ST. MARY'S UNIVERSITY

School of Science, Engineering, and Technology
Department of Engineering

E.L.A.I.N.E.

Environmentally Lightweight Artificial Intelligence Navigation Experiment

By

Matteo K. Borri

Senior Design Presented to the Department of Engineering
In Partial Fulfillment of Requirements
For the Degree of

Bachelor of Science
In
COMPUTER ENGINEERING

San Antonio, TX
April, 2007

Supervising Advisor:

Dr. **Mehran Aminian**,
PROFESSOR OF ELECTRICAL ENGINEERING

Table of Contents

Acknowledgments

Special thanks to the following people for their support and help :

Mehran Aminian
Felix Carrillo
Crystal Carrillo
Melody Doepner
Chip Gracey
Dave Hatch
Daniel Horn
Djaffer Ibaroudene
Gary Karshner
Christina Keller
Tim O'Connor
Aurelia Ortega
Bahman Rezaie
Franz Schorp
Antonio Solda'
Paul Uhlig

This project is dedicated to the memory of Pietro Monti

Abstract

This project proposed to design, build and program an autonomous sailboat from an existing RC sailboat kit; in order to achieve this with as few modifications as possible to the original platform, and make use of modular design techniques, the single-board computer controlling the boat will be able to can interface with any standard radio controlled vehicle, and give it autonomous navigation and telemetry capabilities. This Navigation and Communication Artificial Intelligence (NAVCOM AI) takes in data from a GPS set and augments it with other sensors (compass, altimeter, accelerometer etc.) to increase positioning precision; this data is used for waypoint-based navigation and/or telemetry. Application-specific sensors can override waypoint tracking to provide for obstacle avoidance; waypoint data can be updated on the fly via the telemetry transceiver, allowing for one AI-controlled vehicle to follow another (or track a homing beacon). A separate watchdog and switchover system allows the user to resume manual control transparently, keeping the AI as a telemetry system only.

Steering a sailboat is one of the most computationally intensive tasks in navigation: to this day no autopilots are commercially available for sailboats. In addition to basic navigation, problems such as compensating for current drift, wind crosspush, and the “tactical” decision on when to tack and jibe depending on bearing to target and wind conditions had to be solved. After implementation, the AI-controlled boat is able to complete a GPS-defined circuit with a two-meter (6ft.) margin of error, and decide autonomously when to tack or when to use its auxiliary motor should she have to go against the wind to reach a waypoint. In addition, the NAVCOM AI computer provides telemetry data similar to those that would be available to the pilot of a real ship, to enhance performance when the boat is human-controlled

I: Introduction

Every year, the world's registered merchant marine uses twice as much fossil fuel as every car, truck and train in the United States. It is estimated that between 25% and 50% of cargo ships at sea at any given moment aren't registered with any nationality; these are usually older vessels that undergo only sporadic maintenance and thus can be assumed to have even lower fuel efficiency.

With the upward trend of oil prices, and basic security issues preventing widespread adoption of nuclear-powered cargo ships a return to sails as a propulsion method for for sea transport -- at least partially -- is foreseeable in the next 10 to 20 years.

The main expenses in operating a ship are fuel and crew; while a wind-powered ship would save enormously on the former, it is a historical fact that a large part of the reason why sailing vessels were all but commercially abandoned in the late 1800s is that they require a large number of skilled professionals to be operated.

An AI-driven sailboat could drastically cut down on the manpower required to run it. On a merchant marine ship, having an AI run the rigging would bring crew requirements down to current powered-vessel levels, while preserving the significant fuel savings from only requiring engine power when launching, docking or becalmed.

A second application is found in research. Unmanned sailboats remain at sea for a long time, generate their own power via solar panels or wind generators, and not affect any data collection with chemicals or noise produced by the engine.

Mankind has been sailing for thousands of years. Sails probably rank just below the wheel in their contribution to shrinking distances. This is a powerful legacy to build on. A (mostly) automated cargo ship would be able to cruise at about half the speed as a current cargo ship, at maybe 25% of the cost or less. This makes AI sailing an excellent fit for goods that are generally sold in bulk (and thus traded as options) and have a long shelf life, including grains, canned food, and most raw materials that do not require refrigeration or special handling.

I have designed and implemented an autonomous sailboat prototype as a means to determine what sort of sensors and processing power is required to effectively and safely operate such a device. The boat will carries separate logic systems, which are able to communicate or function independently of each other: the goal of this project is to prototype an AI-controlled sailboat that can function on its own power and intelligence until multiple malfunctions occur. The mechanical platform used in this project is a 1-meter remote controlled sailboat

whose remote control will be kept installed alongside the AI to allow manual override; from a nautical standpoint, the boat is classified as a single mast, single hull yawl with boomed jib; it will be supported by the following subsystems.

