Microcontroller Systems for a UAV
Auto Piloting and Camera Trigger System

Alexandros Skafidas
# Table of Contents

A. Abstract .................................................................................................................. 3
B. Acknowledgements ................................................................................................. 3
C. Objective ................................................................................................................. 3
1. Introduction .............................................................................................................. 4
2. What is a Microcontroller ....................................................................................... 4
    2.1. What is a UAV? ............................................................................................... 5
    2.2. Software development for microcontrollers ................................................... 5
    2.3. Servos .............................................................................................................. 7
    2.4. GPS Unit & NMEA Protocol ......................................................................... 7
    2.5. SPI Protocol .................................................................................................... 8
3. Camera Trigger Unit ............................................................................................... 9
    3.1. Details of Camera Trigger Unit ...................................................................... 10
        3.1.1. Camera ................................................................................................... 10
        3.1.2. External EEPROM IC .......................................................................... 11
    3.2. Camera Trigger Program Flow ....................................................................... 11
        3.2.1. ICP Interrupt ......................................................................................... 11
        3.2.2. Timer Interrupt ...................................................................................... 12
        3.2.3. UART Interrupt ..................................................................................... 12
        3.2.4. Saving picture information to the EEPROM ............................................ 13
        3.2.5. Uploading picture information to a PC .................................................... 14
4. Waypoint Direction Finder Unit .......................................................................... 16
    4.1. Waypoint Direction Finder Program Flow .................................................... 17
    4.2. Future improvements ....................................................................................... 17
5. Altimeter / Altitude Hold Unit ............................................................................... 18
    5.1. Pressure Sensor ............................................................................................... 18
    5.2. Analogue to Digital Converter ....................................................................... 19
    5.3. Altimeter Program Flow ................................................................................. 20
    5.4. Altitude from Pressure .................................................................................... 20
    5.5. Accuracy .......................................................................................................... 21
    5.6. Signal Conditioning ......................................................................................... 21
    5.7. Future Improvements ....................................................................................... 22
D. Conclusion .............................................................................................................. 23
E. Abbreviations ......................................................................................................... 24
F. Sources .................................................................................................................... 25
G. Internet Links .......................................................................................................... 25
H. Appendix (Program Flow Charts) ......................................................................... 26
    a. Camera Trigger Unit ......................................................................................... 26
    b. Waypoint Direction Finder Unit ...................................................................... 26
    c. Altimeter / Altitude Hold Unit .......................................................................... 26
J. Appendix (C-Code) ................................................................................................. 35
    a. Altitude Hold ...................................................................................................... 35
    b. Waypoint Direction Finder Unit ...................................................................... 39
    c. Camera Trigger Unit ......................................................................................... 43
    d. Common Subroutines (LCD, Delay) ................................................................. 49
    e. NMEA field extracting code .......................................................................... 51
K. Appendix (Schematics) ......................................................................................... 53
L. Appendix (PCB) ...................................................................................................... 56
A. Abstract

This report describes the development of three electronic devices, primarily designed for use aboard an unmanned aerial vehicle (UAV). Descriptive names were given to these devices (also called units) to emphasize their function. These devices are:

- “Camera Trigger Unit”
- “Waypoint Direction Finder Unit”
- “Altimeter / Altitude Hold Unit”

The first device connects to a camera and a commercial Global Positioning System (GPS) receiver. A photograph can be snapped manually from the ground with a receiver / transmitter, or automatically when the GPS overflies a preprogrammed location.

Secondly the waypoint direction finder unit is used to guide/steer the aircraft towards a preprogrammed waypoint or a route of waypoints. This is achieved partly with help of the onboard GPS.

Finally the altimeter unit is used for measuring altitude. Indications are given about how it can be further developed into an “altitude hold unit”.

Each device is built with various electronic components, but they all have one thing in common. Each device has a special kind of microprocessor, called a microcontroller which is the “heart” of the device. Most of the work for this project involved the development of software for these microcontrollers. As a result this report gives a detailed description of the program flow. A description of the various physical parts that make up these units is also given. The production phase was completed with the design and manufacturing of printed circuit boards, thus making these units truly stand alone electronic devices.

All the units met the requirements and some units even offered additional features not included in the original plans. The project proved to be very educative, as it involved all sorts of debugging, from hardware to software. Theory was always put to test in the constant interaction between theory and practice. In addition it offered a good understanding of the development stages of an electronic product from a rough sketch to a working device.

B. Acknowledgements

I would like to thank the following people for their financial, as well as their educational support.

- Thomas Lindblad, KTH (Royal Institute of Technology)
- Nils Hollman, NDH Marketing
- Sten Hollman, NDH Marketing
- Johan Puscov

C. Objective

The objective is to produce a number of electronic microcontroller based devices for integration into the autopiloting and payload system of an unmanned aerial vehicle. The autopiloting unit should be responsible for steering the aircraft towards a predefined waypoint, or along a route of waypoints. It should also attempt to stabilize the aircraft to a certain altitude. Another device should be able to trigger the UAV’s payload, which consists of a digital camera.
1. Introduction

The given assignment was to construct a number of electronic devices to be used aboard a UAV. These devices are as previously mentioned: a “camera trigger unit”, a “waypoint direction finder unit” and an “altimeter/altitude hold unit”.

The easiest way to meet the requirements specified by the project’s objective with a limited amount of hardware was by the use of a microcontroller. But the decreased complexity in hardware offered by the microcontroller results in increased complexity in the software within it. It was natural to expect that this project would require a considerable amount of software, but also hardware skills. Before going into details about these three units, a description of all the components used to build them will be given.

2. What is a Microcontroller?

“A microcontroller is a computer-on-a-chip, or, if you prefer, a single-chip computer. Micro suggests that the device is small, and controller tells you that the device might be used to control objects, processes, or events.” [1]

“Primarily, the microcontroller is capable of storing and running a program (its most important feature). The microcontroller contains a CPU, RAM, ROM, I/O lines, serial and parallel ports, timers, and sometimes other built-in peripherals such as A/D and D/A converters.” [2]

The “heart” of each of the three devices described in this project is in fact a microcontroller. Two different models were used; the AT90S2313 [Figure 2] and the more powerful AT90S8515 [Figure 1]. Both are 8 bit microcontrollers with RISC architecture manufactured by Atmel. They belong to a family of microcontrollers known by the meaningless three letter combination: AVR.

<table>
<thead>
<tr>
<th>AT90S2313</th>
<th>AT90S8515</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Memory</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>RAM</td>
<td>128</td>
<td>512</td>
</tr>
<tr>
<td>EEPROM</td>
<td>128</td>
<td>512</td>
</tr>
<tr>
<td>Max Clock Speed</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>UART</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SPI</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>8bit timer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>16bit timer</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>I/O Lines</td>
<td>15</td>
<td>32</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>2.8</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1. Comparison between the AT90S2313 and AT90S8515 microcontrollers.

Their program memory is used to store and execute the compiled C, or assembly code (machine language). The program memory is of Flash type, which has many advantages over other types of memory. Most important of all, it is rapidly erasable and programmable. This is useful for reducing the time between “burning” and evaluation of the compiled code.

The EEPROM can be compared to a PC’s hard-disk. Its most important feature is that, unlike RAM, it can retain its memory contents even after the power has
been turned off. In microcontroller applications it is used for saving initialization or calibration information. It is important to point out that the EEPROM also has two main disadvantages. Unlike RAM memory, this kind of memory has a limited life expectancy. The memory gets “worn out” after around 100,000 read/write cycles. Therefore the program should be written so that it makes little use of the EEPROM, and if possible, no use at all. The second disadvantage is that the “write time” can be 5ms, but since programs rarely write to the EEPROM, if ever, this does usually not pose a problem.

The clock speed basically decides how fast the code gets executed. Most of the simple assembler instructions can be executed in a single clock cycle. This means that, at least theoretically, at 8MHz the microcontroller is executing 8 MIPS.

The full duplex UART is one of the most exploited microcontroller features in this project. Polled or interrupt mode can be used to check for incoming characters. To reduce the load on the program, the interrupt mode was chosen. Every incoming character fires an interrupt which redirects the program flow to a part of the code responsible for handling the incoming characters. In the UART interrupt subroutines the program usually waits for a special sequence of characters, in other words, it filters the incoming stream. Requested characters are temporarily stored in a string in RAM memory.

The I/O ports are of course essential to a microcontroller, for its ability to control. The AT90S8515 delivers four ports with 8 pins each, a total of 32 I/O pins. Any pin can be configured as an input or an output, and this pin assignment does not have to remain static but can change dynamically in runtime. Very few peripheral devices were used in this project, and such a big number of I/O pins was not required.

All signal levels are digital, so the microcontroller can connect only to digital devices. The pressure sensor used in the altimeter unit is an analogue device. Therefore for it to interact with the microcontroller, an ADC was used, to convert from analogue to digital signal levels.

Two timers/counters an 8bit and a 16bit, are available in both microcontrollers for counting and timing purposes. The number of bits determines the accuracy of the measurement. A timer is for example used for measuring the width of a pulse in the Camera Trigger Unit.

2.1. What is a UAV?
The acronym UAV stands for Unmanned Aerial Vehicle. A UAV is usually a small sized aircraft, able to fly autonomously thanks to advanced electronics aboard. It is a vehicle which, with no external input, can control itself and perform given tasks. It can essentially be seen as a robot that can fly.

