0 - Introduction

This documentation covers Logomatic Kwan Firmware v1.1, a complete rewrite of the main part of the firmware for the Sparkfun Logomatic v2 hardware.

This code was originally derived from the published Logomatic firmware from Sparkfun (v2.3, retrieved April 2009) but has been modified almost completely. The 'main' folder has been completely restructured, and much of the supporting SD card access routines have been modified, but not beyond recognition. The USB code has been completely removed, as there is no use of the USB during normal operation. The USB mass storage driver and boot loader remain untouched.

New features include:

- Simultaneous recording of both serial ports and ADC data to the same file
- ADC oversampling
- Serial packet formats NMEA (GPS), SiRF (GPS), fixed-length binary or text, and framed binary or text
- ADC written in Text, binary (as with old firmware) as well as NMEA and SiRF protocols to mix with GPS data
- Ability to transmit commands to serial-attached devices
- Automatic baud-rate configuration
- Time-stamps of all analog measurements
- Optional timestamp of all serial packets
- Optional time stamp synchronization using a SiRF or NMEA GPS device attached to a serial port
- Automatic recording of ancillary data about the hardware and firmware
- Ability to measure battery voltage as a ninth ADC channel
- Ability to go into powersave mode to extend battery life

Code improvements include:

- Improved code modularity
- Reduced memory usage
- No dynamic memory allocation
- Improved configuration file parsing
- Tuning to improve code

Target use case – the Rollercoasterometer

"I'm not just President of St. Kwan's, I'm also a client"

--Chris Jeppesen, president of St' Kwan's home for the terminally ADD

I use the Logomatic myself as the core of my real project, the Rollercoasterometer. This uses a Logomatic as a recorder and driver, a GPS receiver, a three-axis accelerometer, and three axes of rotation sensors. The inertial sensors are all analog and use the ADC feature of the Logomatic hardware, while the GPS uses the digital serial input of the Logomatic.

The ultimate goal of all of this hardware is to measure as precisely as possible the track of a roller coaster, including indoor coasters such as Space Mountain. Outdoors, the GPS and inertial sensors back each other up. Indoors, it is up to the inertial sensors.

Since I have a bunch of analog devices and a digital device, I need a way to read both the ADCs and serial port simultaneously, and interleave the data into a single file which preserves the proper timing of each ADC and GPS measurement, while simultaneously keeping the data in an organized manner such that they can be separated in post-processing. The Logomatic hardware is up to the challenge, but the firmware needs some work.

So, while I was in there, I changed some of this, some of that, a bit here, a bit there, reorganized the code, tested the device and found it needed some extra features, thought of other nice features, spotted some unused but useful hardware on the Logomatic, and so on. When I was done, the program had been rewritten almost from the ground up.

1 – Using the new features

Oversampling

The Logomatic can easily sample the analog outputs at 1600Hz or more, while simultaneously reading the UART and doing all of its other work. The problem is that it can't write to the SD card at that rate. So, we use a common technique for digital instruments, oversampling and binning.

Because of the way the ADC hardware works, each sample is just that, a sample of what the incoming analog wave looks like for a brief instant (less than a microsecond). The hardware has no idea what happens to the signal in between samples. Specifically, the ADC reading is *not* the average value of the channel over the time between two samples.

The firmware maintains a number of *bins,* one for each ADC channel it is reading. A bin is just a single number. Each ADC cycle, it reads the ADC channels, then adds the value just read to the bin for that channel. Every *n*th cycle, the contents of the bins are used to create and write an ADC packet.

For instance, suppose the ADC is set to 1000Hz, and the binning is set to 10. Each millisecond, it reads the ADC and adds it to the bin. Each reading may be a number between 0 and 1023, inclusive. Each 10 milliseconds it writes out the ADC bin. Consequently, the reported binned value may be a number between 0 and 10230.

For many purposes, such as inertial measurement, we would really rather have a measurement of the average, rather than just a sample. By oversampling, we get 10 measurements over the time in question, and therefore can average those.

Also, binning represents a noise improvement. The ADC has a noise of at least one data number (DN, 1024 DN in 3.3V). If the noise has certain statistical properties, you can use calculus to prove that the noise of the binned result is the square root of the oversampling rate. In our example, we would expect the noise to be sqrt(10)=about 3 times better than just

sampling at 100Hz. In the case I typically use, I sample at 1600Hz and bin by 16, so I expect a reduction in noise by a factor of 4.