The NAVCOM AI will be designed from the start to be modular, its components able to be fit into other vehicle platforms; other subsystems are specific to ELAINE as a sailing vessel. A number of subsystems are planned:

Power Plant -- Wind-powered, with internal battery to run the electronics. Electric motor provides auxiliary propulsion. In a production model, the motor could act as a generator – larger vessels may also carry solar panels.

Obstacle Avoidance -- Single bow-mounted ultrasonic range sensor to detect obstacles (radar and underwater sonar could be mounted on larger vehicles).

Wind Direction -- Hall-sensor based weather vane mounted topmast, gives relative wind direction. Knowing the wind direction allows the boat to angle its sails optimally.

GPS - Gives current position and velocity, and tracks waypoints. The GPS communicates with other electronics via a serial cable using the NMEA standard.

Accelerometer – Returns current pitch and roll of the vessel, allowing it to detect dangerous weather conditions.

Radio Modem – Waypoint coordinates and other commands are broadcast to the AI through the radio modem. A route system determines if a particular waypoint is a final destination or an intermediate destination. The radio modem can also be used to directly control the boat if required. The modem can communicate back to a PC current status, distance to waypoint, AI state and other telemetry information.

Auxiliary Motor -- A small electric motor is mounted over the rudder, to use during approach/docking and when wind speed is insufficient to guarantee motion.

Solar Panels – The original intent was to have solar panels power some or all of the boat's systems. Due to weight and cost considerations, this has proven unpractical – however, a larger boat could easily support this.

Voltage Sensor -- A simple low-battery indicator alerts the processor, which can then decide to underclock itself to save power and use the high-drain components more sparingly.

Rudder Servo -- Used to steer the boat.

Sail Servo -- Heavy duty large servo used to angle the sail; one servo handles both sails through a simple pulley system.

Computation -- Microcontrollers will be used to run the boat, one to preside over each sensor , and one to coordinate. These will be able to clock themselves up or down (up to a max. 8Mhz) depending on the power situation. A prolonged zero-velocity situation will allow use of the aux motor and disable the tacking AI until wind conditions change; a scripting language allows behavior updates over the RF link. If a sensor malfunctions, the associated processor will report the condition and shut itself off; the CPU will try to reactivate downed subsystems at appropriate intervals.

Interface -- A laptop will be used to remotely monitor the boat. GPS coordinates for waypoints will be transmitted by the modem which will also be used for manual commands. The interface is ASCII-based to allow use on standard serial terminals, but an enhanced graphical interface is provided.

Payload -- No active payload is planned at this stage; however, data logging to record prevailing wind direction and mark the location of discovered obstacles is possible with the specified sensors. A trailed sensor pod containing temperature or water pH sensors is possible as a future development.

The boat will be able to follow a route using GPS. If the boat is required to go against the wind, it will plot a course allowing it to tack against it, and will try to optimize the tacking angle for shortest travel time (which does not equal shortest travel distance). In addition, the boat will modify this calculated course to quickly veer away should it find an unexpected obstacle.

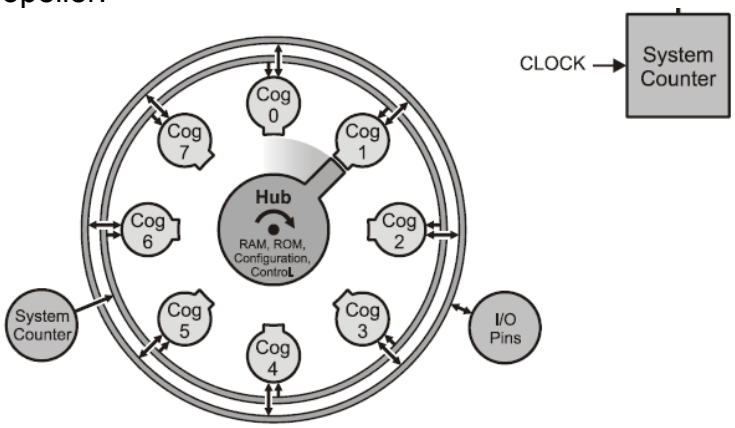
II: Software – NAVCOM AI

In order to perform any navigation operation, the speed, altitude, and relative location and direction from the plane to the target must be known. The "backbone" of the Navigation and Communication Artificial Intelligence system (NAVCOM AI) interprets and interpolates GPS and sensor data to provide and transmit digital navigation information. The application-specific AI rides on top of this system and uses its data to perform navigation tasks.

