOG_HEADER>
<STORE_STAR_PAIR>
<READ_STAR_PAIR>
<STORE_STAR_ID>
<READ_STAR_ID>
<STORE_UVECTOR>
<READ_UVECTOR>
```

FIGURA 5.41- Comando de control de control para el catálogo de estrellas

Podemos ver como tenemos comandos tanto para guardar en la EEPROM como para su posterior lectura, para comprobar la correcta escritura de los datos. Los pares de estrellas, los IDs y los vectores unitarios se almacenan y leen de uno en uno, no todos en un sólo comando, por tanto haremos uso de las funciones en un lazo, hasta completar todos los elementos del array de pares, Ids o vectores unitarios.

5.3.4.- ALGORTIMOS DEL SENSOR ESTELAR

Ya tenemos todo listo para que el algoritmo del sensor estelar lea la imagen de la cámara, la guarde en la RAM, y junto con el catálogo almacenado en la EEPROM, sea capaz de calcular la actitud que presenta el satélite. Mediante el comando <STAR_TRACKER_MODE>, pasamos el Uno32 a modo StarTracker, y realizaremos la secuencia de la figura 3.1 y que ya hemos descrito anteriormente. Hemos adaptado las funciones de nuestro algoritmo probado en Matlab a nuestro hardware de pruebas, cambiando el lenguaje de programación a C++, y teniendo en cuenta que la

imagen se encuentra almacenada en RAM y accedemos a ella vía SPI, y el catálogo se encuentra almacenado en la EEPROM, y accedemos a ella vía I2C.

La función *catinit* de la figura 3.1 no es necesaria. El catálogo lo hemos recibido del PC por el puerto serie. A partir de aquí el catálogo se encuentra almacenado en la EEPROM y leeremos las variables que en él están contenidas a través de la variable *Eeprom* derivada de la clase *Catalog_EEPROM*. La figura 5.42 muestra un ejemplo de acceso al cabecero del catálogo y a la información del par de estrellas i-ésimo.

```
Eeprom.Header.Resolution_X/2
Eeprom.read_StarPair(i, Pair);
```

FIGURA 5.42 Acceso al catálogo de estrellas con la variable Eeprom

La función *loadimage* se realiza mediante la adquisición de una imagen de la cámara OV7670 en la FIFO, y su posterior envío a la RAM. A partir de aquí, la imagen se encuentra disponible en la RAM y accederemos a su contenido según vayamos necesitando en los siguientes pasos del algoritmo de identificación de estrellas de nuestro *Star Tracker*. El acceso de los pixels lo realizaremos mediante la siguiente función, a la que especificaremos la columna y la línea del pixel en la imagen:

uint8_t ReadPixel_RAM(int Column, int Line)

La secuencia que ejecuta el Uno32 para implementar el sensor estelar, tras recibir el comando <STAR_TRACKER_MODE>, se muestra en la figura 5.43

```
// loadimage
CaptureImage_to_FIFO();
GetImage_FIFO_to_RAM(1);

// centroid
Centroides = Centroid (Ithresh, aRoi, &cont_CM);

// uvec
Img_Uvectors = uvec(Centroides, cont_CM);

// starid
starid(Img_Uvectors, cont_CM, Candidate_Uvectors, Catalog_Uvectors, Catalog_Star_IDs, &NumStarsMatched);

// attdet
Attitude = adet(Candidate_Uvectors, Catalog_Uvectors, NumStarsMatched );
```

FIGURA 5.43- Secuencia del Uno32 para el sensor estelar

Al final de la secuencia, obtenemos en la variable *Attitude*, la matriz directriz coseno con la actitud de la cámara en el sistema inercial. Vemos como el proceso comienza tomando la imagen de la cámara y guardándola en la RAM, para calcular los centroides de las estrellas presentes en la imagen con la función *Centroid*. Con estos centroides, calculamos los vectores unitarios de las estrellas candidatas localizadas con la función *uvec*. Pasamos como parámetros estos vectores unitarios a la función *starid* que nos devolverá los vectores unitarios de las estrellas candidatas y los vectores unitarios de sus correspondiente estrellas en el catálogo. Con esta información obtenida, la función *adet* nos proporciona la actitud del satélite.

Estas funciones hacen uso de otras funciones auxiliares también implementadas para el Uno32. La figura 5.44 recoge las más importantes. Utilizaremos las funciones de la librería de MPIDE math.h para calcular medias, raíces cuadradas, cosenos... El fichero *MatrixMath.h* contiene la clase MatrixMath con operaciones de suma, resta, transposición... entre matrices, necesarias para la función *adet*. El fichero SVD.h contiene el código necesario para hallar la descomposición en valores singulares de una matriz, también necesario en la función *adet*.

Además hemos implementado las funciones *BinarySearch* para la búsqueda binaria en el array de pares de estrellas y sus distancias en la EEPROM. Recordemos que en el catálogo se encontrarán ordenados los pares de menor a mayor distancia. También hemos implementado la función *Find*, que busca un valor determinado en un array, la función *norm*, que calcula la norma de un vector, la función *Mult_Uvectors*, que multiplica dos vectores unitarios y finalmente la función *Determinant* que halla el determinante de una matriz 3x3. Se han declarado como *inline* aquellas funciones que se llaman sucesivamente en bucles, evitando la pérdida de eficiencia en los pasos contexto por las llamadas a funciones.