Finally, binning represents a resolution improvement.

To activate binning, first decide the rate at which you want ADC packets, say 100Hz. Next, decide how much you want to bin, say 16 times. Then, multiply the two. Write the product in ADC frequency line, and the binning rate in the ADC binning line. So, in this case it would look like:

```
ADC frequency=1600
ADC binning=16
```

Binning cannot be disabled, but if binning is set to 1, it has the effect of being disabled. Binning cannot be set to zero, or Bad Things will happen. You have been warned.

Commanding

The Logomatic Kwan is able to send data to each serial port, for instance to set a GPS to a certain mode.

To send commands, put them in the file called `COMMAND.txt` which will be created by the Logomatic if it is not present. Command format is in four parts as follows:

```
N 10 1 PSRF103,08,00,01,01
```

First field: Type of command

Type Character	Type
N	NMEA protocol
S	SiRF protocol
T	Text
B	Bare text (no CR/LF at the end)
H	Hexadecimal
;	Comment

NMEA and Text are very similar, as are SiRF and Hexadecimal.

Second field: When to send the command. Commands are sent once per second, so in this case, the command is sent 10 seconds after the Logomatic reaches normal operating mode.

Third field: Serial port to send the command to, 0 or 1. Command is sent at the same baud rate as the port is listening on.

Fourth field: Command data (up to the ending carriage return, ASCII character 0x0A). For a text command, the line end used in the file is sent to the port. For all other modes, the line end just marks the end of the data.

Lines starting with a semicolon are not commands. You can include comments on these kind of lines.

Text mode is simplest. The text specified is sent out to the port specified at the time specified.

Bare text is the same as text, except the line ending is not sent out the serial port. Actually, the last two characters are chopped off, but when the file is saved in MS-DOS mode, these two characters will be CR (0x0D) and LF (0x0A) which are exactly the characters that should not be sent.

Hex mode is useful for when you have to send unprintable ASCII characters. Here, each character to be sent is specified as a pair of hex digits, in the range from 0-9A-F (not case sensitive). You must supply an even number of digits, but you may include spaces wherever you wish, for enhanced human readability. For instance, if you wish to send the bytes 0x10,0x20,0x0A,0x41,0x42,0x43, you must use hex mode, because in text mode, the 0x0A would break the command. You can represent this command as

```
H 5 1 10200A414243
```

Or by using spaces for grouping, as

```
H 5 1 1020 0A 414243
```

The spaces are purely for human readability, and do not affect the data sent.

NMEA mode is almost exactly like Text mode, except that a $ is pre-pended to the command, and a proper checksum is appended. For the SiRF series of GPS devices, the following command will cause the device to reply with the current GGA sentence. To send it, use the command line

```
N 5 1 PSRF103,00,01,00,01
```

which will result in sending to the device on port 1

```
$PSRF103,00,01,00,01*25
```

with the dollar sign pre-pended and the proper checksum automatically calculated and appended.

SiRF mode is similar to binary mode, except it is packaged in a SiRF protocol packet, again with the proper header and footer. To send the command to go from SiRF back to NMEA mode, use the following command line

```
S 10 1 8400
```

which will result in the following data being sent to the device:

```
A0A20002**8400**0084B0B3
```

where you can see the payload in bold and the header and footer in normal. You may use spaces to break up the data, just like with a hex command.

Commands are written to the log file just as if they were received across the serial port. Any given command is always written to the log file before any packets received from the device after the command is sent. In other words, it does exactly what you think it should do.

Auto-baud

Under certain conditions, the Logomatic Kwan is able to automatically measure the baud rate for a serial port. For it to work, the external device attached to the serial port must be set to 8 data bits, 1 stop bit, no parity (This is pretty typical) and must be in one of the standard baud rates: 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, or 115200.

To use Auto-baud, set the baud rate to `Auto` .

Upon start-up, the Logomatic will listen for one second to each the port at each of the standard baud rates. If it gets more than 20 characters and less than 1% of the characters received have errors, it will immediately use that baud rate. Otherwise, it will use the baud rate with the lowest number of errors (not percent).

This method is not reliable. If you know the baud rate, it is preferable to write it explicitly. This way, up to 9 seconds per port are saved during startup, and there is no chance of mistakenly getting the wrong rate.