There is no one ‘regular’ shape or size for UAVs, they are usually designed specifically for a task, and that task only. Some of the more common current UAV tasks involve aerial surveillance, aerial surveying and meteorological measurement.

2.2. Software development for microcontrollers
Programming a microcontroller normally involves the following steps:

• Coding / Debugging in a high level language (such as C, or Java) or assembler.
This is the most time consuming step. A compiler for a high level language helps to reduce production time. To program the microcontrollers the ICCAVR PRO v6 C-compiler from Imagecraft was used. Although inline assembly was possible, the programming was done strictly in the C language. The source code has been commented to facilitate any occasional future improvement and maintenance.

• **Compiling the code into machine language.**

The machine language file (.hex) does not make any sense to the layman, (or even to an experienced programmer) when examined with a text editor. But it is the only thing the microcontroller will understand, because it contains the original program code converted into a hexadecimal format. During this step the programmer will be warned about eventual errors in the program. An interesting feature in the professional version of the ICCAVR program (in which the code was written) was the so called Code-compressor option, which allowed more code to be packed into the program memory. When the program seems to be complete and free of errors the programmer can move to the next step.

• **Burning the machine language (hex) file into the microcontroller’s program memory.**

This is achieved with a dedicated programmer, which attaches to a PC’s peripheral (such as a COM, or a parallel port). In this project the STK500 programmer transferred the machine language file into the microcontroller’s program memory.

Part of this project involved debugging. The AVR Studio v4.0 software besides of communicating with the programmer, it offered advanced debugging options. Despite this, debugging was not done with the traditional debugging software, but with a standard LCD to show variables of interest.

Communications problems that arose between different integrated circuits (SPI communication), were resolved with a digital oscilloscope.

The microcontrollers were programmed in approximately two seconds with a high speed programming mode. The program memory which is of Flash type, has, just like the EEPROM, a limited lifespan. On the AVR microcontroller family it may be reprogrammed up to a thousand times without any risk of data corruption.

• **Evaluation**

If the microcontroller performs all the required tasks and behaves as expected the software development phase is over. If not, the whole procedure will have to be repeated again.

Most of the work with this project involved the above steps, repeated in a circular fashion, until evaluation of the unit revealed if the requirements were fulfilled.

One of the difficulties of programming microcontrollers is the limited amount of resources the programmer has to deal with. In PCs resources such as RAM and processing speed are basically limitless when compared to microcontrollers. In contrast to a PC, the code on microcontrollers should be as low on resources as possible.
2.3. **Servos**

Servos like the one in the picture [Figure 4] are widely used in hobby related constructions. Most of the moving parts of a UAV are controlled by these servos, which in turn, are controlled by a transmitter from the ground. The lever position on a servo is controlled by a square pulse (Pulse Width Modulation) signal. The period of the pulse regulates the lever position. The period varies between one and two milliseconds. A 1ms pulse sets the lever on one end and a 2ms pulse sets the lever on the opposite end. The relationship between the lever position and the pulse period is linear; from this follows that a 1.5ms pulse is required to place the lever in the central position. If the servo is of the latest generation, known as the “digital” servo, then the signal does not have to be repeated. But these servos are at present time still not so common.

A typical servo requires a constant pulse train to guarantee that its lever will stay in the same position. Therefore the receivers that connect to these servos will regenerate the pulse every 20ms.

![Figure 5. Timing characteristics of the PWM signal, used to control servos.](image)

2.4. **GPS Unit & NMEA Protocol**

The GPS receiver used was the *GPS315* manufactured by *Magellan*. For this project it was important that the GPS unit could transmit its information to the outside world. This GPS receiver does in fact send all data out to a COM port.

The data is sent in standardized protocol called NMEA developed to enable data exchange between electronic marine instruments, navigation equipment in a standardized way. Data between devices is transferred in the form of "sentences". An NMEA sentence always starts with the dollar sign ($), followed by a two character *talker* identifier, e.g. "GP" for GPS. The next three characters are called the "sentence formatter" to define the type and format of the data that follows, e.g. "GLL" (geographic position Lat/Lon). After this the data fields follow, separated by a comma. "Null" fields are allowed in a sentence, and most data fields may be variable in length. Numeric variable fields may also contain a decimal
point and leading/trailing zeros. A sentence should be concluded by an "*" followed by a valid checksum.

Example of an NMEA sentence:
$GPAPB,A,A,0.0,L,N,A,,357.3,M,SIM002,347.8,M,,,*6A

2.5. **SPI Protocol**

SPI (Serial Peripheral Interface) is a synchronous serial link used for exchange of data between integrated circuits on printed circuit boards (PCBs). It is designed for high speeds and a minimum of external components. An SPI system consists of two data lines and two control lines.

*Master Out Slave In (MOSI)*
This data line supplies the output data from the master which is shifted into the input(s) of the slave(s).

*Master In Slave Out (MISO)*
This data line supplies the output data from a slave to the input of the master. There may be no more than one slave which is transmitting data during any particular transfer. (Guaranteed by the SS line, see below)

*Slave Select / Chip Select (SS or CS)*
This control line allows slaves to be turned on and off with hardware control. Normally only one slave can be activated at a given time

*Serial Clock (SCLK)*
This control line is driven by the master and regulates the flow of the data bits. The master may transmit data at a variety of baud rates; the SCLK line cycles once for each bit that is transmitted.
3. Camera Trigger Unit

A usual task for UAVs in general and military UAVs especially, is reconnaissance. That is why the preferred payload in our UAV had to be either a video transmitter/recorder or a camera. A video can be set to transmit/record manually prior to takeoff and stopped after landing. A camera on the other hand has to be triggered. This could be done in a number of ways. If there is no specific target to be monitored, and only the path of the UAV is of interest, then a timer could be used to trigger the camera in predefined intervals. Although the UAV is supposed to fly autonomously, the payload could be controlled from the ground. Therefore a camera could be triggered with a standard transmitter. But there is still no real way of taking a picture over a specific target simply by watching the UAV from a distance. This is especially true if the UAV is to be flown far away from its operational base. Therefore it is a good idea to use the GPS located in the UAV and to extract positioning information for triggering the camera (or any other payload) to take a picture over a specific coordinate. The target of interest is to be programmed into the GPS as a waypoint prior to take-off.

Due to the inability of a camera to focus on a moving target (ground below) a very high percentage of the pictures taken from UAVs are blurred. Therefore it was normal to settle for a digital camera, where only satisfactory pictures are saved, and the rest can just be deleted from memory.

Figure 7. Camera Trigger Unit

One of the earliest approaches for triggering the camera was a simple mechanical solution consisting of a servo-lever pushing down the trigger button. Unfortunately
this approach proved to be very unsatisfactory for various reasons, such as the need to fasten the camera and other mechanical parts in the payload box, but mainly due to the unreliability of the “take picture” command sent from the ground which rarely resulted in a picture being taken.

Another reason for choosing a digital camera was that it made the triggering by electronic means much “safer”. Apart from their ability to communicate with PCs for downloading pictures, many digital cameras have extra commands that allow them to execute extra, sometimes “hidden”, functions. One such command is usually the “take picture” command. Upon receiving this command the camera will be triggered to take a picture. Therefore the next, and the final, approach was to trigger the camera electronically.

While triggering the camera automatically over specific coordinates (waypoints), it was considered a good idea to keep the capability to trigger the camera upon request from the ground. The problem that arose now was that when the UAV returned from its mission there was no way of linking the pictures to the programmed waypoints, since sometimes the triggering was done “manually” from the ground. It therefore seemed a good idea to add an external memory device that could save the coordinates of the locations over which the camera was triggered. A button was then added, than when pressed, would upload this information from the memory to a PC through the same communication port that the camera was connected to during flight.

3.1. Details of Camera Trigger Unit

The physical contact used by digital cameras is usually a COM or a USB port. Because just about every 8 bit microcontroller has a UART incorporated, communication with the RS-232 standard is relatively easy. Since the RS-232 standard is used almost exclusively by COM ports, the first requirement was to use a digital camera with such a contact.

3.1.1. Camera

The camera model DC210 by Kodak [Figure 9] was chosen for this task. The next step was to try and understand the protocol used by the camera and its software running on the PC. In other words the set of rules and commands that govern the communication between the PC and camera had to be partially understood, so that they could be simulated by the microcontroller. This is normally a trivial step, since manufacturers rarely release such information to the public. Fortunately a few websites on the internet [3] revealed information on this subject. The camera did in fact have a “take picture” command, and it consisted of a sequence of 8 bytes sent to the camera at the default speed of 9600 bps.

<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Take Picture</td>
<td>7C 00 00 00 00 00 00 1A</td>
</tr>
<tr>
<td>Status Request</td>
<td>7F 00 00 00 00 00 00 1A</td>
</tr>
</tbody>
</table>

Table 2. Two of the commands used to control the camera

It is important to understand that this command only works with cameras of this manufacturer (Kodak). For this unit to work with cameras from other manufacturers,
the command(s) used to trigger the particular camera have to be known and reprogrammed into the microcontroller.

If the camera receives this command while it is not in the busy state it will execute it. A picture will be snapped and confirm that it has received the command correctly by sending the single byte character \(0xD1\), which functions as an acknowledgement. On an older version of the code this acknowledgement was used by the microcontroller to confirm that a picture had actually been taken.