```
#include <math.h>
#include "MatrixMath.h"
#include "SVD.h"
inline int BinarySearch(double Value)
inline int Find(int *Vector, int Size, int Value)
inline double norm(unit_vector vector)
inline static double Mult_Uvectors(unit_vector Uvector_1, unit_vector Uvector_2)
static double Determinant (double** Matriz)
```

FIGURA 5.44- Funciones auxiliares para el sensor estelar

5.3.5.- SOFTWARE DE CONTROL EN PC

Ya hemos comentado anteriormente que el Uno32 ejecuta en su procedimiento cíclico principal una escucha del puerto serie a la espera de las ordenes de control que el PC, a través de una aplicación desarrollada para esta misión, le dé. Este programa está realizado en el lenguaje C# con el entorno de desarrollo Microsoft Visual Studio 2010 (se puede ver su código en la documentación adjunta) y presenta el aspecto que podemos observar en la figura 5.45:

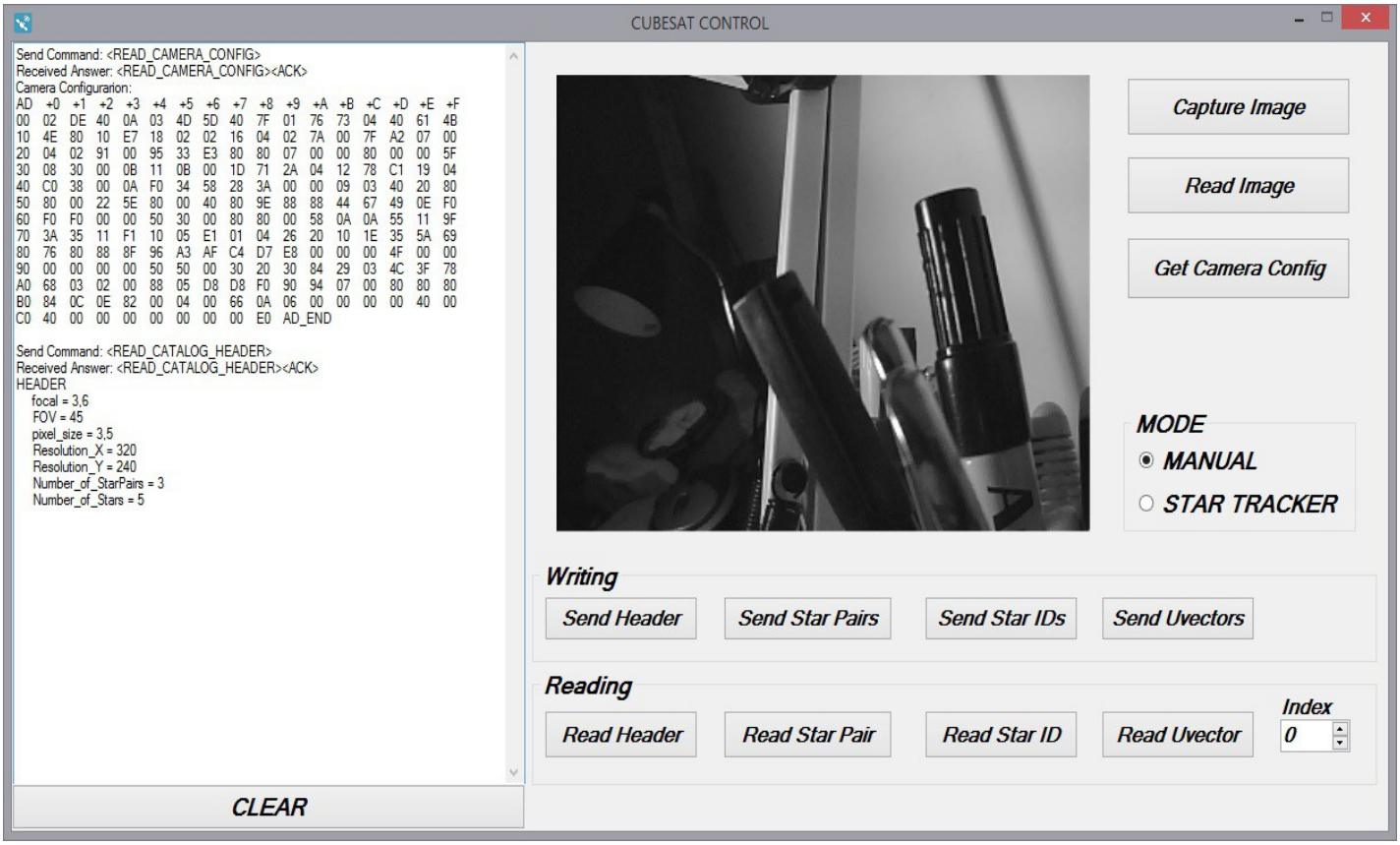


FIGURA 5.45- Aspecto de la aplicación de control del Star Tracker

La aplicación consta de una ventana de eventos en su parte izquierda donde podemos ver todos los comandos que le enviemos al Uno32 así como su respuesta. Esta vrntana se puede limpiar mediante el botón "*Clear*". En la parte central está un cuadro donde representaremos la imagen que el Uno 32 nos envíe cuando le pidamos que lo haga. Además disponemos de una serie de botones de control de la cámara, en la parte derecha de la ventana de la aplicación, y de control de la EEPROM con el catálogo de estrellas en la parte inferior.

El Uno 32 arranca en modo MANUAL por defecto, permitiendo su configuración, y deberemos de pasarlo a modo STAR_TRACKER por medio de la aplicación de control. La comunicación entre la aplicación de control y el Uno32 se realiza mediante el envío y recepción de comandos ASCCII a través del puerto serie de comunicaciones. Veamos a continuación la lista y la descripción de los comandos establecidos:

- ***READ_CAMERA_CONFIG***: Ordena al Uno32 a pedir a la cámara OV7670 todos los registros de control a través del puerto I2C, para posteriormente enviarlos y mostrarlos en la ventana de eventos.

- ***CAPTURE_IMAGE_FIFO***: Pedimos al Uno32 que capture una imagen de la cámara OV7670 a su FIFO y su posterior envío a la RAM por el bus SPI.

- ***READ_IMAGE_FIFO***: Pedimos al Uno32 que nos envíe por el puerto serie la imagen que tenga almacenada en la RAM. Para ello, el Uno32 leerá la imagen de la RAM por el bus SPI, y la enviará al PC por medio del bus serie, línea a línea. Posteriormente la aplicación del PC muestra la imagen en el cuadro central de la ventana de la aplicación.

- ***STORE_CATALOG_HEADER***,: Envío al Uno32 de los datos del cabecero del catálogo de estrellas para que los almacene en la EEPROM vía I2C.

- ***READ_CATALOG_HEADER***: Lectura del Uno32 vía I2C del cabecero del catálogo de estrellas almacenado en la EEPROM y su posterior envío al PC vía serie.

- ***STORE_STAR_PAIR***: Envío al Uno32 de los datos de un par de estrellas del catálogo, es decir, sus IDs y la distancia angular entre ellas, para que los almacene en la EEPROM vía bus I2C. Necesario como parámetro el índice del par en el array de pares de estrellas.

- ***READ_STAR_PAIR***: Lectura del Uno32 vía I2C de los datos de un par de estrellas del catálogo almacenado en la EEPROM y su posterior envío al PC vía serie. Necesario como parámetro el índice del par en el array de pares de estrellas.

- ***STORE_STAR_ID:*** Envío al Uno32 del ID de una estrella del catálogo, para que lo

almacene en la EEPROM vía bus I2C. Necesario como parámetro el índice de las estrellas en el array de IDs de estrellas.

- ***READ_STAR_ID***: Lectura del Uno32 vía I2C del ID datos de una estrella del catálogo almacenado en la EEPROM y su posterior envío al PC vía serie. Necesario como parámetro el índice de las estrellas en el array de IDs de estrellas.

- ***STORE_UVECTOR***: Envío al Uno32 de los datos del vector unitario de una estrella del catálogo para que los almacene en la EEPROM vía bus I2C. Necesario como parámetro el índice del vector unitario en el array de vectores unitarios de estrellas.

- ***READ_UVECTOR***: Lectura del Uno32 vía I2C de los datos del vector unitario de una estrella del catálogo almacenado en la EEPROM y su posterior envío al PC vía serie. Necesario como parámetro el índice del vector unitario en el array de vectores unitarios de estrellas.

- ***MANUAL_MODE***: Envío al Uno32 de la orden de que pase a modo MANUAL, para poder enviar cualquiera de los comandos mencionados anteriormente.

- ***STAR_TRACKER_MODE***: Envío al Uno32 de la orden de que pase a modo STAR_TRACKER para que ejecute el algoritmo de identificación de estrellas y cálculo de la actitud. El Uno32 ya no hará caso a los comandos de modo MANUAL, excepto al comando de cambio a modo MAUAL anterior.

El envío de todos estos comandos está controlado por los botones que presenta la pantalla de la aplicación. Al seleccionar en el sub-cuadro de MODE entre AUTO y STAR_TRACKER, enviaremos los comandos respectivos de cambio de modo al Uno32. Además el resto de botones tienen la siguiente función:

- ***Capture Image:*** Envía al Uno32 el comando <CAPTURE_IMAGE_FIFO>.

- ***Read Image:*** Envía al Uno32 el comando <READ_IMAGE_FIFO> y muestra en el cuadro central la imagen que le encía el Uno32 por el puerto serie.

- ***Get Camera Config:*** Envía al Uno32 el comando <READ_CAMERA_CONFIG> y muestra en la ventana de eventos el resultado de la lectura de todos los registros de la cámara.

- ***Send Header:*** Envía al Uno32 el cabecero del catálogo de estrellas que queremos almacenar en la EEPROM, contenido en un fichero tipo csv cuya ruta la aplicación nos pregunta a través de un cuadro de diálogo. El cabecero se envía mediante el comando <STORE_CATALOG_HEADER>.

- ***Send Star Pairs:*** Envía al Uno32 todos los pares de estrellas del catálogo de estrellas que queremos almacenar en la EEPROM, contenido en un fichero tipo csv cuya ruta la aplicación nos pregunta a través de un cuadro de diálogo. Se recorren todos los pares de estrellas del fichero y se envía de uno en uno con el comando <STORE_STAR_PAIR>.

- **Send Star Ids:** Envía al Uno32 todos los IDs de las estrellas del catálogo de estrellas que queremos almacenar en la EEPROM, contenido en un fichero tipo csv cuya ruta la aplicación nos pregunta a través de un cuadro de diálogo. Se recorren todos los IDs del fichero y se envía de uno en uno con el comando <STORE_STAR_ID>.

- ***Send Uvectors:*** Envía al Uno32 todos los vectores unitarios de las estrellas, del catálogo de estrellas que queremos almacenar en la EEPROM, contenido en un fichero tipo csv cuya ruta la aplicación nos pregunta a través de un cuadro de diálogo. Se recorren todos los los vectores unitarios del fichero y se envía de uno en uno con el comando <STORE_UVECTOR>.

- ***Read Header:*** Se lee del Uno32 el cabecero contenido en la EEPROM a través del bus I2C y posteriormente por el puerto serie que comunica el PC con el Uno32. Se muestra en la ventana de eventos el resultado de la lectura.

- ***Read Star Pair:*** Se leen del Uno32 los datos del par de estrellas que ocupa la posición *Index*-ésima (recogida por el control ***INDEX***) en el array de pares de estrellas en la EEPROM a través del bus I2C y posteriormente por el puerto serie que comunica el PC con el Uno32. Se muestra en la ventana de eventos el resultado de la lectura.

- ***Read Star ID:*** Se lee del Uno32 el ID de la estrella que ocupa la posición *Index*-ésima (recogida por el control ***INDEX***) en el array de Ids de estrellas en la EEPROM a través del bus I2C y posteriormente por el puerto serie que comunica el PC con el Uno32. Se muestra en la ventana de eventos el resultado de la lectura.