The NAVCOM AI backbone is a simple real-time operating system that runs on the Parallax Propeller 8-core microcontroller: it not only provides applications with a constantly updating memory map containing position and navigation data, but also present an API (Application Programming Interface) for the navigation application to control servos, send data to a base station, and even change interpolation parameters on the fly; simple applications can put in expression form and loaded wirelessly, while more complex applications such as the tacking AI used on ELAINE or the drop-distance prediction used on the *Ithuriel* aircraft must be loaded at compile-time as a subprogram within the OS.

The Parallax Propeller is the first in what will probably be a long line of multi-core microcontrollers. It has 8 separate processors, or cogs, that can run function independently or in conjunction; the NAVCOM AI OS takes full advantage of this by allocating cogs dynamically.

Figure X: Interaction of eight cogs with the hub, or timing synchronizer of the Parallax Propeller.



Hub and Cog Interaction

On average, four or five cogs are used by the OS and one by whatever application is being run at the moment; the other two or three can be configured as FPUs by loading high-speed math routines into them. All non-FPU cogs

produce and consume information stored in a memory map consisting of a number of 32-bit memory locations mapped as IEEE-784 floating point numbers. Cog 0 always runs the main program, which in turn 'assigns' the next operation to the next idle cog on a first-come, first-serve basis. It is possible for an application to either "reserve" one FPU cog or prevent the running application from using any in case the application developer feels that deterministic timing for math operations is important (using the default behavior, the application can queue itself for a FPU and will be served depending on availability).

The NAVCOM AI OS is logically structured into subroutines, most of which present an API to the application; some of these always run on their own cog (generally to allow for parallel processing), while others are run by the calling cog. The math subroutines are a special case that will be discussed later.

NAVCOM AI's subprograms that are called dynamically by the main program (and often also by each other)

NAVCOM AI Processes	
0	Main/Scheduler
1	Memory map
2	Stack Length Debugger
3*	Servo Driver
4	Simple Serial Port Driver
5*	Full Duplex Serial Port Driver
6*	GPS Parser and Interpolator
7*	Sensor Parser
8**	Math (Basic)
9**	Math (Trig/Nav)
10	Expression Parser
11	Sync Timer
12	Printing Functions (Console)
13	Command interpreter
14	Route Handler
15*	TV Signal Driver
16 onward	AI Applications

**: May run in own cog
*: Must run in own cog

In terms of reliability, the dynamic allocation of cogs allows the program to continue functioning, albeit slower, if one or two cogs malfunction as long as cog 0 keeps working (the scheduler runs on Cog 0; some tasks generate a "heartbeat" value that the scheduler can check against in order to restart a stalled process). For example, if one cog malfunctions, NAVCOM AI would move its dedicated floating-point processing to only one cog – this forces more processes

to do their own math without using a FPU, and can cause synchronization errors if the interpolation rate is not reduced (the scheduler detects sync errors, and progressively reduces the interpolation rate). This causes the NAVCOM AI to trade off precision for accuracy, but allows it to keep functioning. Should another cog present problems, the analog TV output can be forced to turn off in order to maintain navigation functionality.

The combination of dynamic processor allocation and a static memory map may seem awkward, but goes well with the Parallax Propeller's architecture: the memory map is mapped as code memory and can be accessed very quickly by all processes near-simultaneously, while the Hub hardware ensures that race conditions are not possible. For contrast, in a normal cluster of personal computers the local memory is much faster than the shared memory, so the unified memory map advantage would not apply – in this case, local memory is limited to each cog's registers.

The Propeller can be coded for using either assembly language or the higher level Spin, which was designed specifically for the Propeller's multiprocessor capabilities. Spin is an interpreted, weakly-typed language whose syntax is derived from C and Python. Assembly is used for most high-speed tasks such as full duplex serial communication, advanced math and DSP; applications for the NAVCOM AI OS must be written in Spin in order to preserve modularity.

An outline of each process follows below:

Main Program (Scheduler)

The main program delegates the tasks that each cog should do, when they should do them, and makes 'sense' of all the data; it also initializes other processes, and monitors them to make sure they haven't hung. Its first task is to initialize the other processes for the first time – this is done by calling the command interpreter with fixed arguments.

As is common for microcontrollers, the main program takes the form of a loop. In pseudocode, the loop first checks whether the GPS interpolator has produced new information; if this is the case, the physical situation has changed and must be dealt with: the main AI function is called – borrowing from game development terminology, the AI has executed a physics "frame". Otherwise, there is no detected physical change, and the scheduler can call lower-priority functions. If the "transmit telemetry data" flag is raised, the telemetry function is called – this generally happens a set number of times per second on a frame immediately following a physics frame, in order to assure that the data are fresh; this is called a transmit frame. If no telemetry is required, the AI alternates between updating one line of text in the TV display (if present) in a display frame, and consuming the radio modem's serial buffer in a receive frame. Thus the AI loop might go through a progression of states such as

depending on interpolator and telemetry frequency. Regardless of frame type, the scheduler then checks for “heartbeat” values from the GPS interpolator and sensor parsers and restarts one or both in case of lockup (if the GPS interpolator locks up, the interpolation frequency is also automatically reduced); finally, before looping over the scheduler checks the control state (whether the AI is in control of the vehicle, or RC input is overriding it) and informs the user of it and GPS signal quality should they have changed from the previous frame.