GPS Sync

The Logomatic can synchronize its clock to UTC as given by a GPS receiver.

To use, hook up a GPS receiver to either of the serial ports. Make sure that the mode of that serial port is set to either NMEA or SiRF, and that the GPS is set to the same protocol.

Now, set the GPS Sync line in LOGCON.TXT.

```
GPS Sync=1
```

This will cause the Logomatic to sync its clock with the GPS every time it gets an interpretable time stamp packet.

The clock is only synchronized to the nearest second. Future versions of Logomatic Kwan will feature synchronization to the PPS output of the GPS, potentially microsecond accurate.

Powersave

The Logomatic can save power in two ways:

- Turn off the blinking lights
- Put the processor to sleep in between ADC or UART reads

Both of these are controlled by the Powersave switch. Powersave is normally 'off' because otherwise there is no outward sign the Logomatic is doing anything at all. To engage Powersave, set the Powersave line in LOGCON.TXT.

```
Powersave=1
```

This will cause the Logomatic to take both power saving measures above. To disengage Powersave, set it to 0 or delete the line entirely.

Note: When the Powersave mode is active, the load indicator in $PKWNL will be inaccurate (all idle time will be charged to which ever task woke the processor up).

2 – Structure of the code

Packets and Circular Buffers

The program has been completely redesigned, and all program activity is now centered around production, transport, and consumption of *packets.*

A *packet* is any sequence of data which should be written to the recording file as a unit, without intervening data from other sources. Examples include a reading of all the ADC channels at a particular instant, or an NMEA sentence from a GPS device.

Previously, the Logomatic captured data from either one UART or from the ADCs, but never both at the same time. Therefore, it was impossible to get data tangled up, and packet handling didn't need to be thought about.

Now, when the ADC and UART are running simultaneously, imagine the following scenario. The UART is in the middle of receiving an NMEA sentence from your attached GPS device, when it is time to read the ADCs. Without careful packet handling, the ADC data may be written right in the middle of the NMEA sentence, ruining both.

To properly handle packets, each packet source (currently the ADC reader and each UART) has one circular dedicated buffer. Each packet source writes to its own buffer, and the packet sources are such that they cannot interfere with themselves. Specifically, each UART handler reads data out of the hardware FIFO for its port, and therefore the data must come out in sequence. Likewise, the ADC reader is interrupt driven, but the timer interrupt it uses is disabled while the handler is running. As a consequence, the ADC reader cannot step on itself either. Each reader is on its own interrupt, but each writes into its own buffer and only its own buffer, and no other task is allowed to write to its buffer, only read.

When each handler detects that it has processed a complete packet, it marks the buffer, making all data in it eligible to be read and processed. In this manner, only completed packets are processed further.

Each buffer then is broken into three parts – the unused portion, the used portion containing an incomplete packet, and the used portion containing completed packets.

There is one more buffer, the SD buffer, which is not allowed to be written to by any interrupt handler, only sequential code.

So, the main activity of the firmware is then filling and draining various circular buffers. Each buffer is capable of storing up to 1023 bytes long. It consists of an array of bytes, and three indexes into that array:

- The *head* pointer, where the next byte of data put into the buffer will be placed
- The *tail* pointer, where the next byte of data taken out of the buffer will come from
- The *mid* pointer, which separates completed and incomplete packets.

The pointers divide the array up into three parts:

- Data between the head and mid pointer is part of an incomplete packet
- Data between the mid and tail pointer consists of completed packets
- Data between the tail and head pointer is currently unused – either not yet written or already read.

The head and tail pointers are used as is conventional with circular buffers. Upon initialization, all three pointers point to the zero element of the buffer, and the buffer is empty. The buffer supports three main operations

1. Fill – add one byte to the head of the buffer and advance the head pointer. When the head pointer goes all the way around the buffer and is one behind the tail, the buffer is full. At this point, any further attempts to fill the buffer will fail, and data to be written may be dropped.
2. Drain – remove one byte from the buffer and advance the tail pointer. Once the tail pointer reaches the mid pointer, any further attempts to drain will fail, signifying that no data is available to be read.
3. Mark – move the mid pointer to the head, signifying that all data in the buffer at this instant is complete packets.