- ***Read Uvector:*** Se leen del Uno32 los datos del vector unitario de la estrella que ocupa la posición *Index*-ésima (recogida por el control ***INDEX***) en el array de vectores unitarios de estrellas en la EEPROM a través del bus I2C y posteriormente por el puerto serie que comunica el PC con el Uno32. Se muestra en la ventana de eventos el resultado de la lectura.

- ***Clear:*** borra los datos contenidos en la ventana de eventos de la parte izquierda de la aplicación.

Por medio de estas funciones podemos enviar y comprobar mediante su posterior lectura todos los datos que forman el catálogo de estrellas, así como realizar capturas manuales de la cámara para su calibración y puesta a punto, dotando al sistema de una gran flexibilidad.

6.- CONCLUSIONES

A lo largo de la memoria de este proyecto hemos visto como implementar la algoritmia para el sensor estelar de un Cubesat. Partiendo de los estudios anteriores de MacBride [2] hemos implementado en Matlab los algoritmos necesarios para, una vez tomada una imagen con nuestro sensor estelar, calcular en ella los centroides de las estrellas presentes, y posteriormente, mediante el método de Voting, identificar las estrellas en un catálogo que con anterioridad hemos generado. Finalmente hemos calculado la actitud del Cubesat, dato que será entregado al sistema de control de actitud ACS del Cubesat.

López Suesma [11] definió una arquitectura hardware para el sensor estelar. Hemos modelado la arquitectura hardware y hemos simulado el algoritmo del sensor estelar, mediante la generación de imágenes sintéticas con actitud aleatoria, demostrando su correcto funcionamiento y la precisión del sistema. Hemos comprobado como una mayor cantidad de estrellas identificadas en cada imagen aumentan la precisión del algoritmo, intentando encontrar un equilibrio entre la cantidad de estrellas en el catálogo, y por tanto la cantidad de operaciones del algoritmo, y la precisión del mismo.

Finalmente hemos creado un sistema real de pruebas basado en el sistema hardware anterior, que implementa toda la algoritmia del sensor estelar sobre una placa de prototipo y componentes *low-cost*. Al compartir el mismo procesador nuestro sistema real y el sistema de Lopez Suesma, podremos reutilizar la mayor parte de nuestro código para el diseño final del sensor estelar.

7.- LÍNEAS ABIERTAS

Como futuro trabajo que se podría desarrollar para continuar con este proyecto, estaría la creación de un sistema real de pruebas, generando mediante un proyector u otro dispositivo imágenes que nuestra cámara fuese capaz de tomar, y que simulasen de una forma lo más real posible las condiciones del satélite en el espacio. Se debería de crear un procedimiento de calibración de la cámara para que las imágenes fuesen representativas del fragmento de firmamento que intentan fotografiar.

Como parte de estas pruebas reales tendríamos que determinar si con la lente actual y el catálogo generado y guardado en la EEPROM obtenemos la precisión que deseamos. También deberíamos probar la robustez del sistema ante ruido, ya sea por reflejos u otros objetos o falsas estrellas que puedan aparecer en las imágenes.

El sistema de control mediante comandos diseñado para el sistema de pruebas, mediante la comunicación serie del Uno32 con una aplicación en PC, podría cambiarse por el sistema real con que se comunicará el sensor estelar con el control general del ACS. Se podría modelar y simular el control de ACS con la actitud calculada por el sensor estelar.

8.- BIBLIOGRAFÍA

[1] C. Liebe, *"Accuracy performance of star trackers -a tutorial"*, IEE Transactions on Aerospace and Electronic Systems, vol. 38, no. 2, pp. 587-599, April 2002.

[2] Cristopher Ryan McBryde, *"A Star Tracker Design for CubeSats"*, Thesis presented to the Faculty of the Graduate School of the University of Texas at Austin. May 2012.

[3] M. Kolomenkin, S. Pollak, I. Shimshoni and M. Lindenbaum, *"Geometric voting algorithm for star trackers"*, IEE Transactions on Aerospace and Electronic Systems, vol. 44, no. 2, 2008.

[4] D. Mortari, *"Search-less algorithm for star pattern recognition"*, *J. Astronaut. Sci.*, vol. 45, pp.179-194, 1997.

[5] G. Lerner, "*Three-Axis Attitude Determination",* J. Wertz, Ed. D. Reidel Publishing C0.: D. Reidel Publishing Co., 1978.

[6] M. Shuster and S. Oh, *"Attitude determination from vector observations",* Journal of Guidance and Control, vol. 4, no. 1, pp. 70-77, Jan-Feb 1981.

[7] F. Markley, *"Attitude determination using vector observations and the singular value decomposition",* J. Astronaut, Sci., vol. 38, no. 3, pp. 245-258, 1998.

[8] D. Mortari, M. Samman, and C. Buccoleri, *"The pyramid star identification technique"*, Navigation, vol. 51, pp. 171-183, 2004.

[9] C. Padgett and K. Kreutz-Delgado, *"A grid algorithm for autonomous star identification"*, IEE Trans. Aerospace Electron. Syst., vol. 33, pp. 202-213, 1997.

[10] C. Cole and J. Cassidis, *"Fast star-pattern recognition using planar triangles"*, J. Guid. Control. Dynam., pp. 1283-1286, 1994.

[11] Miguel Ángel López Luesma, *"Subsistema de Control de Actitud (ACS) de un Cubesat"*. Proyecto Fin de Máster en Ingeniería de Sistemas y Control de la UNED. Febrerio 2014.

[12] Sferco, Juan Esteban. *"Determinación de orientación satelital en base a imágenes estelares"*. 1999.

[13] D. Litwiller. *"CMOS vs. CCD: Maturing Technologies, Maturing Markets"*. Photonics Spectra. Laurin Publishing. DALSA Technology with vision. 2005.

[14] David Senabre Albujer. *"GOTO Controlador Autónomo para Telescopios de Aficionados"*. Proyecto Fin de Carrera. Ingeniería Electrónica. Universidad Computense de Madrid. 2012.

[15] Catálogo de estrellas HYG. *http://www.astronexus.com/hyg*

[16] *Marcel J. Sidi. "Spacecraft Dynamics & Control. Apractical Engineering approach".* Cambriddge University Press. 1997.

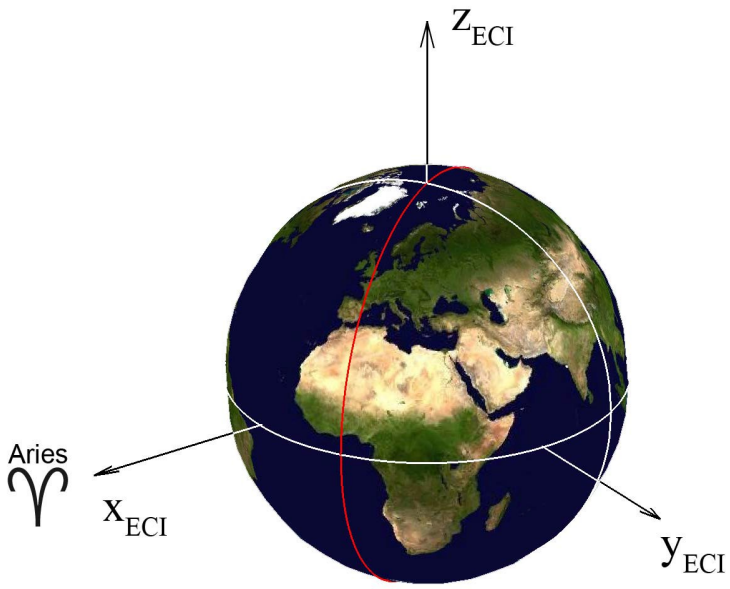
[17] Proyecto Humsat. *http://humsat-uah.org/*

9.- ANEXOS

ANEXO I.- SISTEMAS DE COORDENADAS

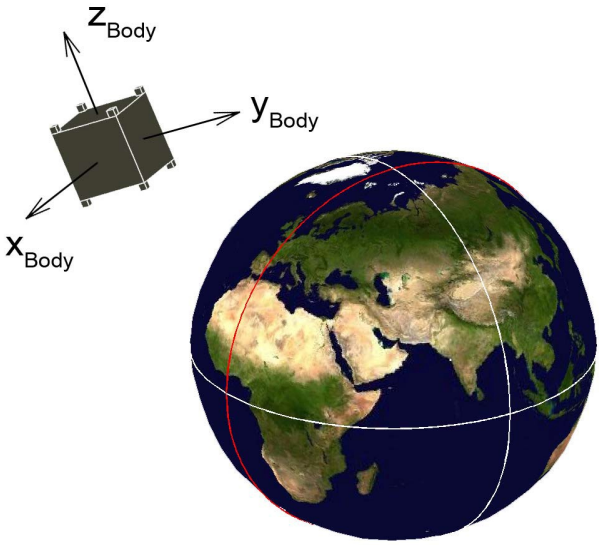
1. Sistema Inercial Centrado en la Tierra (ECI)

El Sistema Inercial centrado en la Tierra (ECI) se centra en la Tierra y se fija en el espacio inercial. El eje X está alineado con el vector radial desde el Sol a la Tierra en el equinoccio de primavera, también conocida como la línea de Aries. El eje Z está alineado con el vector momento angular orbital de la Tierra. El eje Y se forma a partir de la regla de la mano derecha entre los ejes X y Z. La siguiente figura muestra las coordenadas ECI.



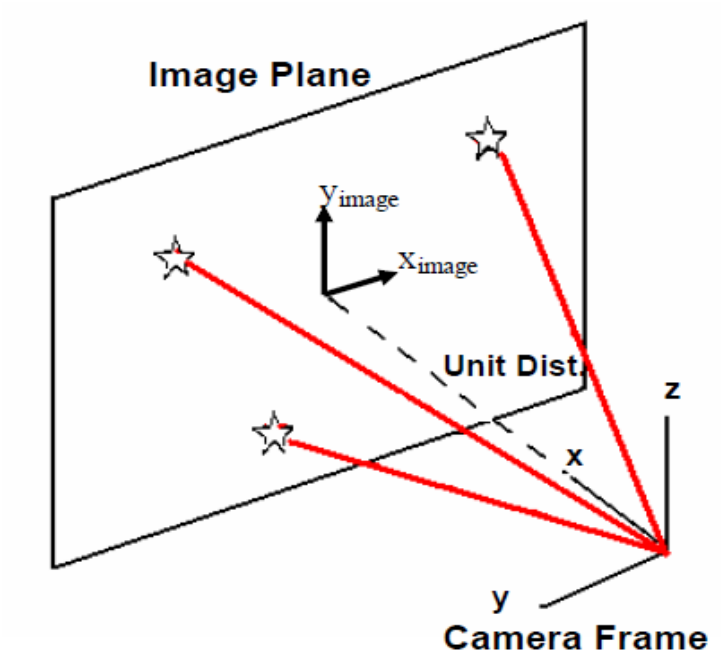
2. Sistema en el Satélite (Body Frame)

El sistema en el satélites está alineado geométricamente con la nave espacial. Los ejes son un conjunto ortogonal de vectores unitarios normalmente alineados con tres de los momentos principales de la inercia de la nave espacial. La siguiente figura muestra este sistema de referencia.



3. Sistema en la Cámara (Camera Frame)

El sistema de referencia de la cámara está alineado con la cámara a bordo de la nave espacial. En aras de la simplicidad se supone que el sistema de la cámara y el sistema del satélite están alineados. Este supuesto se puede generalizar mediante una sola matriz de rotación entre los dos sistemas, una vez que el vehículo final está integrado y las orientaciones de cada sistema son conocidas. En el sistema de la cámara, el eje X es normal al plano de la imagen, en una unidad de distancia. El eje Y de la cámara es paralelo al eje X negativo sobre la imagen, y el eje Z es paralelo al eje Y en la imagen.



ANEXO II.- Código Fuente Matlab

A continuación se muestran el código más importante desarrollado en Matlab para este proyecto. Ver la documentación adjunta para el código completo.

1. Centroid