### 3.1.2. External EEPROM IC

For saving picture information, such as picture number, time and coordinates a memory device had to be used. The EEPROM device M95256 manufactured by SGS Thompson was chosen. It has a storage capacity of 32 kB. This device also has a special writing mode that allows up to 64 bytes to be written to the device at once, without additional control characters. This simplified programming a bit, reduced write-time and therefore this mode was chosen. Each time a picture is taken 64 additional bytes are saved to the EEPROM.

The AT90S2313 microcontroller would have been perfect for this task, since it has the right amount of program memory and supports UART. But unfortunately it does not support SPI, which is required for communicating with the EEPROM memory. The SPI routines could have been programmed into this device, but instead the more powerful AT90S8515 microcontroller was chosen which already supports this mode.

After each mission it is time to upload the picture information to a PC. If a PC is not available immediately after landing the power can simply be disconnected from the camera trigger unit without any risk of data loss. This thanks to the EEPROM which has a data retention capacity of 40 years. Uploading can be done at any suitable occasion. Please note that a null-modem serial cable has to be used.

### 3.2. Camera Trigger Program Flow

The program starts with an initialization of internal devices, such as UART, SPI, and timers. It also initializes the only external device, the EEPROM. After initialization the program falls in an endless while loop.

Within this loop the “send to COM” button and the camera trigger state are constantly checked. Basically the program is interrupt-driven, which means that the majority of the code gets executed only when an interrupts occurs. There are three interrupts that disrupt normal program execution, and all of them are highly active.

#### 3.2.1. ICP Interrupt

This interrupt is of vital importance for the manual camera triggering option. The simplest way to trigger the camera from the ground is to use a free servo channel from the receiver aboard the UAV.

<table>
<thead>
<tr>
<th>Vector Number</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reset</td>
</tr>
<tr>
<td>4</td>
<td>Timer Capture (ICP)</td>
</tr>
<tr>
<td>8</td>
<td>Timer0 OverFlow</td>
</tr>
<tr>
<td>10</td>
<td>UART Rx Complete</td>
</tr>
</tbody>
</table>
Table 3. Four out of 15 interrupt vectors with the corresponding interrupt source used by the AT90S8515

Out of the three interrupts used the ICP has the highest priority (see Table 1). Its job is to constantly monitor activity on the ICP pin from an external signal source. The signal source in this case is the receiver’s servo channel output assigned for camera triggering.

The first time, this pin is configured to trigger on the positive edge of an incoming signal. If an interrupt is not already being executed, then one will be generated that will disrupt all normal program flow. Within this interrupt the edge-select trigger is changed from positive to negative, and the 16 bit timer used to measure the pulse period is cleared. The program then resumes normal execution at the same time as the timer increments on every clock cycle. The next ICP interrupt will occur within one or two ms, at the end of the pulse. When this happens the current value of the register is saved to a variable for later comparison to a the variable “relaytrigger”, to deduct if a camera trigger request has being received.

3.2.2. Timer Interrupt

The digital camera used, has a minimum time interval between which pictures can be taken, mainly due to the fact that digital cameras have to store the image in some volatile memory, and this is usually a time consuming task. After a picture has being taken the microcontroller’s timer will start to countdown and a new “take picture” command will not be sent until the timer allows that. This camera in particular takes about seven seconds to store a picture. During this time the camera is in a busy mode, and ignores every command sent to it from the COM port.

3.2.3. UART Interrupt

The UART interrupt contains one big part of the total code of the program. This interrupt gets fired every time a character is received by the microcontroller. The GPS sends roughly 200 characters every second. Received characters are stored in strings (string is an array of chars with a terminating null character) and the code filters out only specific parts of the GPS’s NMEA transmission. For the camera trigger unit the two NMEA sentences of interest are GGA and APB.

The APB sentence (Autopilot Sentence “B”)
The information extracted from this sentence tells the microcontroller if the UAV has entered the “arrival circle”.

```
$GPAPB,A,A,0.0,L,N,A,,357.3,M,SIM002,347.8,M,,,*6A
```

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Status</td>
</tr>
<tr>
<td>2</td>
<td>Status</td>
</tr>
<tr>
<td>3</td>
<td>Magnitude of XTE</td>
</tr>
<tr>
<td>4</td>
<td>Direction to Steer (Left/Right)</td>
</tr>
<tr>
<td>5</td>
<td>XTE units (N = Nautical Miles)</td>
</tr>
<tr>
<td>6</td>
<td>A = Arrival Circle Entered</td>
</tr>
<tr>
<td>7</td>
<td>A = Perpendicular Passed at Waypoint</td>
</tr>
<tr>
<td>8-9</td>
<td>Bearing origin to destination</td>
</tr>
<tr>
<td>10</td>
<td>Destination waypoint ID</td>
</tr>
</tbody>
</table>
This arrival circle has a radius of 50 m from the centre of the waypoint. When the GPS is within this imaginary circle it sets the character A in the 6th field of the APB sentence. The field filtered by the microcontroller is in this case only a character. When the APB sentence is detected the microcontroller awaits character number 20. It might seem unsafe to extract information this way, and in fact it is. But for this particular sentence, up to this field the number of characters remains always the same. The maximum value of the XTE field for example is 9.9.

Since some fields are variable width, or may be omitted, the receiver should locate the desired data fields by counting commas, rather than by character position within the sentence. Code for such a function has been written [Appendix 2 – NMEA Field Extracting Code] but not used with the microcontroller.

If the UAV has entered the “arrival circle” then a “camera trigger” request is sent. If the timer allows a new picture to be taken the necessary command byte will be sent to the camera’s COM port.

3.2.4. Saving picture information to the EEPROM
Immediately after a picture has been taken, the (getcoordinate) flag is set, that will cause the first received GGA sentence to be saved on the EEPROM. In fact only the first five fields of the GGA sentence are saved.

The GGA sentence (GPS Fix Data)
Information extracted from this sentence reveal the exact location of the GPS receiver.

```
$GPGGA,124706.59,5923.8146,N,01803.3650,E,1,04,2.0,-0026,M,,,,*38
```

<table>
<thead>
<tr>
<th>1</th>
<th>UTC of Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-3</td>
<td>Latitude – (North/South)</td>
</tr>
<tr>
<td>4-5</td>
<td>Longitude – (East/West)</td>
</tr>
<tr>
<td>6</td>
<td>GPS Quality Indicator</td>
</tr>
<tr>
<td>7</td>
<td>Number of satellites in use</td>
</tr>
<tr>
<td>8</td>
<td>Horizontal dilution of precision</td>
</tr>
<tr>
<td>9</td>
<td>Antenna altitude above/below MSL</td>
</tr>
<tr>
<td>10</td>
<td>Units of antenna altitude</td>
</tr>
<tr>
<td>11</td>
<td>Geoidal Separation</td>
</tr>
<tr>
<td>12</td>
<td>Units of geoidal separation (M = Meters)</td>
</tr>
<tr>
<td>13</td>
<td>Age of Differential GPS Data</td>
</tr>
<tr>
<td>14</td>
<td>Differential reference station ID</td>
</tr>
</tbody>
</table>

Table 5. The GGA sentence contains information about the present position of the GPS unit.

The data fields saved on the EEPROM contain: the sentence ID (always the same), the exact time, and the exact coordinates over which the picture was taken. When this
data has been saved the `getcoordinate` flag is reset and the microcontroller resumes normal program execution. 

Each time a picture is taken 64 additional bytes are saved to the EEPROM. With 32kB, 512 data picture-information blocks can be saved.

![Image: screenshot from HyperTerminal, showing the pic-info uploaded from EEPROM to PC, by the press of a button. The red boxes indicate the fields of interest, and their content.]

### 3.2.5. Uploading picture information to a PC

The picture information saved in the EEPROM since the camera trigger device was powered can be sent to a PC at any time. Just disconnect the camera and connect a null-modem cable to a PC running a COM port monitoring program, such as *Hyperterminal*. The data is sent at 9600 bps, with no parity and one stopbit. In the left you will see a number which corresponds to the picture number in the camera. This assumes that the camera has been cleared from all its pictures before each flight.

Future improvements of this program could utilize the picture number information directly from the cameras reply on the “get status” command. A program that establishes a real two way communication between the camera and PC was in fact written. But it was later removed from the code, due to the difficulty of connecting both the GPS and camera to the microcontroller’s single UART device.
Figure 11. Pictures over a small airfield taken from an experimental UAV.
4. Waypoint Direction Finder Unit

This unit attempts to steer a UAV towards a pre-programmed waypoint or along a route of waypoints. The program filters the NMEA sentences of interest and generates the required PWM signal. This signal controls a servo which in turn controls the aileron of the UAV.

The coordinates of interest are programmed in the GPS as sequential waypoints using the "route" function. The steering of the aircraft is not handled in terms of absolute coordinates, but in terms of current heading and bearing. When the heading is equal to the bearing then aircraft is heading straight towards the waypoint.

The microcontroller’s task is twofold:

- filter out the NMEA sentences of interest sent by the GPS.
- use this information to generate a square wave (PWM) for a servo.

The microcontroller listens out for the RMB and the RMC sentences sent by the GPS receiver once every second. The APB sentence could very probably also be used, because it contains the required information into a single sentence, thus making filtering a bit simpler.

The RMB sentence (Generic Navigation Information)

The information extracted from this sentence includes direction to steer and bearing.

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>$GPRMB,A,0.00,L,SIM001,SIM002,5923.8461,N,01803.3590,E,00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0,350.,021.7,A*13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6. Information extracted from the RMB sentence includes direction to steer and bearing.