As an example: The UART routines read data from the two UART serial ports and store each incoming byte in a circular buffer, one for each port. They are driven by interrupt handlers, which can occur at any time, even in the middle of another interrupt. Periodically, the main loop reads the UART buffers and writes them to the SD card. But, the UARTs are relatively slow compared to the rest of the system. The main loop may go around many times before a complete packet is received. When the UART system detects that a complete packet has finally been received, it marks its buffer by moving the mid pointer to the head. At that instant, there is no data in between the head and mid pointers, and there is (at least) one completed packet between the mid and tail pointers.

Now when the main loop comes around, it checks how much data, if any, is between the mid and tail pointers. If there is any, it pulls the data out

one byte at a time and processes it, stopping when it reaches the mid pointer. Now all this time, the UART may receive another interrupt and write to the very buffer that the main loop is reading, but they won't interfere. If so much data arrives that the buffer becomes full, no more data may be written to it. This may mean that new data from the UART is lost, but old packets will not be corrupted.

Real-time clock and GPS synchronization

The LPC2148 features a nice hardware real-time clock with the following features:

- Calendar registers, containing year, month, day of month, hour, minute, and second
- Divide registers allowing the clock to run at any chosen rate off of the VPB clock (PCLK)
- Independent clock power and 32768Hz crystal oscillator, so the clock can run when the rest of the controller is powered off

Due to the way that the Logomatic is wired, the independent power cannot be used. However, all other features of the clock can and are used in this firmware.

The PCLK is also used to run the timers on the chip. Timer 0 is used used to time the ADC readings, and is set up to generate an interrupt at periodic intervals. Timer 1 is used as a general purpose 16-nanosecond resolution timer for applying timestamps.

The two timers and RTC are very independent. The timers run at PCLK frequency, but have independent divisors and counters, so they can be set and reset independently. The RTC can be run off of the 32kHz crystal (which the Logomatic does provide) and as such doesn't even fundamentally tick at the same rate as PCLK. However, since as the Logomatic hardware is wired, there is no way to independently power the RTC, it makes sense to program the RTC to run off of PCLK. In this way, its rate can be adjusted to match an external source, such as a GPS.

The LPC2148 also features a capture input. When the hardware detects an edge on the pin, it can capture the timer value of one of the timers, set off an interrupt, or do both. This is useful for capturing the PPS output of a GPS device.

Now, with all this timing circuitry, effectively three independent clocks, the hardest thing to do is keep everything in sync.

This firmware supports GPS synchronization in two separate modes. In one mode, the PPS signal is ignored and the RTC seconds and above is kept in sync with the clock is kept in sync to within 1 second. In the other mode, the PPS signal is used to steer the RTC and Timer1 frequency to keep the clock in sync to the limit of the accuracy of the attached GPS.

In mode 1, the $GPRMC sentence or equivalent SiRF packet is interpreted. This sentence contains the UTC date and time to one second accuracy, but not enough information to tell when the second starts. This data is used to just jam the RTC calendar registers, and no effort is made to steer the clock frequencies.

Mode 2, described below, is not yet implemented.

In mode 2, there is an initialization step and a steering step. The initialization step is performed when GPS lock is first acquired, and is repeated whenever the receiver drops and reacquires, or when the Logomatic receives GPS data again after not getting it for so many seconds.

The initialization step consists of two parts. First, a PPS signal is used to fire an interrupt. The interrupt handler zeros both the timer 1 and RTC subsecond counters (T1TCR is set to 2, then to 1. CCR is set to 2, then to 1.). The interrupt handler is then switched from initialization to steering. Next, the next $GPZDA message or equivalent SiRF packet are used to set the RTC clock registers.

After initialization, the RTC is allowed to run "free", meaning that the calendar registers are not changed and are allowed to update automatically. What is changed is the RTC divisor registers.

The PPS signal is disconnected from the interrupt, and just captured. When the main loop notices that the PPS has been captured, it calculates how many PCLK cycles have occurred between the last PPS and this one, then sets the limit on timer 1 and the divisors on the real-time clock to represent this amount of time. Basically, the next second is set to have the same number of PCLKs as the last second. Since the PCLK is in phase-lock with a quartz crystal oscillator, each second will be almost the same amount of time, and since timer 1 is used to so some care will have to be taken to calculate the number of seconds since the last PPS, since timer 1 has a 50% chance of wrapping each time.