```
function [ Star_Centers ] = centroid( Image, Ithresh, aRoi )
% Definitions
r_avge = 7;

[X Y] = size(Image);
cont=1;

for x=1:X
   for y=1:Y
       if ( Image(x, y) > Ithresh)
           % Calculate ROI
            x_start = x - ((aRoi -1)/2);
            y_start = y - ((aRoi -1)/2);
            if (x_start<=0 || y_start<=0 )
                break; % Next pixel
            end
            x_end   = x_start + aRoi;
            y_end   = y_start + aRoi;
            if (x_end>X || y_end>Y )
                break; % Next pixel
            end

            % Find the average intensity value of the border pixels Iborder
            I_botton = 0;
            for i=x_start:(x_end-1)
                 I_botton = I_botton + Image(i,y_start);
            end
            I_top = 0;
            for i=(x_start+1):(x_end)
                 I_top = I_top + Image(i,y_end);
            end
            I_left = 0;
            for j=(y_start):(y_end-1)
                 I_left = I_left + Image(x_start,j);
            end
            I_right = 0;
            for j=(y_start+1):(y_end)
                 I_right = I_right + Image(x_end,j);
            end
            Iborder = (I_botton + I_top + I_left + I_right) / (4*(aRoi - 1));

           % Normalized light intensity matrix Inorm
           Inorm = zeros(aRoi-2,aRoi-2);
           for i=(x_start+1):(x_end-1)
                for j=(y_start+1):(y_end-1)
                    Inorm(i,j) = Image(i,j) - Iborder;
                end
           end

           %Calculate the centroid location (xCM; yCM)
           B =0;
           for i=(x_start+1):(x_end-1)
               for j=(y_start+1):(y_end-1)
                   B = B + Inorm(i,j);
               end
           end
           xCM = 0;
           for i=(x_start+1):(x_end-1)
               for j=(y_start+1):(y_end-1)
                   xCM = xCM + (i * Inorm(i,j) / B);
               end
```

```
            end
            yCM = 0;
            for i=(x_start+1):(x_end-1)
                for j=(y_start+1):(y_end-1)
                    yCM = yCM + (j * Inorm(i,j) / B);
                end
            end