A known problem in NAVCOM AI 1.1 is that a receive frame can take a potentially large amount of time if during it a command is received fully, because it must be processed – the command interpreter is called with the contents of the serial RX buffer string as an argument; in that case, all frames up until the next physics frame are preempted.

Subprograms:

Main AI function

The main AI function updates the derived values in the main memory map and does basic navigation calculations such as determining distance and turn radius to the next waypoint; should a waypoint have been reached, the function calls the route handler to determine what the next waypoint in the route is. The main AI function then runs the expressions for each servo, and then calls for a custom AI application (which can override expressions or routes, if necessary) – note that for basic waypoint-to-waypoint navigation, all tasks can be handled purely by expressions. The main AI function then uses the OS timer to determine whether telemetry should be transmitted during the next frame; Finally, it updates the servo values, which will be consumed by the servo driver on its next iteration.

Custom AI Applications

As of the current iteration of the NAVCOM AI OS (1.1), the source code file for the scheduler also contains other code, most notably the code for AI application; this is currently in the interest of expediency, but to promote portability this will probably change in the next release. There are up to 64 of these separate functions available, but only seven are currently being used (0-6).

Table X: Vehicle Specific Functions in the Main Program as of version 1.1

0	None
1	Sailboat
2	Powerboat
3	Plane Drop
4	Car

5	Plane Initialization
6	Plane Drop Simulation

The default ‘ai’ value is zero; this can be used for basic navigation by associating servo outputs with expressions through the radio modem, or can be configured just for telemetry; in this case, no application is called.

Each application can be thought of as its own program, called by the main loop; by coding a finite state algorithm (using the global variable AIState) it's possible for an application to make use of past information while making a navigation decision. Most NAVCOM AI functions are accessible to the application – it is possible for the application to change the interpolation rate, or force a sensor reset, or even shut down the entire AI (say, once a mission is complete). Given the way applications are called (an application step runs immediately after a physical-state-change step), anyone who is familiar with Moore FSA's can quickly and elegantly code an AI application.

1) Memory Map

In order to access universal variables in the multiple dynamically allocated cogs, a memory map in the form of a library was set up. It contains 81 four-byte variables that can all be accessed by their address. The pointers are set up as constants spaced four numbers apart, and are manipulated as follows:

long[@SensorData + SDAO + GPSTracking]

The variable type in this example is ‘long’, which is a 32-bit floating-point number in IEEE-754 format; Almost all the variables in NAVCOM AI are floating-point longs, except for global flag values. The ‘@’ indicates a pointer, and SensorData is the starting address of the memory map. SDAO is an offset indicating where the memory map is within the entire memory. This constant is what enables the memory allocated statically so that each dynamic processor always knows where everything is stored. GPSTracking is equated as a constant to 164. The next variable, lat_delta, is equated with 168, and so on. The variable GPStracking is therefore stored in the memory location which is defined by the sum of the pointer, the offset, and 164.

Since the map is a considered a library, each subprogram only needs to initialize a constant pointer for the starting address in order to access all the variables in the library.

2) Stack Length Debugger

This is a simple subprogram that ensures that the length of the stack does not exceed the limits set for it in the library: process stacks are initialized with pseudorandom numbers whose seed is preserved, thus allowing this process to monitor the maximum stack usage of other processes. Currently the stack length

debugger is mainly used as a tool to determine stack requirements, and is only called during certain display frames; in the next iteration, it will either be removed entirely as stack sizes are finalized, or be run by the scheduler during non-physics frames and be made able to detect a stack overflow and force a reinitialization of the offending process. The debugger's code was derived from Parallax's original debugger, but has been integrated into the NAVCOM AI API and greatly reduced in size while retaining its functionality.

3) Servo Driver

Servos operate on varying pulse widths, usually between 1000ms to 2000 ms for analog servos and 750ms to 2250ms for digital servos. Parallax wrote an assembly Pulse Generator to produce pulse widths by dedicating a cog to it. The NAVCOM AI OS uses this code as a base, but expands upon in by adding sanity checks (timing limits for the range of pulse widths used by servos) needed for this specific application. The limits prevent mechanical interference and loading if there is something physically in the way of the servo moving its full range; while the driver is able to generate 32 pulse trains, only 4 are currently used.