The RMC sentence (Transit Specific)

The only information extracted from this sentence is bearing.

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$GPRMC,124705.19,A,5923.8061,N,01803.3652,E,21.7,357.2,07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1002,03.,E*6F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| 1 | Time, UTC of position fix |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Status</td>
</tr>
<tr>
<td>3-4</td>
<td>Latitude (N/S)</td>
</tr>
<tr>
<td>5-6</td>
<td>Longitude (E/W)</td>
</tr>
<tr>
<td>7</td>
<td>Speed over ground (knots)</td>
</tr>
<tr>
<td>8</td>
<td>Course over ground (degrees)</td>
</tr>
<tr>
<td>9</td>
<td>Date (ddmmyy)</td>
</tr>
<tr>
<td>10</td>
<td>Magnetic Variation (degrees)</td>
</tr>
<tr>
<td>11</td>
<td>Magnetic Variation (sense, E/W)</td>
</tr>
<tr>
<td>12</td>
<td>Checksum</td>
</tr>
</tbody>
</table>

Table 7. Format of an RMC sentence.

### 4.1. Waypoint Direction Finder Program Flow

Every second the GPS sends “bearing”, “heading” and “direction to steer” information through the COM port as NMEA sentences. The microcontroller receives these sentences and filters out only the required fields. These information fields are sent as parameters to the “degreestowpt” function which calculates the difference in degrees between heading and bearing. The function then returns the result which is used to judge how much the servo lever has to be deviated from its central position in order to change the current heading of the UAV so that it matches the bearing. Remember that when heading and bearing are equal, the aircraft is on the right path to its destination. The direction to steer is of course used to decide in which direction to move the servo lever.

The PWM signal generated on one of the microcontroller’s pins is linearly proportional to the deviation from the waypoint. A deviation of more than 90 degrees, between heading and bearing, results in a full scale deviation of the servo lever.

### 4.2. Future improvements

In the present design the (PWM) pulse is generated purely by software. Instead of generating the PWM within the main’s while-loop, the microcontroller’s built in PWM support could be used. This will reduce the microcontroller’s load substantially, as well as increase the accuracy of the pulse.
5. Altimeter / Altitude Hold Unit

This section describes how to construct an altimeter and explains how it could be modified for stabilizing a UAV to a certain altitude. The theoretical accuracy of the altimeter is calculated, and suggestions for increasing it are provided.

To illustrate the effects of pressure consider a diver, diving into the sea. As the depth increases, the pressure exerted on the diver’s body increases. Similarly our atmosphere could be compared to a gigantic sea of air, with a person standing at the sea bed. The sea bed in this example represents the earth. Air pressure is caused by the weight of air pressing down on the earth. Earth’s gravity, of course, causes the downward force that we know as “weight”. Since the pressure depends on the amount of air above the point where you are measuring the pressure, the pressure falls as you go higher. Atmospheric pressure is inversely proportional to altitude. As altitude increases pressure decreases. The air pressure changes also with the weather. For a given location during the course of the day the air pressure may change considerably depending on the weather. An altimeter that measures altitude by measuring pressure, will therefore need to be calibrated in regular intervals.

This altimeter uses atmospheric pressure to calculate the altitude and it can be calibrated to correct for pressure variations. The interface to the user is three buttons and an LCD. The buttons are used for calibration purposes. Two of the buttons allow the sea level (reference) pressure to be increased, or decreased by steps of one hPa. When the power is disconnected this calibration is lost, unless the third “save” button is pressed. This button saves the reference pressure shown on the LCD to the EEPROM. The next time the unit is powered; the last saved sea-level pressure is retrieved from the EEPROM memory, displayed on the LCD and used for altitude calculations. For measuring altitude a single calibration is not enough. Just like airplane pilots, the latest sea level pressure for the area will have to be obtained and fed into the altimeter. This information for virtually any location can be obtained from the local weather station, or from web-based weather services. (See Internet Links section) The atmospheric pressure can vary quite a bit during a 24 hour period, and if it is not kept up to date in the altimeter, the reading will be false.

At sea level the atmospheric pressure is on average 101.3 kPa. This value is not constant, it can vary by approximately ±5% between so-called high and low pressure systems.

5.1. Pressure Sensor

The pressure sensor used is the MPX5100 manufactured by Motorola. [Figure 13] It can sense pressures from 115kPa to 15kPa. The pressure can be measured indirectly at the output of the sensor as an analogue voltage which varies between 4.7V to 0.2 V. The linear equation which relates pressure to voltage is:

\[ V_{out} = 5(P - 0.009 - 0.095) \]  \[ 5.1 \]

Since the microcontroller cannot work with analogue levels, an AD converter has to be used.
5.2. Analogue to Digital Converter

The ADC used is the MAX187 manufactured by Maxim. It is a 12-bit successive approximation converter, used to convert an analogue voltage from the pressure sensor to a 12 bit serial stream. It is designed to interface and communicate with a microcontroller, and is therefore suitable for this application. The result of each conversion is sent upon request to the microcontroller through an SPI interface. The picture below shows the serial data stream going from the ADC to the microcontroller in more detail. The 12 bit conversion result is read from the ADC, two bytes at the time. The data is then processed as shown below by shifting various bits and storing the final result in the variable “adcdig”.

![Figure 14. Voltage Output from MPX5100 as a function of pressure](image)
Table 8. Details of the SPI communication between ADC and microcontroller.

5.3. **Altimeter Program Flow**

This program is also interrupt driven, but with a single interrupt. Most of the time it is in an idle state, waiting for the timer interrupts to be fired. When an interrupt occurs it jumps to the timer interrupt handling routine. A sampling request is sent to the ADC through the SPI, which sends the result of the conversion back to the microcontroller through the same interface. This sample is then added to a variable. Subsequent pressure samples are all added to the same variable. When fifteen samples have been acquired the program flow is redirected to a part of the code that divides this variable by fifteen. The LCD then updates the local pressure and altitude. We now have the average value of fifteen samples all taken with evenly spaced intervals from each other.

5.4. **Altitude from Pressure**

To keep the UAV steady on a certain altitude, knowing the actual altitude, in meters, above the ground (or sea level) is not necessary. Pressure measurement is less trivial than altitude measurement. Therefore since altitude and pressure are related, it is enough to try to stabilize the UAV to a certain pressure. Despite this, below an explanation is given of how this conversion could be done.

The relationship between pressure and the voltage from the sensor output is linear. This makes the sensor a very accurate device for measuring atmospheric pressure over a vast altitude range. Unfortunately the relationship between altitude and pressure is not linear, although at very low altitudes (below 1 km) it could be considered as such. For converting from local pressure (referred to as QFE in aeronautics) and sea level pressure (QNH) to altitude the following formula is used. Since the ratio of the two pressures is taken, any unit can be used.

\[
\text{altitude}(m) = 43634.1 \times \left( \frac{\text{sealevel}_{\text{qnh}}}{\text{local}_{\text{pressure}}} \right)^{0.190255} - 1
\]

**Equation 1.** Converts sea level pressure and local pressure to an altitude
In C code this formula looks like this:

```c
43634.1*(pow((qnh/pressure),0.190255)-1);
```

The “`float pow(float x, float y)`” math function, returns `x` raised to the power of `y`. For the AT90S8515, the formula above, when compiled into machine language, results in a 30% increase of program memory.

### 5.5. Accuracy

The $\frac{1}{2}$ km closest to the earth the pressure drops by 1 hpa for every increase in altitude by eight meters. From the pressure sensor’s perspective a 1 hpa pressure variation corresponds to a 4.5mV change. This in turn means that a one meter change in altitude causes an equivalent change in output voltage of roughly 0.6 mV. $4.5/8 = 0.5625 \text{ mV/m}$.

If the ADC is set to convert voltages between zero and five volts: $5/0.006 = 8333$ levels would be required to be distinguished by an ADC if one meter accuracy is required. Since a 13bit ADC can distinguish between $2^{13} = 8192$ levels, a 13 bit ADC is required if resolution of 1 meter is desired for the specified parameters.

<table>
<thead>
<tr>
<th>ADC bit resolution</th>
<th>Theoretical accuracy in meters</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>34</td>
</tr>
<tr>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>8.7</td>
</tr>
<tr>
<td>11</td>
<td>4.3</td>
</tr>
<tr>
<td>12</td>
<td>2.17</td>
</tr>
<tr>
<td>13</td>
<td>1.08</td>
</tr>
<tr>
<td>14</td>
<td>0.5</td>
</tr>
<tr>
<td>15</td>
<td>0.27</td>
</tr>
<tr>
<td>16</td>
<td>0.14</td>
</tr>
</tbody>
</table>

Table 9. Accuracy in meters for different ADCs, considering ADC voltage input varies between zero and five Volts, and pressure sensor sensitivity is 4.5 mV per hPa.

Accuracy can be increased in a number of ways. By looking at the table [Table 8] above, it would be easy to think that this can be done simply by substituting the 12-bit ADC with a 14 or 16 bit ADC. Although simple, this approach is not recommended, not only because ADCs of such resolution are very expensive, but because there is something that is fundamentally wrong with this setup, which doesn’t make full use of the resolution provided by the ADC.