             Centroides(cont).xCM = xCM;
             Centroides(cont).yCM = yCM;
             Centroides(cont).processed = 0;
             cont = cont +1;

        end % if ( I2(x, y) > Ithresh)
    end % for y
end % for x

cont_CM =1;
for u=1:cont-1
    if ( Centroides(u).processed ==0)
        Centroides_CM(cont_CM) = Centroides(u);
        Centroides(u).processed =1;
        for v=1:cont-1
            if(u~=v)
                if( (abs(Centroides(v).xCM -  Centroides(u).xCM) <= r_avge) &&
(abs(Centroides(v).yCM -  Centroides(u).yCM) <= r_avge) )
                 Centroides_CM(cont_CM).xCM = (Centroides_CM(cont_CM).xCM  + Centroides(v).xCM) /
2 ;
                 Centroides_CM(cont_CM).yCM = (Centroides_CM(cont_CM).yCM + Centroides(v).yCM) /2 ;
                 Centroides(v).processed =1;
               end
            end
        end %FOR v
        Star_Centers(cont_CM,:) = [Centroides_CM(cont_CM).xCM Centroides_CM(cont_CM).yCM ];
        cont_CM = cont_CM+1;
    end
end % FOR u

end
```

2. starid

```
function [ Candidate_Uvectors, Catalog_Uvectors, Catalog_Star_IDs ] = starid( Image_UnitVectors,
Catalog_StarPairs, Catalog_UnitVectors, Cat_StarsID, FOV, Pixel_Size, tol )

% ********************************************************************************************
% ************** CANDIDATE MATCHING ******************************************************
% ********************************************************************************************
num_candidates = size(Image_UnitVectors,1);
num_stars_catalog = size(Catalog_UnitVectors,1);
num_catalog_starspairs = size(Catalog_StarPairs,1);
matches = zeros(num_candidates,num_stars_catalog);
matches_2 = zeros(num_candidates,num_stars_catalog);

for i=1:num_candidates-1
   for j=(i+1):num_candidates % Symetric distance a->b and b->a
        dist = Image_UnitVectors(i,1:3)*Image_UnitVectors(j,1:3)';
        dist_deg = acos(dist)*180/pi;
        if (dist_deg <= FOV)
             start_index = BinarySearch(Catalog_StarPairs(:,3),(dist_deg - tol));
             for h=uint32(start_index):uint32(num_catalog_starspairs)
                 dist_catalog = Catalog_StarPairs(h,3);
                 if ( (dist_deg - tol)<=dist_catalog && dist_catalog<=(dist_deg + tol) )
                     matches(i,Catalog_StarPairs(h,1)) =  matches(i,Catalog_StarPairs(h,1))+1;
                     matches(i,Catalog_StarPairs(h,2)) =  matches(i,Catalog_StarPairs(h,2))+1;
                     matches(j,Catalog_StarPairs(h,1)) =  matches(j,Catalog_StarPairs(h,1))+1;
                     matches(j,Catalog_StarPairs(h,2)) =  matches(j,Catalog_StarPairs(h,2))+1;
                 end
                 if ( dist_catalog>(dist_deg + tol) )
                    break;
                 end
             end
        end
  end % For j
```

```
end % For i


% En matches tenemos para cada estrella candidata un array con la cantidad
% de veces que ha dado positivo el algoritmo de matching por cada estrella
% del catálogo. Cada posición de este array representa la posición de
% estrella en el array de estrellas del catálogo Para conocer el ID de la
% estrella se debe recuperar el valor de stars
final_matching_stars = zeros(1,num_candidates); % ID de la estrella en el catálogo
final_matching_index = zeros(1,num_candidates); % Índice de la estrella en el catálogo
for k=1:num_candidates
    uu= max(matches(k,:));
    uuu = find(matches(k,:)==uu);
    index = find(matches(k,:)== max(matches(k,:)), 1,'first');
    final_matching_stars(1,k) = Cat_StarsID(1,index);
    final_matching_index(1,k) = index;
end

% ********************************************************************************
% ************** VERIFICATION AND FINAL RESULT ***********************************
% ********************************************************************************
for i=1:num_candidates
  for j=(i+1):num_candidates % Symetric distance a->b and b->a
    dist_cat =
Catalog_UnitVectors(final_matching_index(i),1:3)*Catalog_UnitVectors(final_matching_index(j),1:3)';
    dist_cat = acos(dist_cat)*180/pi;
    dist = Image_UnitVectors(i,1:3)*Image_UnitVectors(j,1:3)';
    dist = acos(dist)*180/pi;
    if ( (dist - tol)<=dist_cat && dist_cat<=(dist + tol) )
         matches_2(i,final_matching_index(i)) =  matches_2(i,final_matching_index(i))+1;
         matches_2(i,final_matching_index(j)) =  matches_2(i,final_matching_index(j))+1;
         matches_2(j,final_matching_index(i)) =  matches_2(j,final_matching_index(i))+1;
         matches_2(j,final_matching_index(j)) =  matches_2(j,final_matching_index(j))+1;
    end
  end % For j
end % For i


cont=0;
T = max(matches_2(:)) -1;
for k=1:num_candidates
    if (sum(matches_2(k,:)) >  T)
         cont = cont +1;
         Catalog_Star_IDs(cont) =  final_matching_stars(1,k);
         FINAL_index(cont) =  final_matching_index(1,k);
         Candidate_Uvectors(cont,:) = Image_UnitVectors(k,:);
         Catalog_Uvectors(cont,:) = Catalog_UnitVectors(FINAL_index(cont),1:3);
    end
end

end
```

3. adet

```
function [ A ] = adet( Candidate_Uvectors, Catalog_Uvectors )

% ********************************************************************************
% ************** ATTITUDE determination ******************************************
% ********************************************************************************
cont = size(Candidate_Uvectors,1);
B = zeros(3,3);
for k=1:cont
    B = B + Candidate_Uvectors(k,:)'*Catalog_Uvectors(k,:);
end

[U,S,V] = svd(B);
U_aux = [1  0   0;
         0  1   0;
         0  0   det(U)];
 V_aux = [1  0   0;
     0  1   0;
     0  0   det(V)];

U_plus = U * U_aux;
```

```
V_plus = V * V_aux;

A = U_plus * V_plus';

end
```

4. Generate_Image

```
function [ IMG, R_I_C, Stars_ID, hand ] = Generate_Image( Resolution_x, Resolution_y, pixel_size,
FOV, focal, Aperture_diameter )

% Camera Parameters:
pixel_size_x = pixel_size;
pixel_size_y = pixel_size;
tau = 1;            % Transmittance
lammda = 0.6;       % Spectral range (0.6um)600nm
l_sat = 13500;      % Saturation limit (photons)
np = 50;
b = 0.015;
t_exp = 0.15;     % Exposure Time

% Random Attitude
alpha = [rand rand rand];
theta = (alpha - [0.5 0.5 0.5]) * (180);
theta = deg2rad(theta);

% Direction cosine matrix
R_1 = [ cos(theta(1))    sin(theta(1))    0;
       -sin(theta(1))    cos(theta(1))    0;
        0                0                1];
R_2 = [ cos(theta(2))    0               -sin(theta(2));
        0                1                0;
        sin(theta(2))    0                cos(theta(2))];
R_3 = [ 1                0                0;
        0                cos(theta(3))    sin(theta(3));
        0                -sin(theta(3))   cos(theta(3))];
R_I_C = R_1 * R_2 * R_3;
u_bore = R_I_C' * [0 0 1]';

% Image Generation
IMG = zeros(Resolution_x,Resolution_y);
IMG_Size_X = Resolution_x * pixel_size_x;
IMG_Size_Y = Resolution_y * pixel_size_y;

cat_hyg_load = load ('cat_hyg.mat');
num_cat = size(cat_hyg_load.Unit_vectors,1);
u_hyg = cat_hyg_load.Unit_vectors; % Unit vector for catalog stars
Mag = cat_hyg_load.Magnitudes;

cont = 1;
for k=1:num_cat
   dist = u_bore' * u_hyg(k,:)';
   dist_deg = acos(dist)*180/pi;
   if ( dist_deg < (FOV/2) )
       star(cont) = cat_hyg_load.Cat_StarsID(k);
       u_star_C  = R_I_C * u_hyg(k,:)';
       u(cont) = focal*(u_star_C(1)/u_star_C(3));
       v(cont) = focal*(u_star_C(2)/u_star_C(3));
       S = t_exp * (pi*(Aperture_diameter^2)/4) * tau * lammda * (10^(-0.4*Mag(k)));
       for j=1:np
           [u_prima(j),v_prima(j)] = rand_circ(1,u(cont),v(cont),b);
           x = round((u_prima(j)/pixel_size_x)+(Resolution_x/2));
           y = round((v_prima(j)/pixel_size_y)+(Resolution_y/2));
           if (x>0 && x<Resolution_x && y>0 && y<Resolution_y)
               IMG(x,y) = IMG(x,y) + (S/np);
           end