If the GPS signal is ever lost, the clocks will run free at whatever their last setting was, and the synchronizer will reset to initialization.

Right now the GPS sync module supports sync with $GPRMC sentences in NMEA, and SiRF packet 0x29.

3 – NMEA Reference

When the Logomatic ADC writer is in NMEA mode, it writes a number of different NMEA sentences, all starting with `$PKWN_` , for **P**roprietary **Kwan,** in accordance with the NMEA specs. Each different kind of packet has a different letter. All packets have a proper NMEA checksum appended to them.

This firmware does not strictly adhere to the NMEA spec, 1) because I don't have it, it's a proprietary document that costs something like $450 and 2) the NMEA protocol dictates a maximum line length, which this does not adhere to.

PKWNA – Analog Readout

```
$PKWNA,842,513,465,524,791,383,473,462,521,0*4F
```

This sentence has a variable number of fields, one for time and one for each analog channel selected.

This sentence is generated at the selected ADC frequency, divided by the selected binning number.

The first number is the number of milliseconds since the last whole second. This number resets to zero each second. The next numbers represent the reading. The maximum value of each reading is 1023 times the binning number.

PKWNB – Baud rate measurement

```
$PKWNB,1,4800,116,0,1*66
```

This sentence is generated during the auto-baud measurement process. In order, the fields are

1. The port number being measured (0 or 1)
2. The baud rate being tested
3. The number of characters received in 1 second

4. The number of errors received in 1 second

5. A final judgement, where 1 means that the baud rate is judged to be correct, -1 means that not enough characters were received to be correct, and -2 means that too many errors were received to be correct. In general, a positive number is good, and a negative number is bad.

These packets are generated once per second during initialization, only if auto-baud is selected.

PKWNC – Analog setup

```
$PKWNC,1600,16,14,0,1.4,1,0.3,2,0.2,3,0.1,4,0.4,5,1.7,6,1.6,7,1.2,8,1.3*
7F
```

This sentence has a variable number of fields.

The first field is the sample rate, as specified in ADC frequency in the LOGCON file.

The second field is the binning rate, as specified in ADC binning in the LOGCON file.

The third is the ADCDIV variable used in configuring the ADC clock. This is set such that the ADC clock is as fast as possible while being less than 4.5MHz. See more details in the LPC2148 user manual.

The next fields come in pairs, one for each active ADC channel. The first field is the pin number as stenciled on the Logomatic, or zero for the battey voltage monitor. The second is the ADC side and channel used internally, where 1.4 represents channel 4 of ADC1. See the LPC2148 user manual for more details.

This sentence is generated once during startup.

PKWNL – Clock and load

```
$PKWNL,2009,7,25,6,9,58,18985713,0,2647612,0,0,0,9520344,0,423,0,0,0,0,0
,0,0*69
```

The first field six fields are the date and time in 4-digit year, month, day, hour, minute, second. If set from GPS, this will be approximately UTC. If set from the LOGCON file, it will be whatever time zone is intended. Otherwise, it is time from Logomatic startup.

Next are 16 fields, each of which holds a number between zero and sixty million. This represents the time used by various tasks in the Logomatic. Imagine the implied numbering on these, from zero to fifteen. Now, each of these numbers represents a combination of tasks, represented by the bits in the binary representation of each number.

Bit zero on means that the UART interrupt was active

Bit one means that the ADC interrupt was active.

Bit two means that the SD writer was active

Bit three means that the load measurement system was active.

Perhaps most importantly, index zero is when there are no tasks active. This is the amount of idle time the machine spends doing nothing. It is a Bad Thing if this reduces to zero.

This sentence is generated once per second at the top of the second, as seen by the internal RTC.

PKWNM – Machine setup

```
$PKWNM,2,4,5,1,1,1,1,1,60000000,60000000*4D
```

This packet is generated once during startup, and is generally the first packet in the log file.

The first two numbers are MAMCR and MAMTIM, registers belonging to the Memory Acceleration Module. For best performance, these should be 2 and 4, respectively. Consult the LPC2148 user's guide for more details on these.

The next set relate to the on-chip Phase Lock Loop used to generate the CPU and peripheral clock. The CPU can run at up to 60MHz, even though the Logomatic has a 12MHz crystal feeding the CPU. It uses this phase lock loop to step up the clock frequency. The numbers all come from the PLLSTAT register, and in the NMEA sentence are broken out into human readable form as follows:

1. MSEL – Multiplier select. This is the decoded (+1) value of the MSEL bits in the PLLSTAT register. The clock is first stepped up by this factor, and in the normal full-speed mode, this is 5.