### 5.6. Signal Conditioning

The problem is that for low level altitude measurements a very large part of the ADC’s analogue conversion range (0 to 5V) is never used. A definitely cheaper approach would be to keep the existing ADC and to use some analogue signal conditioning. The goal is to make full use of the 0-5 Volt span provided by the ADC. Signal conditioning consisting of basic amplifier circuits could be developed to enhance the part of the span that is mostly used, and “removing” part of the span that
is rarely or never used. This will mean that the altimeter would not be able to measure extremely low/high altitudes. But simultaneously the accuracy of the measured altitude within the new reduced altitude range would increase dramatically.

Unfortunately the construction of such a signal conditioning circuit is not very straightforward, due to the nature of analogue electronics. The circuit has to be simulated repeated times. It must then be built with high precision components, and finally verified with physical tests.

Figure 15. Theoretical explanation of how the accuracy could be increased with signal conditioning.

5.7. Future Improvements

When the difficult task of acquiring the altitude (or pressure) is accomplished, it is relatively easy to use this information to stabilize the UAV to a predefined altitude. A servo arrangement similar to the one used with the “Waypoint Direction Finder Unit” could be used. This time instead of controlling the ailerons, the elevator on the tail of the UAV would have to be controlled. The microcontroller would have to repeatedly compare the current altitude to the programmed altitude in a feedback loop, and from there deduct if it has to ascend or descend to reach the desired pre-programmed altitude. Note that in a real aircraft, or UAV, all aircraft dynamics involved have to be taken into consideration.

Figure 16. Basic description of altimeter altitude hold, feedback loop arrangement.
D. Conclusion.

The camera triggering unit meets all requirements, as it offers not only manual, but also an advanced automatic triggering option. An additional feature is the capability of saving the coordinates of locations photographed by the camera.

The waypoint direction finder unit also meets the requirements. The datastream sent from the GPS is successfully filtered. Extracted information generates a signal which points the servo lever correctly towards the current waypoint.

The altimeter successfully determines the altitude. The requirements were fully met though, as it does not (at present time) provide a way of stabilizing the aircraft to a certain altitude. However a description of how this could be accomplished is given.

This project proved to be very educative, because it covered many areas of modern electronic design. It did not deal strictly with software or hardware, but a combination of both. It was interesting follow the evolution of an electronic device in all its phases, from an idea to an actual product.

With modern electronic products becoming more and more digitalized, software development for microcontrollers is becoming an important aspect of the electronics of today and tomorrow.
### E. Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electronically Erasable Programmable Read Only Memory</td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
</tr>
<tr>
<td>ICP</td>
<td>Input Capture Pin</td>
</tr>
<tr>
<td>LCD</td>
<td>Liquid Crystal Display</td>
</tr>
<tr>
<td>MIPS</td>
<td>Million Instructions Per Second</td>
</tr>
<tr>
<td>NMEA</td>
<td>National Marine Electronics Association</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulation</td>
</tr>
<tr>
<td>QFE</td>
<td>Local pressure</td>
</tr>
<tr>
<td>QNH</td>
<td>Sea level pressure</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>SPI</td>
<td>Synchronous Peripheral Interface</td>
</tr>
<tr>
<td>UART</td>
<td>Universal Asynchronous Receiver Transmitter</td>
</tr>
<tr>
<td>UAV</td>
<td>Unmanned Aerial Vehicle</td>
</tr>
<tr>
<td>XTE</td>
<td>Cross Track Error</td>
</tr>
</tbody>
</table>
F. Sources

6. Digital Altimeter <http://www.qsl.net/ok2xdx/Altimeter/altimeter.html>

G. Internet Links

Micropilot
http://www.micropilot.com

RISC Architecture
http://cse.stanford.edu/class/sophomore-college/projects-00/risc/

National Weather Service, Internet Weather Source
http://weather.noaa.gov/index.html

Microcontrollers Overview
http://www.ami.bolton.ac.uk/courseware/micros/overview.html

CodeVision AVR – C-compiler for AVR Microcontrollers
http://www.hpinfotech.ro/

Digital Photography by Radio Control
http://www.rc-soar.com/tech/dwaerialphoto.htm

Weather Basics
http://www.usatoday.com/weather/tg/whighlow/whighlow.htm

Ricoh RDC-2E Serial Line Protocol (not Kodak Protocol)
http://www.netspace.net.au/~bmiller/linux/rdc2e/protocol.html

Altimeter
http://home.t-online.de/home/Matthias.Stehr/hohe.htm

H. Product Sheets

The latest product sheets for the microcontrollers used in this project can be obtained from Atmel.

Atmel 8 bit RISC AVR Microcontrollers
http://www.atmel.com/atmel/products/prod23.htm
I. Appendix (Program Flow Charts)

   a. Camera Trigger Unit
      Main
      ICP & Timer Interrupt
      UART Interrupt

   b. Waypoint Direction Finder Unit
      Main
      UART Interrupt

   c. Altimeter / Altitude Hold Unit
      Main
      Timer Interrupt
Camera Trigger Unit

Main

- Initialize Variables

Start

ICP timer1 content > pulstrigger?

"Send to PC" button pressed?

send "take picture" command to COM port

send picture information to COM port

Yes

No
Camera Trigger Unit
ICP Interrupt

ICP Interrupt

falling edge?

Yes

• save timer1 contents
• change trigger to rising edge

No

• clear timer1 contents
• change trigger to falling edge

Exit Interrupt

Timer0 Interrupt

Timer0 Interrupt

wait = 0 ?

Yes

No

decrement wait

Exit Interrupt
Camera Trigger Unit
UART Interrupt

character valid? Yes

getcoordinate = y? Yes

GGA sentence? Yes

APB sentence? Yes

42 characters ready? Yes

save pic-info to EEPROM

set ICP timer1 contents > relaytrigger ("automatic take picture command")

Exit Interrupt

character 20 = "A"? No

No

No
Waypoint Direction Finder Unit

Main

- Initialize Variables
- Start
- *pulsewidth* function
  - direction of servo displacement
  - amount of servo displacement
Waypoint Direction Finder Unit

UART Interrupt

character valid?

character = $ ?

Yes

have 34 characters arrived?

No

Exit Interrupt

NMEA sentence = $xxxxMx ?

No

No

NMEA sentence = $xxxxB ?

Yes

sentence = $GPRMB

extract direction to steer

degreestowpt function

extract bearing

direction of servo displacement

sentence = $GPRMC

extract heading

amount of servo displacement
Altitude Meter

Main

Initialize Variables

Start

showqnh function

Retrieve last QNH from EEPROM and show on LCD

QNH- Button pressed?

Yes ➔ decrement QNH

No ➔ QNH+ Button pressed?

Yes ➔ increment QNH

No ➔ Save QNH Button pressed?

Yes ➔ saveqnh function

No ➔ No showqnh function

Retrieve last QNH from EEPROM and show on LCD

Initialize Variables

Start

showqnh function

Retrieve last QNH from EEPROM and show on LCD

QNH- Button pressed?

Yes ➔ decrement QNH

No ➔ QNH+ Button pressed?

Yes ➔ increment QNH

No ➔ Save QNH Button pressed?

Yes ➔ saveqnh function

No ➔ No showqnh function

Retrieve last QNH from EEPROM and show on LCD

Initialize Variables

Start

showqnh function

Retrieve last QNH from EEPROM and show on LCD

QNH- Button pressed?

Yes ➔ decrement QNH

No ➔ QNH+ Button pressed?

Yes ➔ increment QNH

No ➔ Save QNH Button pressed?

Yes ➔ saveqnh function

No ➔ No showqnh function

Retrieve last QNH from EEPROM and show on LCD

Initialize Variables

Start

showqnh function

Retrieve last QNH from EEPROM and show on LCD

QNH- Button pressed?

Yes ➔ decrement QNH

No ➔ QNH+ Button pressed?

Yes ➔ increment QNH

No ➔ Save QNH Button pressed?

Yes ➔ saveqnh function

No ➔ No showqnh function

Retrieve last QNH from EEPROM and show on LCD
Altimeter / Altitude Hold Unit

Timer Interrupt

timervar = 255 ?

Yes

Show pressure on LCD

adcdigmean

15

error compensation

No

save instantaneous pressure to var: "adcdigmean"

Exit Interrupt

timervar = 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240

Yes

Show altitude on LCD

\[
\text{altitude}(m) = 43634.1 \times \left( \frac{\text{sealevel}_{\text{sub}}}{\text{local}_{\text{pressure}}} \right)^{0.190255} - 1
\]
J. Appendix (C-Code)

a. Altitude Hold

/* Altimeter v1.0
CPU = AT90S8515
XTAL = 8MHz
PROG MEM USE = 62% (?%)
PERIPHERALS
1. 12-bit ADC (MAX187)
2. pressure sensor (MPX5100AP)
NOTES: Samples from ADC 15 times with equal intervals during a 2 second period
and then updates the LCD.
IMPORTANT: use a small capacitor on pressure sensor outputs
AUTHOR = Alex Skafidas
UPDATE = 4 October 2002
PINS:
3 CPU - 1K res to gnd + switch
4 CPU - 1 EEPROM (spi CS)
5 CPU - 7 ADC (spi CS)
6 CPU - 5 EEPROM (spi MOSI)
7 CPU - 6 ADC - 2 EEPROM (spi MISO)
8 CPU - 8 ADC - 6 EEPROM (spi clock)
10 CPU - Vcc
15 CPU - 6 LCD
16 CPU - 4 LCD
20 CPU - Gnd
40 CPU - Vcc
4 EEPROM - Gnd
8 EEPROM - Vcc
7 EEPROM - Vcc
*/

#include <io8515v.h>
#include <macros.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <eeprom.h>