The NAVCOM AI OS standardizes on 32-bit floating point, and represents servo positions with values from -1.0 to 1.0 – the actual centering and servo swings are defined as constants and can be changed to accommodate different servo manufacturers. For example, setting Servo1 to 0 with default settings will cause the Servo1 pin to output a 1.5 millisecond pulse. While this conversion introduces a small delay, it allows application developers to have a consistent frame of reference that can be worked with reliably once the mechanical part of the design has been finalized.

4) Simple Serial Port Driver

This driver allows a cog to read serial data from a pin without assistance from another cog; it is an improvement upon the SSPD written by Parallax in that it allows setting a timeout value and will therefore not halt processing if, for example, nothing is connected to the serial port's RX pin. This is used for all sensor input pins, and for the GPS input pin should a dedicated cog not be available; in NAVCOM AI 1.1 the transmit function has been commented out to save space.

An issue with this driver is that since the SSPD cannot assume that one of the hardware timers is available, a software timer is used for the timeout feature – this restricts said features to baud rates of 9600 or below. Since both sensors and GPS transmit at this speed or below (NMEA standard specifies 4800 baud), this is not a high-priority issue.

5) Full Duplex Serial Port Driver

The FDSPD dedicates a cog to “ping-pong” serial transmitting and receiving which provides effective full duplex at speeds up to 1Mbps; this driver, written by Parallax, was modified to allow variable-length transmit or receive

buffers, and burst mode transmission by writing to the transmit buffer directly thus allowing the process communicating with the FDSPD to do so quickly.

Currently, a 64 bit receive buffer and a 128 bit transmit buffer are used – 64 bits being the maximum length of NMEA strings used by the NAVCOM AI, and 128 bit being the maximum size of a response packet from the AI to the base station. This driver is used for the radio modem in receiving commands and transmitting telemetry and responses, and for the GPS when available in order to improve reliability.

6) GPS Parser and Interpolator

The NAVCOM AI OS treats GPS parsing and interpolating as a single process for efficiency reasons: both activities are strictly necessary for navigation, both are mutually exclusive, and receiving GPS data takes a few milliseconds and occurs once per second; therefore the same cog is tasked with both operations, preventing unnecessary and time-consuming process switching (This is a violation in spirit of the NAVCOM AI's modularity, but was kept for its efficiency).

All known implementations of the NAVCOM AI currently use the Garmin Etrex GPS receiver, but any receiver that has a serial ASCII text output at up to 9600 baud could be used; formats supported are NMEA, GAMA and Garmin TXT, at 1Hz or 0.5Hz (the process needs at least two consecutive transmissions from a GPS upon startup in order to determine which format is used, and how often transmissions are made by the GPS unit); eventual irregularities in the transmission frequency are dealt with by using a separate cog to run the serial port, and having the parser/interpolator read from its buffer; in order to free a cog for other uses, the serial port driver is shut down as soon as the GPS sends a termination character.

Whatever the source, the process distributes GPS data into the global memory map, performing the eventual necessary calculations to derive whatever information the GPS is not providing in direct form (for example, some acceptable NMEA sentences provide heading, while the Garmin TXT format only gives velocity vectors that must be converted to polar form).

Once this is done, the interpolator is activated and set to run a specified number of times a second: values in the memory map are updated using linear interpolation, corrected using sensor input that the sensor parser has stored into the memory map (the process is asynchronous to enable the system to use the latest available data); thus, this interpolated position is obtained by dead reckoning if no other sensors are available, but makes use of bearing sensors (compasses or gyros) and speed sensors (tachometers, PSID instruments, accelerometers or solcometers) if they are available – a NAVCOM AI that has at least one speed and bearing sensor connected can know where it is with a theoretical maximum precision of 8 inches between GPS transmissions.

At the end of every interpolation step, the previous results are saved in a secondary memory map along with the difference between the new results and

them – this memory map is accessible to application developers who may have a use for the first derivatives of navigation parameters in their applications.

Two thirds through the interpolation process, the serial port driver is reactivated (the 0.33 second buffer time was added after noticing that some low-end GPS receivers were fairly imprecise with the frequency they sent information at). The interpolation process then resumes.

At the end of the interpolation process, control is returned to the GPS parser subroutine and the loop goes on. In case of GPS failure, or the GPS reporting invalid satellite data, the GPS parser (after notifying the main function) tries to interpolate its position based on the available data by having the interpolator perform one extra iteration; as explained with the interpolator, depending on what sensors are available the precision of this fail over system may be enough to allow GPS-less navigation for extended periods of time.