2. PSEL – Clock divisor. The clock is next stepped down by this factor, and in the normal full-speed case, this is 1.

3. PLL Enabled, should be 1

4. PLL Connected, should be 1

5. PLOCK, PLL in lock, should be 1

The next number is VPBDIV, the VLSI Peripheral Bus Clock divisor. The CPU clock is divided by this number to generate the clock for all on-chip peripherals, such as the UARTs, ADCs, and timers. In the normal full-speed case, this is 1, resulting in a peripheral clock running at the same speed as the CPU clock.

Finally, we have the CPU clock frequency and VPB clock frequency, which in the normal case is 60MHz each.

PKWNP – PPS detection

```
$PKWNP,2009/07/25,06:10:31.225123*47
```

This is the date and time of a PPS pulse, according to the on-board RTC. Usually a GPS broadcasts a time correlation message along with this pulse to get an exact time to microsecond accuracy.

To receive these pulses, the PPS output of your GPS must be attached to the SCK pin of the Logomatic.

To be technical, the date and time to the second are from the RTC, but the millisecond part is from timer 1. However, these are kept in sync, so this shouldn't be a problem.

This sentence is generated as pulses are seen at the input pin.

PKWNU – Serial port setup

```
$PKWNU,0,38400,1562,97,16*7D
```

This is the setup parameters for one of the UART serial ports.

The first field is 0 or 1, saying which port is involved. The second is the requested baud rate. The third is the calculated clock divisor, and the fourth and fifth are the divisors broken up as required for the UART setup registers. See the LPC2148 manual for details.

This sentence is generated each time the baud rate is set. This is once per port for a specific baud rate, or up to nine times per port during auto-baud.

PKWNV – Logomatic Firmware Version

```
$PKWNV,Logomatic Kwan v1.0 Jul 25 2009 00:39:49*13
```

This contains the version string of the firmware, and the date and time, in the local time zone where the compiler is (Mountain daylight or standard time for me) that the program was compiled

This sentence is generated once during the startup process.

4 – SiRF packets

The Logomatic can be set to generate packets compatible with the SiRF protocol. In this way, ADC and Logomatic ancillary data can be cleanly interleaved with incoming GPS data.

Each NMEA packet described above has a corresponding SiRF packet. The full documentation for these packets are in the source code, (Use the Source, Luke!) but they are generally the same as the NMEA packets, except for being binary. Each field is usually only as wide as is necessary to contain the expected range of values, and the fields are typically unsigned big-endian integers of 1, 2, or 4 bytes.

SiRF packet ID	Matching NMEA sentence
0x2C (44)	PKWNA
0x16 (22)	PKWNB
0x2A (42)	PKWNC
0x15 (21)	PKWNL
0x17 (23)	PKWNM
0x19 (25)	PKWNP
0x2B (41)	PKWNU
0x18 (24)	PKWNV

5 – LOGCON.TXT

```
UART0 mode=None
UART0 baud=0
UART0 trigger=$
UART0 end=0x0A
UART0 size=64
UART0 timestamp=N
UART1 mode=None
UART1 baud=0
UART1 trigger=$
UART1 end=0x0A
UART1 size=64
UART1 timestamp=N
Start Time=20090704000000
GPS Sync=1
Powersave=0
ADC mode=Text
ADC frequency=100
ADC binning=1
AD0=Y
AD1=Y
AD2=Y
AD3=Y
AD4=Y
AD5=Y
AD6=Y
AD7=Y
AD8=Y
```

If the Logomatic does not find LOGCON.TXT on its storage device, it will create one as above. This version of Logomatic actually reads the tag before the equals, so you may add or delete lines. If a line is not present, its default is usually such that the associated feature is turned off. The labels on the AD settings correspond to the numbers silkscreened onto the physical part, except for AD0, which is an internal battery monitor.

5 – Version History

1.0 Initial release

1.1 Bug fixes, interpretation of NMEA sentence $GPRMC so that GPS Sync works when the GPS is in NMEA mode. $PKWNP at microsecond accuracy. Power save mode installed

6 – Fine Print