       end
       cont = cont +1;
   end;
```

```
end

for i=1:Resolution_x
    for j=1:Resolution_y
       if((IMG(i,j)/l_sat) <= 1)
           IMG(i,j) = IMG(i,j)/l_sat;
       end
       if((IMG(i,j)/l_sat) > 1)
           IMG(i,j) = 1;
       end
    end
end
hand = figure; imshow(IMG); title('Original + Centroides');
hold on;

cont_2= 1;
for i=1:cont-1
    xx = round((u(i)/pixel_size_x)+(Resolution_x/2));
    yy = round((v(i)/pixel_size_y)+(Resolution_y/2));
    if (xx<Resolution_x && xx>0 && yy<Resolution_y && yy>0)
        Stars_ID(cont_2) = star(i);
%         plot(yy ,xx,'o');
%         text(yy+5 ,xx+5, num2str(star(i)),'FontSize',10,'Color',[0 1 1]);
         cont_2 = cont_2 + 1;
    end
end
hold off;

imwrite(IMG,'Fotos/Imagen_1.bmp','bmp');
```

5. MAIN

```
%clearvars;
clear all;
close all;
% General settings
num_Rep = 100;
NumStar_Generated = 0;
NumStar_Indentified = 0;
Camera = 2;  % 1=Kodak, 2=OV7670
GenerateCatalog = 1;
GenerateImage = 1;
% Centroding
Ithresh = 100;
aRoi = 11; % Must to be odd values  %11
% Camera Settings
switch (Camera)
    case 1  % 1=Kodak
        FOV = 60;           % deg
        Mag_limit = 3.75;
        focal =6;           % mm
        pixel_size =0.0075; % mm
        Resolution_y = 648; % Pixels
        Resolution_x = 488; % Pixels
        d = 6000;           % Aperture diameter (6 mm)
    case 2  % 2=OV7670
        FOV = 45;         % deg
        Mag_limit = 3.5;
        focal = 3.6;        % mm
        pixel_size =0.0036; % mm
        Resolution_y = 640; % Pixels
        Resolution_x = 480; % Pixels
        d = 6000;           % Aperture diameter (6 mm)
   end
% Read Catalog
if (GenerateCatalog == 1)
    [ Cat_starPairs, Cat_Uvect, Cat_StarsID] = catinit( 'hygfull.csv', FOV, Mag_limit);
else
    [ Cat_starPairs, Cat_Uvect, Cat_StarsID] = catinit( 'cat_hyg.mat');
end
for rep=1:num_Rep
    if (GenerateImage == 1)
        [IMG, RIC, Original_StarsID, h_1]  = Generate_Image( Resolution_x, Resolution_y, pixel_size,
FOV, focal, d );
```

```
    end
    NumStar_Generated = NumStar_Generated + size(Original_StarsID,2);
    % Load Image
    Image = loading( 'Fotos/Imagen_1.bmp' );

    % Calculate Centroides
    Star_Centers = centroid( Image, Ithresh, aRoi );
    h_2 = figure; imshow(Image); title('Original + Estrellas localizadas');
    hold on;
    for i=1:(size(Star_Centers,1))
        plot(Star_Centers(i,2) ,Star_Centers(i,1),'o');
    end
    hold off;
    % Calculate Unit Vectors of image stars
    Image_Unit_Vectors = uvec( Star_Centers, focal, pixel_size );

    % Star Matching: Voting method
    diagonal = sqrt(Resolution_x^2 + Resolution_y^2);
    tol = FOV / diagonal;
    [ Candidate_Uvectors, Catalog_Uvectors, Catalog_Star_IDs ] = starid( Image_Unit_Vectors,
Cat_starPairs, Cat_Uvect, Cat_StarsID, FOV, pixel_size, tol );
    NumStar_Indentified = NumStar_Indentified + size(Catalog_Star_IDs,2);

    % Attitude Determination
    A = adet( Candidate_Uvectors, Catalog_Uvectors );

    % Error Calculation
    Attitude_Error = inv(A)*RIC;
    Err_x = abs(3600*(90 - rad2deg(acos(Attitude_Error(3,2))))); % arcseconds
    Err_y = abs(3600*(90 - rad2deg(acos(Attitude_Error(3,1))))); % arcseconds
    Err_z = abs(3600*(90 - rad2deg(acos(Attitude_Error(2,1))))); % arcseconds

    % Save Data
    Original_StarsID = sort(Original_StarsID);
    Catalog_Star_IDs = sort(Catalog_Star_IDs);
    Err = [Err_x Err_y Err_z];
    ERROR_TOTAL(rep,:) = Err;
    Path = sprintf('Results/Results_%i',rep);
    mkdir(Path);
    img_orig = strcat(Path, sprintf('/Original_%i',rep),'.jpg');
    img_fin = strcat(Path, sprintf('/Final_%i',rep),'.jpg');
    print(h_1, img_orig,'-djpeg');
    print(h_2, img_fin,'-djpeg');
    close all;

    File =strcat(Path, sprintf('/Result_%i.txt',rep));
    fid = fopen(File,'wt');
    fprintf(fid,'Original:   \t');fprintf(fid,'%i\t',Original_StarsID); fprintf(fid,'\n');
    fprintf(fid,'Calculated:  \t');fprintf(fid,'%i\t',Catalog_Star_IDs); fprintf(fid,'\n');
    fprintf(fid,'Error X-Y-Z: \t');fprintf(fid,'%i\t',Err); fprintf(fid,'\n');
    fclose(fid);