7) Sensor Parser

The sensor parser works in a similar way to the GPS parser, with the main difference that it handles up to five sensors and only accepts NMEA-like or tab-return format entries; in a departure from NMEA standards, the parser assumes that sensors produce data continuously (a NMEA instrument will still work connected to the sensor parser, but may stall it for some time). A serial string for a sensor is read, processed, and the process is repeated in a loop with other sensors – the NAVCOM AI motherboard contains a battery sensor and connectors for four more.

Between sensors readings, the presence of a radio signal from a RC transmitter (from a pulse detector, also pre-built into the motherboard) is also checked for and notified to the main program. Sensors can be built in any way consistent with the format expected by the parser:

$Pt,<x>,1234,<y>,1234.5,<z>,123[*]<fffff>[cr][lf]

The '$P' is a symbols that indicates this is a proprietary sentence in NMEA format (that is, not a standard NMEA sentence). The t letter indicates the sensor type; an uppercase letter indicates that the microcontroller running the sensor has already performed some processing, while a lowercase letter indicates raw values. Sensor types can be B/b for battery level, C/c for compass, S/s for speed and so on; an application developer can easily add their own sensor types by using the ParseNextInt and ParseNextFloat API functions and copying the logic of an existing sensor type parser.

The letter also serves as a priority indicator to follow for the next iteration; for example, a battery-level sensor can be considered lower priority than an altitude sensor for an airplane, and may be skipped during the next few loops to save time. All priorities are relative to those of the other sensors; if only a battery sensor is connected, it will be checked as often as possible regardless of its base priority. The priority system is handled by a 9-bit counter that increments each

sensor parser loop; sensor types are given a power-of-two priority that is compared bitwise with the counter value to determine whether to go through with the parsing or not. The counter value is also communicated to the scheduler, doubling as a heartbeat signal from the sensor parser process – a stalled value will cause the scheduler to reinitialize the sensor parser.

Continuing on the parsing, the x,y,z letters represent identifiers for the following numerical values; they are occasionally used in NMEA sentences, but not used and ignored. Numerical values are processed in the order in which they appear, and used appropriately depending on sensor type – for example, a c-type sensor (raw compass data) is assumed to be built from two perpendicular Hall effect magnetometers, and the two incoming numerical values are treated as magnetic field intensity for the x and y axis respectively.

The characters *, carriage-return and line-feed are all treated as valid terminators for that string; upon receiving a terminator, the sensor parser moves to interpret the received data. The NMEA format allows for a checksum between the * and the newline; currently this is also ignored, but this may change in the next revision.

Heading and speed sensors are parsed differentially – the change between the current and last value is recorded, and added or subtracted from the GPS-derived data. This has the useful effect of simplifying interpolation operations, and allowing a compass sensor to be installed in any orientation and still be able to give correct heading once at least one GPS packet is received. Once values have been received, they are stored in the global memory map to be consumed by the GPS interpolator or the main function.

Sensors can be connected and disconnected on the fly; by directly reading the existence of a voltage on the sensor receive pins, the parser is aware of the change almost instantly (should a sensor be disconnected mid-transmission, its last transmission is discarded). If no voltage change is detected between checks, the pin is assumed to either not have a sensor or have a sensor configured at an incorrect baud rate, and skipped for that iteration of the loop – this prevents hangups should a sensor cease to function.

It is possible to have multiple sensors of one type to ensure redundancy, as long as they are scaled in the same way – this was deemed to not be an issue because it's safe to assume that redundant sensors would be configured as to give identical data during normal operation.

8-9) Dynamic Math Library (Basic and Trig)

All floating point mathematical operations in NAVCOM AI are performed by this process. Since it is a collection of functions to be called, equations cannot be written in the typical way, i.e

x:=3+4

The math library must be called in the following manner

x:=m.fadd(3.0,4.0)

using prefix notation because the Spin language does not allow the overloading of operators.

In this example, the dynamic library is assumed to have a public function within it called ‘fadd’ (for floating point add). In most cases, this is all that the calling function needs to know; the math library justifies its “dynamic” name by dynamically allocating zero, one or two cogs to processing the operation.

During normal operation, the math library asks the Hub if there is an available cog; if there is, the FPU assembly routines are quickly loaded into it, the cog is started and given the relevant arguments (in this case the two addends), the result is read from the cog's memory and the cog is stopped. If there isn't a cog available, the “slow” correspondent for that function is called (in this case, m.fadd would redirect to m.slowfadd) and the calling processor be made to perform the operation while still running the Spin interpreter.

This is not the only possible case: Some operations such as Cartesian-to-polar coordinate transform use two cogs in parallel if available (one for magnitude and one for direction); some operations such as truncate or int-to-float do not take advantage of the FPU at all because, due to their simplicity, it would take about as long to start and stop a cog than to have the calling cog do the operation itself; finally, a few operations would take so long to evaluate in Spin that only an assembly version was coded – in that case, the calling cog is made to wait until a FPU cog is available (NAVCOM AI 1.1 only has this happen for the atan2 operation; this bottleneck will be removed in the next version by using a lookup table for it).