#pragma interrupt_handler timer_interrupt:8

#define RW 4         // not used - connect to gnd
#define E 5
#define RS 6
#define CS_ADC 4     // PB
#define CS_EEP 3
#define REFRESHRATE 0x04

/* Variables */
char ready =1;
char eepromdata;
char bytehi;
char bytelo;
char timervar;
char adcdigalpha[8];
char eepromaddress = 0;
unsigned int qnh;
signed int calc;
unsigned int adcdig;
unsigned int adcdigmean;
unsigned int pressure;

/* Prototypes */
void LCD_init(void);
void LCD_clear(void);
void Delay(int times);
void LCD_putstr(char * str);
void LCD_goto(char y, char x);
void LCD_putch(char rs, char rw, char data);
void SpiInit(void);
void port_init(void);
void init_devices(void);
char Spi.ReadByte(char spitx);
void showqnh(void);
void saveqnh(void);
void updatelcd(void);

void main(void)
{
    init_devices();
    // Formatted for 4x20 LCD
    LCD_putstr("pressure hPa altitude m");
    LCD_goto(2,1);
    LCD_putstr("QNH:");
    showqnh();
    SEI();
    while (1)
    {
        if (((PIND & BIT(2)) == BIT(2) && ready ==1))
        {
            ready = 0;
            qnh = qnh + 10;
            updatelcd();
        }
        else if (((PIND & BIT(3)) == BIT(3) && ready ==1))
        {
            ready = 0;
            qnh = qnh -10;
            updatelcd();
        }
        else if (((PIND & BIT(4)) == BIT(4) && ready ==1))
        {
            TCCR0 |= 0x00;
            CLI();
            EEPROM_WRITE(0x1,qnh);
            TCCR0 |= REFRESHRATE;
            SEI();
        }
    }
}

void updatelcd(void)
{
    itoa(adcdigalpha,qnh,10);
adcdigalpha[4] = \0';
    LCD_goto(2,6);
    LCD_putstr(adcdigalpha);
}

void showqnh(void)
{
    EEPROM_READ(0x1,qnh);
    if ((qnh < 10000) || (qnh > 11500))
    {
        qnh = 10350;
        EEPROM_WRITE(0x1,qnh);
    }
void timer_interrupt (void)
{
    TCCR0 |= 0x00;
    timervar++;
    if (timervar == 255)
    {
        char decim;
        char i;

        // pressure from ADC reading
        adcdigmean = adcdigmean / 15;
        pressure = (((adcdigmean/4095.0)+0.095)/0.009)*100;
        pressure = pressure - 141;
        itoa(adcdigalpha,pressure,10);

        // format pressure, with decimal point
        decim = adcdigalpha[4];
        adcdigalpha[4] = '\0';     // Bug if pressure is below 1000 hpa ?
        LCD_goto(1,10);
        LCD_putstr(adcdigalpha);
        LCD_putch(1,0,'.);
        LCD_putch(1,0,decim);

        // pressure to altitude
        calc = abs((43634.1*(pow(((qnh+1.0)/pressure),0.190255)-1)));
        LCD_goto(3,10);
        // Set sign for altitude
        if (qnh < pressure)
            LCD_putch(1,0,'-');
        else
            LCD_putch(1,0,'+');
        itoa(adcdigalpha,calc,10);
        LCD_putstr(adcdigalpha);
        LCD_putstr("    ");
        adcdigmean = 0;
        timervar = 0;
        ready = 1; // enable buttons
    }
    else if (((timervar & 0x0F) == 0x0F)
    {
        PORTB &= ~BIT(CS_ADC);    // turn off CS
        bytehi = SpiReadByte(0x00);    // CS should NOT go LOW between these
        bytelo = SpiReadByte(0x00);    // readings
        PORTB |= BIT(CS_ADC);    // turn on CS
        adcdig = ((int)(bytehi << 5) | (bytelo >> 3));
        adcdigmean = adcdigmean + adcdig;
    }
    TCCR0 |= REFRESHRATE;
}

void init_devices (void)
{
    CLI();
    port_init();
    SpiInit();
UCR = 0x00; //disable while setting baud rate
UBRR = 0x33; //set baud rate
UCR = 0x08; //enable

LCD_init();
LCD_clear();
TCCR0 |= REFRESHRATE;
TIMSK |= 0x02; // timer 0 overflow interrupt enable

void port_init(void)
{
  PORTA = 0xFF; // 7654 3210
  DDRA = 0xFF;
  PORTB = 0xFB; // 1111 1011
  DDRB = 0xFB; // 1111 1011
  PORTC = 0xFF;
  DDRC = 0xFF;
  PORTD = 0xE3; // 1110 0011
  DDRD = 0xE3; // 1110 0011
}

void SpiInit(void) // SPI can ONLY use port B
{
  DDRB |= 0xB0; // Set SCK, MOSI & SS as outputs
  PORTB &= 0x5F; // clear bits MOSI, & SCK
  SPCR = 0x53; // + SPE + MSTR; Write to control register
}

/*
** SpiReadByte() first writes a byte (a dummy, since
** that byte is to generate clock signals to "poll" home
** the byte from the slave. The function returns the
** received byte.
*/
char SpiReadByte(char spitx)
{
  SPDR = spitx;
  while (!(SPSR & 0x80));
  return SPDR;
}

For LCD and Delay subroutines see Appendix 2. Common Subroutines (LCD, Delay)
b. Waypoint Direction Finder Unit

/* Waypoint Direction Finder v1.0
CPU = AT90S2313
XTAL = 8MHz
PROG MEM USE = 25% (22%)
NOTES: MAJOR MODIFICATIONS
AUTHOR = Alex Skafidas
UPDATE = 30 July 2002
*/

#include <io8515v.h>
#include <macros.h>
#include <string.h>
#include <stdlib.h>

// always include math.h before calling any floating point functions
// otherwise your program may not work

#pragma interrupt_handler UART_RX_interrupt:10

/* UART Buffer Defines */
#define DOLLAR '$'
#define FALSE 0
#define TRUE 1
#define RS 6
#define E 5
#define RW 4
#define UART_RX_BUFFER_SIZE 33

/* Static Variables */
unsigned int charcount = 0;
char UART_RxHead = 0;
static char UART_RxBuf[UART_RX_BUFFER_SIZE];
char direction;
int answer;
int headint;
int bearint;

/* Prototypes */
void LCD_init(void);
void LCD_clear(void);
void Delay(char times);
void LCD_goto(char y, char x);
void LCD_putchar(char* str);
unsigned char ReceiveByte(void);
void InitUART(unsigned char baudrate);
int degreestowpt(int, int);
void LCD_putstr(char* str);
void LCD_putch(char rs, char rw, char data);
void pulsewidth(int times);

void port_init(void)
{
  PORTA = 0xFF;
  DDRA = 0xFF;
  PORTB = 0xFF;
  DDRB = 0xFF;
  PORTC = 0xFF;
  DDRC = 0xFF;
PORTD = 0x7E;
DDRD = 0x7E;
}

void InitUART(unsigned char baudrate)
{
    unsigned char x;
    UBRR = baudrate; // set the baud rate
    // enable UART receiver, and receive
    interrupt
    UCR = ( (1<<RXCIE) | (1<<RXEN) );
x = 0; // flush receive buffer
    UART_RxHead = x;
}

/* interrupt handler */
void UART_RX_interrupt(void)
{
    unsigned char data;
    unsigned char tmphead;
    char answer2[4];
    static char bearing[4];
    static char heading[4];

    if (((USR & BIT(FE)) == BIT(FE)) ) // if received byte is weird, discard
        return;
    UCR &= ~BIT(RXCIE); // disable all interrupts
    data = UDR; // read the received data
    UART_RxBuf[UART_RxHead] = data; // store received data in buffer
    if ( (UART_RxHead + 1 == UART_RX_BUFFER_SIZE)
    {
        if (UART_RxBuf[4] == 'M') // if received byte is weird, discard
            return;
        // RMB - bearing to waypoint
        else if (UART_RxBuf[5] == 'B') // RMB - bearing to waypoint
            { // LCD_putstr(UART_RxBuf); // print RMB sentence on LCD
                UART_RxBuf[UART_RX_BUFFER_SIZE] = '\0';
                //LCD_putstr(UART_RxBuf[49]);
                //arrived = UART_RxBuf[39];
                //print RMB sentence on LCD
                //LCD_putstr(UART_RxBuf);
                direction = UART_RxBuf[14];
                bearing[3]='\0';
bearint = atoi(bearing);
                UART_RxHead = 0;
tmphead = 0;
                charcount = 0;
            }
    // RMC - heading
    else // RMC - heading
    {
        //LCD_putstr(UART_RxBuf);
        //print RMC
        character at end
        //LCD_putstr(UART_RxBuf);
        //print RMC
strncpy(heading, UART_RxBuf + 16, 3);
//LCD_goto(2,7);
//LCD_putchar(heading); // print heading on LCD
heading[3] = '\0'; // put null on 4th character for atoi function
headint = atoi(heading); // atoi returns 0 on error

answer = degreestowpt(headint, bearint);
//itoa(answer2,answer,10);
//LCD_goto(2,15);
//LCD_putchar(answer2);
//LCD_putchar(1,0,direction);
//LCD_putchar(" ");