end
% Errores
Total_OK = 0;
File =sprintf('Results/ERROR.txt');
fid = fopen(File,'wt');
fprintf(fid,'Error X-Y-Z: \n');
 for i=1:(size(ERROR_TOTAL,1))
     fprintf(fid,'%i\t',ERROR_TOTAL(i,:)); fprintf(fid,'\n');
     if ( ERROR_TOTAL(i,1)<100 && ERROR_TOTAL(i,2)<100 &&ERROR_TOTAL(i,3)<100 )
         Total_OK = Total_OK+1;
     end
 end
fclose(fid);
% Otros datos
File =sprintf('Results/Results.txt');
fid = fopen(File,'wt');
fprintf(fid,'Estrellas Generadas:   %i\n', NumStar_Generated);
fprintf(fid,'Estrellas Localizadas: %i\n', NumStar_Indentified);
fprintf(fid,'Error Medio X: %d\n', mean(ERROR_TOTAL(:,1)));
fprintf(fid,'Error Medio Y: %d\n', mean(ERROR_TOTAL(:,2)));
fprintf(fid,'Error Medio Z: %d\n', mean(ERROR_TOTAL(:,3)));
fprintf(fid,'Error Medio TOTAL: %d\n', mean([mean(ERROR_TOTAL(:,1)) mean(ERROR_TOTAL(:,2))
mean(ERROR_TOTAL(:,3))] ));
fprintf(fid,'TOTAL Imágenes OK: %d\n', Total_OK);
fclose(fid);
```

Features

- Single 2.5V or 2.7V to 3.6V Supply
- RapidS® Serial Interface: 66 MHz Maximum Clock Frequency
  - SPI Compatible Modes 0 and 3
- User Configurable Page Size
  - 256 Bytes per Page
  - 264 Bytes per Page
  - Page Size Can Be Factory Pre-configured for 256 Bytes
- Page Program Operation
  - Intelligent Programming Operation
  - 4,096 Pages (256/264 Bytes/Page) Main Memory
- Flexible Erase Options
  - Page Erase (256 Bytes)
  - Block Erase (2 Kbytes)
  - Sector Erase (64 Kbytes)
  - Chip Erase (8 Mbits)
- Two SRAM Data Buffers (256/264 Bytes)
  - Allows Receiving of Data while Reprogramming the Flash Array
- Continuous Read Capability through Entire Array
  - Ideal for Code Shadowing Applications
- Low-power Dissipation
  - 7 mA Active Read Current Typical
  - 25 µA Standby Current Typical
  - 5 µA Deep Power Down Typical
- Hardware and Software Data Protection Features
  - Individual Sector
- Sector Lockdown for Secure Code and Data Storage
  - Individual Sector
- Security: 128-byte Security Register
  - 64-byte User Programmable Space
  - Unique 64-byte Device Identifier
- JEDEC Standard Manufacturer and Device ID Read
- 100,000 Program/Erase Cycles Per Page Minimum
- Data Retention – 20 Years
- Industrial Temperature Range
- Green (Pb/Halide-free/RoHS Compliant) Packaging Options

8-megabit 2.5-volt or 2.7-volt DataFlash®

AT45DB081D

1. Description

The AT45DB081D is a 2.5V or 2.7V, serial-interface Flash memory ideally suited for a wide variety of digital voice-, image-, program code- and data-storage applications. The AT45DB081D supports RapidS serial interface for applications requiring very high speed operations. RapidS serial interface is SPI compatible for frequencies up to 66 MHz. Its 8,650,752 bits of memory are organized as 4,096 pages of 256 bytes or 264 bytes each. In addition to the main memory, the AT45DB081D also contains two SRAM buffers of 256/264 bytes each. The buffers allow the receiving of data while a page in the main Memory is being reprogrammed, as well as writing a continuous data stream. EEPROM emulation (bit or byte alterability) is easily handled with a self-contained three step read-modify-write operation. Unlike conventional Flash memories that are accessed randomly with multiple address lines and a parallel interface,

the DataFlash uses a RapidS serial interface to sequentially access its data. The simple sequential access dramatically reduces active pin count, facilitates hardware layout, increases system reliability, minimizes switching noise, and reduces package size. The device is optimized for use in many commercial and industrial applications where high-density, low-pin count, low-voltage and low-power are essential.

To allow for simple in-system reprogrammability, the AT45DB081D does not require high input voltages for programming. The device operates from a single power supply, 2.5V to 3.6V or 2.7V to 3.6V, for both the program and read operations. The AT45DB081D is enabled through the chip select pin ($\overline{CS}$) and accessed via a three-wire interface consisting of the Serial Input (SI), Serial Output (SO), and the Serial Clock (SCK).

All programming and erase cycles are self-timed.

2. Pin Configurations and Pinouts

Figure 2-1. MLF (VDFN) Top View

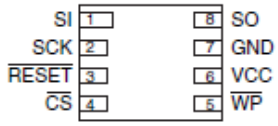
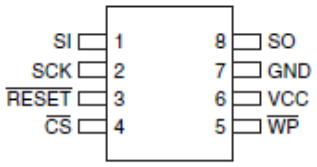


Figure 2-2. SOIC Top View



Note: 1. The metal pad on the bottom of the MLF package is floating. This pad can be a "No Connect" or connected to GND.

ANEXO IV.- Memoria SRAM CY7C1061

CYPRESS

CY7C1061AV33

1M x 16 Static RAM

Features

- High speed
  - —t_{AA} = 8, 10, 12 ns
- Low active power
  - —1080 mW (max.)
- Operating voltages of 3.3 ± 0.3V
- 2.0V data retention
- Automatic power-down when deselected
- TTL-compatible inputs and outputs
- Easy memory expansion with $\overline{CE}_1$ and CE_2 features

Functional Description

The CY7C1061AV33 is a high-performance CMOS Static RAM organized as 1,048,576 words by 16 bits.

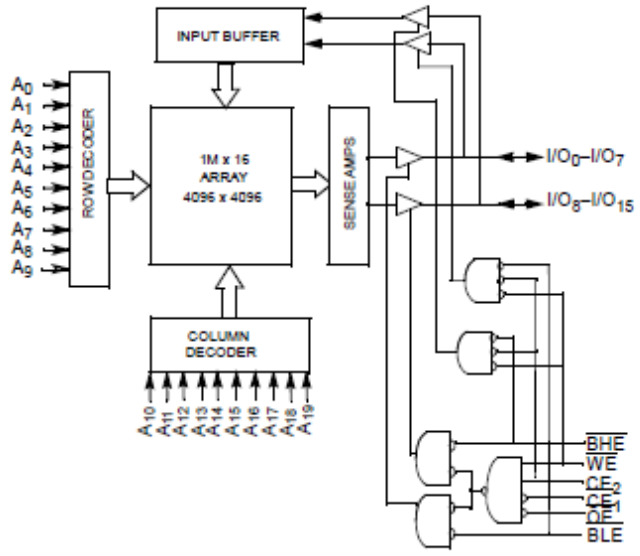
Writing to the device is accomplished by enabling the chip ($\overline{CE}_1$ LOW and CE_2 HIGH) while forcing the Write Enable ($\overline{WE}$) input LOW. If Byte Low Enable ($\overline{BLE}$) is LOW, then data from I/O pins (I/O_0 through I/O_7), is written into the location specified on the address pins (A_0 through A_{19}). If Byte High Enable ($\overline{BHE}$) is LOW, then data from I/O pins (I/O_8 through I/O_{15}) is written into the location specified on the address pins (A_0 through A_{19}).

Reading from the device is accomplished by enabling the chip by taking $\overline{CE}_1$ LOW and CE_2 HIGH while forcing the Output Enable ($\overline{OE}$) LOW and the Write Enable ($\overline{WE}$) HIGH. If Byte Low Enable ($\overline{BLE}$) is LOW, then data from the memory location specified by the address pins will appear on I/O_0 to I/O_7. If Byte High Enable ($\overline{BHE}$) is LOW, then data from memory will appear on I/O_8 to I/O_{15}. See the truth table at the back of this data sheet for a complete description of Read and Write modes.

The input/output pins (I/O_0 through I/O_{15}) are placed in a high-impedance state when the device is deselected ($\overline{CE}_1$ HIGH/CE_2 LOW), the outputs are disabled ($\overline{OE}$ HIGH), the $\overline{BHE}$ and $\overline{BLE}$ are disabled ($\overline{BHE}$, $\overline{BLE}$ HIGH), or during a Write operation ($\overline{CE}_1$ LOW, CE_2 HIGH, and $\overline{WE}$ LOW).

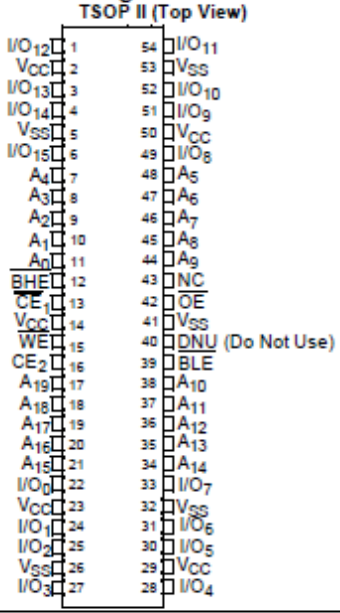
The CY7C1061AV33 is available in a 54-pin TSOP II package with center power and ground (revolutionary) pinout, and a 48-ball fine-pitch ball grid array (FBGA) package.

Logic Block Diagram



Pin Configuration

TSOP II (Top View)



Selection Guide

		-8	-10	-12	Unit
Maximum Access Time		8	10	12	ns
Maximum Operating Current	Commercial	300	275	260	mA
	Industrial	300	275	260	
Maximum CMOS Standby Current	Commercial/Industrial	50	50	50	mA

ANEXO V.- CHIPKIT UNO32

chipKIT™ Uno32™ Board Reference Manual

Revision: July 17, 2012

Note: This document applies to REV C of the board.

DIGILENT
www.digilentinc.com
1300 NE Henley Court, Suite 3
Pullman, WA 99163
(509) 334 6306 Voice | (509) 334 6300 Fax

Overview