A drawback of the dynamic approach is that, depending on FPU availability, the timing for evaluating an operation becomes very nondeterministic: this is handled in two fashions.

First, the “slow” version of a function is always made accessible through the API should an application developer decide to forgo speed for determinism. Second, four commands exist to temporarily disable dynamic allocation to either taking over a FPU cog permanently and keeping it idle when not called (m.lock, with m.unlock to release) or force using the slow functions (m.forceslow and m.allowfast to stop); the last two are different from simply calling the “slow” functions directly in that an application may be forced slow depending on program flow, thus allowing an application developer to have their AI application “relax” and allow itself to run slower when navigation is steady and quick response is not required.

In in the next revision, these commands will be changed to m.fast, m.slow and m.auto by essentially combining the allowfast and unlock commands in order to further simplify the API.

To get past the local-memory bottleneck presented by running assembly functions in a cog and to allow a degree of parallel processing in navigation-specific operations, there are two FPU configurations, one containing basic math functions and the other containing advanced trigonometric functions such as arcsine or arctangent2 (the add, negate, multiply and sine functions exist in both

configurations to reduce the need for switchovers); the second configuration is still being optimized, and is stored separately to allow for quick portability when optimizations do happen.

Currently, the math library supports these functions:

Math Library Operations				
Function	Arguments	0	1	2
add	2	x	x	x
subtract	2	x	x	
multiply	2	x	x	x
divide	2	x	x	
square root	1	x	x	
int–>float	1	x		
float–>int truncate	1	x		
float–>int round	1	x		
negate	1	x	x	x
abs value	1	x		
log	1	x	x	
ln	1	x	x	
exponential	1	x	x	
sine(rad or deg)	1	x	x	x
cosine(rad or deg)	1	x	x	
tangent(rad or deg)	1	x	x	
arctangent	1			x
atan2 (math)	1			x
atan2 (nav) aka 2d heading	2			x
3d heading	3			x
Cartesian–>polar	2			x
math angle–>nav angle	1	x	x	
nav angle–>math angle	1	x	x	
compare–>int	3	x	x	
compare–>float	3	x	x	
2d distance	2	x	x	
3d distance	3	x	x	

(Note that polar–>Cartesian is not used often in practical navigation, and therefore has no prebuilt operation associated with it – however, it's possible to perform the operation by running a series of simpler operations)

10) Expression Parser

The NAVCOM AI can be programmed on the fly by the use of six alphanumerical expressions; one for each servo, one for tracking (determining where to go), and an extra equation that can be shown on the TV display or used

as variable for any of the others. This functionality requires an expression parser; in order to speed up processing, a RPN (reverse polish notation, or postfix) parser was used.

These expressions can be defined on the fly wirelessly, or within an AI application – particularly useful is the fact that if no AI function is used, use of expressions can still provide basic navigation. The expression parser takes RPN expressions made up of numbers (integer or floating point with the . character as the decimal point), lowercase and uppercase letters, and the following symbols:

Expression Parser Operations

Symbol	Functionality	Args
+	Addition	2
-	Subtraction	2
_	Subtraction reverse	2
*	Multiplication	2
/	Division	2
\	Division reverse	2
$	Square root (symmetric)	1
^	Power	2
&	Choice	3
(	Sine (degrees)	1
)	Cosine (degrees)	1
[	Sine square wave	1
]	Cosine square wave	1
~	Angle adjust	1
%	Modulus	1
:	Swap A and B	2
|	Absolute value	1
!	Negate	1
}	Clamp/Dead zone	2
{	Standard steering function	4
>	Greater than	2
<	Less than	2
	Equals (to nearest integer)	2

While most symbols are present in standard arithmetic, others require an explanation:

The $ symbol is used for symmetric square root, which can be expressed as the square root of the absolute value of a number, multiplied by the sign of that number – thus $(9) equals 3, and $(-9) equals -3. When graphed, this generates a curve that is symmetric about the origin (hence the name) – this is

used to both provide an uninterrupted curve, and to avoid unpredictable results should a negative value be used as an argument.

The _ and \ symbols are used to help developers and users get used to the RPN notation. On occasion, the inexperienced user will make an ordering mistake with non-symmetric operations and only notice after entering the two operands; these two extra operations allow for quickly correcting the error. On the other hand, developers may find these useful when testing an expression to later rewrite in prefix notation for use in Spin – when these two symbols are used rather than – and /, all one has to do to go from postfix to prefix is rewrite every operand and operation right to left. Prefix notation was considered for the expression parser, but postfix won out for ease of numerical comparison with commercially available HP calculators that use postfix.