UART_RxHead = 0;
tmphead = 0;
charcount = 0;
if ((answer > 50) && ((PIND & BIT(7)) != BIT(7)))
    answer = 50; // pulse 1.00 - 2.00
else if ((answer > 50) && ((PIND & BIT(7))) == BIT(7))
    answer = 25; // pulse 1.25 - 1.75
else
    UART_RxHead = 0;
tmphead = 0;
charcount = 0;
}
else
{
    UART_RxHead = 0;
tmphead = 0;
charcount = 0;
}
else
{
    if (UART_RxBuf[0] == DOLLAR)
    {
        charcount++;
        if (charcount < 16 || charcount > 48)
        {
            tmphead = ( UART_RxHead + 1 );
            UART_RxHead = tmphead;
        }
    }
}

UCR |= BIT(RXCIE); // restore interrupts - cleared when UDR is read?

void main( void )
{
    InitUART( 103 ); // for 4800 baud, UBR=51 for 4MHz, UBR=103 for 8MHz
port_init(); // needed - specify inputs & outputs
LCD_init();
LCD_clear();
LCD_putstr("check com port2");
_SEI(); // enable global interrupts => enable UART interrupts
while ( 1 )
{
    if (direction == 'L')
        pulsewidth( 150 - answer);
    else if (direction == 'R')
        pulsewidth( 150 + answer);
}

void pulsewidth(int times)
{ char a;
 unsigned b;
 int stabilize;
 char answer3[4];
 itoa(answer3,times,10);
 LCD_goto(4,1);
 LCD_putstr(answer3);
 for (stabilize = 0; stabilize < 5; stabilize++)
 {
   UCR &= ~BIT(RXCIE);   //dis UART interrupt
   PORTD |= BIT(PD3);
   for (a=0; a < times; a++)
     for (b=0; b < 5; b++); // 2 for 4MHz and 5 for 8MHz
   PORTD &= ~BIT(PD3);
   UCR |= BIT(RXCIE);     // restore UART interrupts
   Delay(12);
 }

 int degreestowpt (int head, int bear)
 {
 int deg2wpt;
 if ( abs(head - bear) < 180)
   deg2wpt = abs(head - bear);
 else
   // if abs( HDG - BRG) > 180
   {
     if ( head > 180)
       deg2wpt = ((360 - head) + bear);
     else if (head < 180)
       deg2wpt = ((360 - bear) + head);
   }
 return deg2wpt;
 }

 For LCD and Delay subroutines see Appendix 2. Common Subroutines (LCD, Delay)
c. Camera Trigger Unit

/* GPS & Pulse Camera Trigger v1.0
CPU          = AT90S8515
XTAL         = 8MHz
PROG MEM USE = 25% (22%)
PERIPHERALS  1. 32 KB EEPROM (M95256)
NOTES:     MAJOR MODIFICATIONS
AUTHOR    = Alex Skafidas
UPDATE  = 2 November 2002

interrupt vectors are for 8515 ONLY
2313 to 8515
-file header
-int vector numbers
-LCD port - & RS/E PINS !!!
-PORT INIT !!!

count: array = 45 + 5
  char = 7
  int = 3 (*2)
  --------
  63 bytes of RAM ?

*/

#include <io8515v.h>
#include <macros.h>
#include <string.h>
#include <stdlib.h>

#define RW 4       // not used - connect to gnd
#define E 5       // PD
#define RS 6
#define relaytriggerlow  230
#define relaytriggerhigh 238
#define DOLLAR      '$'
#define UART_RX_BUFFER_SIZE 45
#define REFRESHRATE 0x05
#define CS_EEP        3

char UART_RxBuf[UART_RX_BUFFER_SIZE];
char timercountalpha[5];
char ready = 1;
char head;
char drop = 0;
char emptydata;
char picturesleft = 99;
char getcoordinate = 'n';
char countdown = 6;
char timercount1 = 150;
char eepadrlow;
char eepadrhig;
unsigned int wait = 200;
unsigned int eepromadress = 0;

void LCD_init(void);
void LCD_clear(void);
void Delay(int times);
void LCD_goto(char y, char x);
void LCD_putstr(char* str);
unsigned char ReceiveByte(void);
void LCD_putchar(char rs, char rw, char data);
void init_devices(void);
void inputcaptureinit(void);
void icp_interrupt(void);
void timer_overflow(void);
void SpiInit(void);
char SpiReadByte(char spix);

void port_init(void)
{
    PORTA = 0xFF; // 7654 3210
    DDRA = 0xFF;
    PORTB = 0xFB; // 1111 1011
    DDRB = 0xFB; // 1111 1011
    PORTC = 0xFF;
    DDRC = 0xFF;
    PORTD = 0xE7; // 1110 0111
    DDRD = 0xE7; // 1110 0111
}

void timer_overflow(void)
{
    TCCR1B &= ~0x03; // stop timer - dangerous when return is executed
    if (wait == 0)
    {
        PORTD &= ~BIT(PD2);
        TCCR1B |= 0x03; // restore timer
        return;
    }
    else
    {
        wait--;
        timercount1 = 111;
        PORTD |= BIT(PD2);
    }
    TCCR1B |= 0x03; // restore timer
}

void InitUART(unsigned char baudrate)
{
    UBRR = baudrate;
    UCR = 0x98;
}

void inputcaptureinit(void)
{
    TIFR |= (1 << ICF1);
    TCCR1B |= (1 << CS11) | (1 << CS10) | (1 << ICNC1) | (1 << ICES1);
    TIMSK |= (1 << TICIE1);
}

void init_devices(void)
{
    CLI();
    InitUART(51);
    port_init();
    TCCR0 = REFRESHRATE;
    TIMSK |= 0x02; // timer0 overflow int enable
    inputcaptureinit();
    SpiInit();
    LCD_init();
    LCD_clear();
    SEI();
}
void SpiInit(void)
{
    DDRB = 0xFD;
    PORTB = 0x00;
    SPCR = 0x53; // + SPE + MSTR; Write to control register

    PORTB |= BIT(CS_EEP); // high to
    PORTB &= ~BIT(CS_EEP); // low transition for EEPROM
    PORTB |= BIT(CS_EEP);
}

void sendcommand (char command)
{
    putchar(command); // 0x7C = take pic, 0x7F = status request
    putchar(0x00);
    putchar(0x00);
    putchar(0x00);
    putchar(0x00);
    putchar(0x00);
    putchar(0x00);
    putchar(0x1A);
}

void main( void )
{
    init_devices();
    LCD_goto(2,1);
    LCD_putstr("target: ");
    LCD_goto(1,1);
    LCD_putstr("pics: ");
    LCD_goto(1,10);
    LCD_putstr("trig: ");
    while(1)
    {
        if ((PIND & BIT(3)) == BIT(3) )
        {
            char i;
            char j;
            char eepromdata;
            _CLI();
            TCCR1B &= ~0x03;
            TCCR0 = 0x00;
            UCR &= ~BIT(RXCIE); // clear
            TIMSK &= ~BIT(TICIE1);
            PORTB &= ~BIT(CS_EEP); // turn off CS
            SpiReadByte(0x03); // Read Data from Mem Array
            SpiReadByte(0x00);
            SpiReadByte(0x00);
            SpiReadByte(0x00);
            for (j = 1; j < 10; j++)
            {
                putchar(j + 0x30);
                putchar('=');
                for (i = 0; i <= 63; i++)
                {
                    eepromdata = SpiReadByte(0x00);
                    putchar(eepromdata);
                }
                putchar(0xA); // LF - LineFeed
                putchar(0xD); // CR - Carriage Return
            }
            PORTB |= BIT(CS_EEP); // turn on CS
            ready = 0;
        }
    }
}
TCCR1B |= 0x03;
TCCR0 = REFRESHRATE;
TIMSK |= BIT(TICIE1);
UCR |= BIT(RXCIE);
_SEI();
}

if ((timercount1 > relaytriggerlow) && (wait == 0) && (timercount1 < relaytriggerhigh)) //&& (picturesleft != 1)
{
    TCCR1B &= ~0x03;
    TCCR0 = 0x00;
    UCR &= ~BIT(RXCIE);
    TIMSK &= ~BIT(TICIE1);

    sendcommand(0x7C);
    picturesleft--;

    LCD_goto(1,7);
    itoa(timercountalpha,picturesleft,10);
    LCD_putstr(timercountalpha);
    LCD_goto(1,18);
    itoa(timercountalpha,timercount1,10);
    LCD_putstr(timercountalpha);
    LCD_goto(1,16);
    LCD_putch(1,0,' ');
    wait = 300;
    getcoordinate = 'y';
    timercount1 = 144;

    TCCR1B |= 0x03;
    TCCR0 = REFRESHRATE;
    TIMSK |= BIT(TICIE1);
    UCR |= BIT(RXCIE);
}
}

void UART_RX_interrupt( void )
{
    TCCR1B &= ~0x03; // stop timer
    TCCR0 = 0x00;

    if ((USR & BIT(FE)) == BIT(FE) )
        return;

    emptydata = UDR;
    UART_RxBuf[head] = emptydata;

    if (emptydata == DOLLAR)
        head = 0;

    if ((UART_RxBuf[0] == DOLLAR) && (head <= 6))
        head++;
    return;
}

if (getcoordinate == 'n')
{
        head++;
    if (head == 20)
        head++;
    UART_RxBuf[21] = '\0';
if (emptydata == 'A')
{
    LCD_goto(1,16);
    LCD_putch(1,0,emptydata);
    PORTD ^= BIT(PD2);
    timercount1 = 235;
}
else
    timercount1 = 149;
}
else if (getcoordinate == 'y')
{
    char i;
        head++;
    if (head == 42)
        {
            head++;
            UART_RxBuf[42]='\0';
            LCD_putstr(&UART_RxBuf[16]);

            eepadrlow = eepromadress & 0xff;
            eepadrhig = eepromadress >> 8;