The chipKIT Uno32 is based on the popular Arduino™ open-source hardware prototyping platform and adds the performance of the Microchip PIC32 microcontroller.

The Uno32 is the same form factor as the Arduino Uno board and is compatible with Arduino shields. It features a USB serial port interface for connection to the IDE and can be powered via USB or an external power supply.

The Uno32 board takes advantage of the powerful PIC32MX320F128 microcontroller. This microcontroller features a 32-bit MIPS processor core running at 80Mhz, 128K of flash program memory and 16K of SRAM data memory.

The Uno32 can be programmed using the Multi-Platform Integrated Development Environment (MPIDE), an environment based on the original Arduino IDE modified to support PIC32. It contains everything needed to start developing embedded applications.

In addition, the Uno32 is fully compatible with the advanced Microchip MPLAB® IDE and the PICKit3 in-system programmer/debugger.

The Uno32 is easy to use and suitable for both beginners and advanced users experimenting with electronics and embedded control systems.

The Uno32 provides 42 I/O pins that support a number of peripheral functions, such as UART, SPI, and I^2C ports and pulse width modulated outputs. Twelve of the I/O pins can be used as analog inputs or as digital inputs and outputs.

Features include:

- Microchip® PIC32MX320F128H microcontroller (80 Mhz 32-bit MIPS, 128K Flash, 16K SRAM)
- compatible with many existing Arduino code samples and other resources
- Arduino Uno form factor
- compatible with many Arduino shields
- 42 available I/O pins
- two user LEDs
- PC connection uses a USB A > mini B cable (not included)
- 12 analog inputs
- 3.3V operating voltage
- 80Mhz operating frequency
- 75mA typical operating current
- 7V to 15V input voltage (recommended)
- 20V input voltage (maximum)
- 0V to 3.3V analog input voltage range
- +/-18mA DC current per pin

OmniVision

Advanced Information
Preliminary Datasheet

OV7670/OV7171 CMOS VGA (640x480) CAMERACHIP™ Sensor with OmniPixel® Technology

General Description

The OV7670/OV7171 CAMERACHIP™ image sensor is a low voltage CMOS device that provides the full functionality of a single-chip VGA camera and image processor in a small footprint package. The OV7670/OV7171 provides full-frame, sub-sampled or windowed 8-bit images in a wide range of formats, controlled through the Serial Camera Control Bus (SCCB) interface.

This product has an image array capable of operating at up to 30 frames per second (fps) in VGA with complete user control over image quality, formatting and output data transfer. All required image processing functions, including exposure control, gamma, white balance, color saturation, hue control and more, are also programmable through the SCCB interface. In addition, OmniVision sensors use proprietary sensor technology to improve image quality by reducing or eliminating common lighting/electrical sources of image contamination, such as fixed pattern noise (FPN), smearing, blooming, etc., to produce a clean, fully stable color image.

Note: The OV7670/OV7171 uses a lead-free package.

Features

- High sensitivity for low-light operation
- Low operating voltage for embedded portable apps
- Standard SCCB interface compatible with I2C interface
- Output support for Raw RGB, RGB (GRB 4:2:2, RGB565/555/444), YUV (4:2:2) and YCbCr (4:2:2) formats
- Supports image sizes: VGA, CIF, and any size scaling down from CIF to 40x30
- VarioPixel® method for sub-sampling
- Automatic image control functions including: Automatic Exposure Control (AEC), Automatic Gain Control (AGC), Automatic White Balance (AWB), Automatic Band Filter (ABF), and Automatic Black-Level Calibration (ABLC)
- Image quality controls including color saturation, hue, gamma, sharpness (edge enhancement), and anti-blooming
- ISP includes noise reduction and defect correction
- Supports LED and flash strobe mode
- Supports scaling
- Lens shading correction
- Flicker (50/60 Hz) auto detection
- Saturation level auto adjust (UV adjust)
- Edge enhancement level auto adjust
- De-noise level auto adjust

Ordering Information

Product	Package
OV07670-VL2A (Color, lead-free)	24 pin CSP2
OV07171-VL2A (B&W, lead-free)	24 pin CSP2

Applications

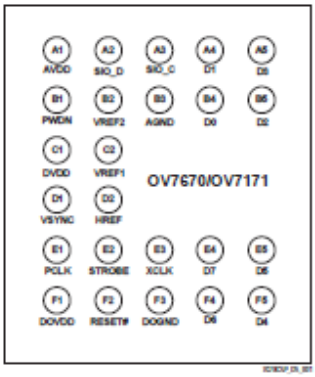
- Cellular and Picture Phones
- Toys
- PC Multimedia
- Digital Still Cameras

Key Specifications

Active Array Size		640 x 480
Power Supply	**Digital Core**	1.8VDC ±10%
	Analog	2.45V to 3.0V
	I/O	1.7V to 3.0V[a]
Power Requirements	**Active**	60 mW typical (15fps VGA YUV format)
	Standby	< 20 µA
Temperature Range	**Operation**	-30°C to 70°C
	Stable Image	0°C to 50°C
Output Formats (8-bit)		• YUV/YCbCr 4:2:2 • RGB565/555/444 • GRB 4:2:2 • Raw RGB Data
Lens Size		1/6"
Chief Ray Angle		25°
Maximum Image Transfer Rate		30 fps for VGA
Sensitivity		1.3 V/(Lux • sec)
S/N Ratio		46 dB
Dynamic Range		52 dB
Scan Mode		Progressive
Electronics Exposure		Up to 510:1 (for selected fps)
Pixel Size		3.6 µm x 3.6 µm
Dark Current		12 mV/s at 60°C
Well Capacity		17 K e
Image Area		2.36 mm x 1.76 mm
Package Dimensions		3785 µm x 4235 µm

a. I/O power should be 2.45V or higher when using the internal regulator for Core (1.8V); otherwise, it is necessary to provide an external 1.8V for the Core power supply.

Figure 1 OV7670/OV7171 Pin Diagram (Top View)



Version 1.4, August 21, 2006

ANEXO VII.- SENSOR OV7670

MICROCHIP

23A1024/23LC1024

1Mbit SPI Serial SRAM with SDI and SQI Interface

Device Selection Table

Part Number	Vcc Range	Temp. Ranges	Dual I/O (SDI)	Quad I/O (SQI)	Max. Clock Frequency	Packages
23A1024	1.7-2.2V	I, E	Yes	Yes	20 MHz(1)	SN, ST, P
23LC1024	2.5-5.5V	I, E	Yes	Yes	20 MHz(1)	SN, ST, P

Note 1: 16 MHz for E-temp.

Features:

- SPI-Compatible Bus Interface:
  - 20 MHz Clock rate
  - SPI/SDI/SQI mode
- Low-Power CMOS Technology:
  - Read Current: 3 mA at 5.5V, 20 MHz
  - Standby Current: 4 µA at +85°C
- Unlimited Read and Write Cycles
- Zero Write Time
- 128K x 8-bit Organization:
  - 32-byte page
- Byte, Page and Sequential mode for Reads and Writes
- High Reliability
- Temperature Ranges Supported:
  - Industrial (I): -40°C to +85°C
  - Automotive (E): -40°C to +125°C
- RoHS Compliant
- 8 Lead SOIC, TSSOP and PDIP Packages

Pin Function Table

Name	Function
$\overline{CS}$	Chip Select Input
SO/SIO1	Serial Output/SDI/SQI Pin
SIO2	SQI Pin
VSS	Ground
SI/SIO0	Serial Input/SDI/SQI Pin
SCK	Serial Clock
$\overline{HOLD}$/SIO3	Hold/SQI Pin
VCC	Power Supply

Description:

The Microchip Technology Inc. 23A1024/23LC1024 are 1 Mbit Serial SRAM devices. The memory is accessed via a simple Serial Peripheral Interface (SPI) compatible serial bus. The bus signals required are a clock input (SCK) plus separate data in (SI) and data out (SO) lines. Access to the device is controlled through a Chip Select ($\overline{CS}$) input. Additionally, SDI (Serial Dual Interface) and SQI (Serial Quad Interface) is supported if your application needs faster data rates.

This device also supports unlimited reads and writes to the memory array.

The 23A1024/23LC1024 is available in standard packages including 8-lead SOIC, PDIP and advanced 8-lead TSSOP.

Package Types (not to scale)

SOIC/TSSOP/PDIP