            // Save picture info to EEPROM
            PORTB &= ~BIT(CS_EEP);    // turn off CS
            SpiReadByte(0x06);           // Write Enable
            PORTB |= BIT(CS_EEP);    // turn on
            Delay(20);
            PORTB &= ~BIT(CS_EEP);    // turn off CS
            SpiReadByte(0x02);      // Write Data to Mem Array
            SpiReadByte(eepadrhig);
            SpiReadByte(eepadrlow);

            for (i = 0; i <= 42; i++)
            {
                SpiReadByte(UART_RxBuf[i]);    // data to save
                Delay(30);
            }

            for (i=43; i <= 63; i++)
            {
                SpiReadByte('*');
                Delay(30);
            }

            PORTB |= BIT(CS_EEP);    // turn on CS

            // Future Improvement: Add an end marker
            ready = 1;
            getcoordinate = 'n';
            eepromadress = eepromadress + 64;
        }
    TCCR0 = REFRESHRATE;
    TCCR1B |= 0x03; // restore timer
}
char SpiReadByte(char spitx)
{
    SPDR = spitx;
    while (!!(SPSR & 0x80));
    return SPDR;
}
void icp_interrupt(void)
{
    TCCR1B &= ~0x03; // stop timer
    TCCR0 = 0x00;
    if (drop == 1)
    {
        timercount1 = ICR1;
        TCCR1B |= (1 << ICES1);
        drop = 0;
    }
    else
    {
        drop = 1;
        TCNT1 = 0;
        TCCR1B &= ~(1 << ICES1);
    }
    TCCR0 = REFRESHRATE;
    TCCR1B |= 0x03;
}

⚠️ For LCD and Delay subroutines see Appendix 2. Common Subroutines (LCD, Delay)
void Delay(int times)
// with a 4 MHz clock times = [ms]
// shortest delay is Delay(0) which gives aprox. 40 us (used for LCD)
{
    char a;
    unsigned int b;

    if (times > 0)
    {
        for (a = 0; a < times; a++)
        {
            for (b = 1; b < 1454; b++) // every loop takes 5.5cc = 687.5 ns @ 8 MHz
                ;
        }
    }
    else
    {
        for (a = 1; a < 128; a++)
            ;
    }

void LCD_putch(char rs, char rw, char data) // for simple data (1,0,data)
{
    if (rs)
        PORTD |= BIT(RS);
    else
        PORTD &= ~BIT(RS);

    if (rw) // not used: if-else can be removed
        PORTD |= BIT(RW);
    else
        PORTD &= ~BIT(RW);

    PORTC = data; // LCD data is on port C

    PORTD |= BIT(E);
    Delay(0);
    PORTD &= ~BIT(E);
    Delay(0);
}

void LCD_init(void)
{
    Delay(16);
    LCD_putch(0, 0, 0x30); // set to 8-bit interface
    Delay(5); // wait at least 4.1 ms
    LCD_putch(0, 0, 0x30); // set to 8-bit interface
    Delay(1); // wait 1 ms
    LCD_putch(0, 0, 0x30); // set to 8-bit interface
    LCD_putch(0, 0, 0x38); // display off
    LCD_putch(0, 0, 0x01); // display clear
    Delay(16);
    LCD_putch(0, 0, 0x06); // displays on and cursor
delay(0, 0, 0x02); // Turns on display and cursor
}

void LCD_clear(void)
{
    LCD_putch(0, 0, 0x01);
    Delay(16);
}
void LCD_putstr(char * str)
{
    char i = 0;

    while(*(str+i) != '\0')
    {
        LCD_putch(1, 0, *(str+i));
        i++;
    }
}

void LCD_goto(char y, char x)
{
    char DDRAM;

    switch(y)
    {
    case 1 : DDRAM = 0x00 + x - 1; break;
    case 2 : DDRAM = 0x40 + x - 1; break;
    case 3 : DDRAM = 0x14 + x - 1; break;
    case 4 : DDRAM = 0x54 + x - 1; break;
    }
    DDRAM = DDRAM | 0x80;
    LCD_putch(0, 0, DDRAM);
}
e. NMEA field extracting code

**Example.** Recall that NMEA sentences have their information fields separated by commas (the separator). This example explains how to use the program below to extract fields (information between separators) from a string with separators. Since some fields are variable width, or may be omitted, desired data fields should be located by counting commas, rather than by character position within the sentence.

To extract the 2\(^{nd}\), 3\(^{rd}\) and 4\(^{th}\) field of this NMEA sentence:

\[
$GPAPB,A,A,0.0,L,N,A,,357.3,M,SIM002,347.8,M,,,*6A
\]

using the code below, you would have to assign the NMEA sentence the variable `mystring`. You then tell what fields should be extracted. In this case we want fields 2, 3 and 4.

We therefore use the following function:

```c
void stringparse(int selectmin, int selectmax);
```

and request fields 2 to 4.

```c
selectmin = first field to extract
selectmax = last field to extract
```

```c
stringparse(2, 4);
```

The function will parse the given string and will store the extracted fields in another string called `info`. The contents of this string will be:

```c
info[] = ,A,A,0.0,
```

Note: If only one field needs to be extracted, `selectmin` and `selectmax` are both assigned the same “field-locator” number.

**C-code.**
Created and compiled successfully in Borland C++ v5.02

**String to parse:** 
","alpha,bravo,charlie,delta,echo,fox,golf"

**Fields:** 
","--1--,--2--,---3---,--4--,--5--,--6--,--7--"

```c
#include <iostream.h>
#include <string.h>
#include <conio.h>

int character;
int character2;
int selectmin;
int selectmax;
int selectdata;
int commacount;
```
char separator = ',';

char info[30];
char count[]=" 1 2 3 4 5 6 7 ";
char mystring[]=" ,alpha,bravo,charlie,delta,echo,fox,golf,";

void stringparse(int selectmin, int selectmax);

main(void)
{
    while(1)
    {
        clrscr();
        cout << "Function for parsing strings using a variable separator\n\n\n";
        cout << "[x x] - Select Substring\n\n";
        cout << "[8 8] - Change Data Separator\n";
        cout << "[9 9] - Exit\n\n";
        cout << count << "\n\n";
        cout << mystring << "\n";
        cout << "String Length: " << strlen(mystring) << "\n";
        cout << "Scanned Letters: " << character << "\n";
        cout << "Separators Found: " << commacount << "\n";
        cout << "Requested Data: " << info << "\n";
        cout << "Select []" << separator << "[]" << separator << "[]": min (space) max\n";
        character = 0;
        character2 = 0;
        selectdata = 0;
        commacount = 0;
        cin >> selectmin;
        cin >> selectmax;
        switch(selectmin)
        {
            case 9:
                cout << "Have a nice DOS...";
                cin.get();
                cin.get();
                return 0;
            case 8:
                cout << "Enter new data separator to use: ";
                cin >> separator;
                break;
            default:
                stringparse(selectmin,selectmax);
                break;
        }
    }
}

void stringparse(int selectmin, int selectmax)
{
    for (character = 0; character < strlen(mystring); character++)
    {
        if (mystring[character] == separator)
            commacount++;
        if ((commacount > selectmin) && (commacount < selectmax+1))
        {
            info[character2] = mystring[character];
            character2++;
        }
    }
    info[character2] = ',';
    info[character2+1] = '\0';
}
K. Appendix (Schematics)

Camera Trigger Unit

Schematic 1. Camera Trigger

Component List

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL1</td>
<td>Connector</td>
</tr>
<tr>
<td>IC2</td>
<td>Max232</td>
</tr>
<tr>
<td>IC3</td>
<td>ST95256</td>
</tr>
<tr>
<td>IC4</td>
<td>AT90S8515</td>
</tr>
<tr>
<td>X3</td>
<td>COM-port male</td>
</tr>
<tr>
<td>X4</td>
<td>COM-port male</td>
</tr>
<tr>
<td>R1</td>
<td>470 Ω</td>
</tr>
<tr>
<td>R2</td>
<td>10 kΩ</td>
</tr>
<tr>
<td>D1</td>
<td>LED 3mm</td>
</tr>
<tr>
<td>S1</td>
<td>Button (normally open)</td>
</tr>
<tr>
<td>Q3</td>
<td>8 MHz crystal</td>
</tr>
<tr>
<td>C7-C10</td>
<td>10 uF, electrolytic</td>
</tr>
</tbody>
</table>
Waypoint Direction Finder

Schematic 2. Waypoint Direction Finder Unit

Component List

IC1    AT90S2313
Q3     8 MHz crystal
X1     COM-port male
SL1    Connector
C8     10 uF, electrolytic
Altimeter

Schematic 3. Altimeter Unit

Component List

IC4  AT90S8515  
Q3  8 MHz crystal  
     MPX5100AP pressure sensor  
SL2  14 pin LCD Connector  
C8  10 uF, electrolytic  
R2-R4  10 kΩ  
S1-S3  Button (normally open)  
R1  Trim-potentiometer  
IC1  MAX187
L. Appendix (PCB)

PCB 1. Altimeter

PCB 2. Camera Trigger Unit