The % symbol is a simple IF...THEN implementation equivalent to the if() function used in many Computer Algebra Systems packages: the syntax is “if condition > 0, return value1, else return value2); in RPN notation, this becomes C V1 V2 & and can be used to make simple decisions.

The ~ symbol adjusts a basic operation to work with angular values (in degrees), returning an angle between +180 and -179.999...; for example, a heading of -45 means north-west and a heading of 180 means south. This is particularly useful when converting from absolute angles (heading, bearing) to relative angles (turn amount, wind direction) or vice versa.

The : symbol swaps the values of the last two numbers or variables encountered in the expression and mostly exists for compatibility with RPN calculators that feature the “Swap x-y” key (incidentally, the NAVCOM AI can perfectly emulate a HP-35 hand held calculator); thus, A B - : is equivalent to A B _ and vice versa.

The sine and cosine operators (and) -- which must not be mistaken for parentheses as postfix notation doesn't use them – return the sine or cosine of a value; as with everything in navigation, this value is expressed in degrees. For example, 90 (would return 1. The square wave equivalents [and] perform the same function, but also promote the result to 1 if it is higher than 0, and to -1 if it is lower, thus generating a square wave corresponding to the sine/cosine wave. This is primarily used in conjunction with the operation timer to generate an alternating motion for a servo.

The >, < and = symbols return 1 if the condition they specify is true and 0 if false; as a note, the = symbol works to the nearest integer due to the possibility of rounding errors in floating-point math not tagging as equal values that are intended to be.

The } symbol is used for a “clamp” operation meant for use with rudder/steering servos: its syntax is V C } which returns V if V is greater than C or lesser than -C, and zero otherwise – this creates a “dead zone” around zero, and can be used to prevent rudders from making many power-consuming minute adjustments that have no practical effect on navigation.

Finally, the { symbol is a shortcut for a very versatile function that finds use in rudders and throttles, defined in standard notation as {(x) = ax^2 + bx + c$x where $ is the symmetric square root described above. This function can associate a value to another in a very flexible manner by simply altering the a, b and c coefficients accordingly, and is useful for testing and debugging.

The next revision of the NAVCOM AI function will incorporate a "hysteresis" function with a change window; this should prove useful as an extension of the clamp function for uses other than steering.

The operations listed above can take variables in addition to numeric values: variables have one-letter names and go from A to Z. At startup, the NAVCOM AI OS maps these values to a default memory map position (for example, A is altitude, B is battery as a ratio of low battery, C is compass and so on); these mappings can be redefined by the user or by the AI application. Using the default mapping, assigning to a servo the expression U 180 / would cause it to move linearly according to the calculated turn amount, with a full swing in either direction should a 180 degree turn be needed and a center position should the AI-controlled vehicle be on course – a good starting point for a rudder mechanism.

Lowercase letters give the change in that particular value from the last GPS interpolation to the current one; for example, using the default mapping and interpolating ten times per second, an a value of -0.1 would mean that during the last second the vehicle's altitude has decreased by one meter. With this format, the first derivative of any navigation parameter can be obtained by dividing the delta of that parameter by the delta of the operation timer (o when the standard mapping is used); thus, the expression s o / (again with standard mapping, s meaning speed) can be used to return acceleration in case an accelerometer sensor is not present. More details on the use of the expression parser can be found in the AI command manual.

From a technical standpoint, the expression parser is divided into a tokenizer and a solver: the tokenizer takes an ASCII string containing an expression, and generates two stacks from it – one containing the operations and operands, with the # character representing a numeric value, and the other containing the numeric values themselves as 32-bit floating point numbers. The tokenizer also performs basic "sanity checks" such as padding the value list with zeros or floating-point ones should an expression be malformed or written in a way that might overflow the RPN stack.

The solver is a standard stack-based implementation of a RPN calculator; the main highlight is if it encounters a "variable" token, it will retrieve the appropriate numerical value from either the main memory map or the secondary memory map that contains the deltas before continuing.

While this division between two is somewhat inefficient in terms of memory, it allows the time-consuming tokenization and error-checking to be executed only when the expression changes because of user or application

input; since the basic six expressions have to be evaluated during every physics frame, this method allows for only the solver to be called, thus allowing a significant decrease in execution time.

11) Dynamic Timer

The dynamic timer provides a way for different processes to use the system-wide (Hub-based) free-running timer to be used for wait functions – this is primarily used by the GPS interpolator when determining how long to wait until the next interpolation step once all the calculations have been made. The dynamic timer function contains a table of often-used time intervals as functions of system wide timer intervals (the Propeller operates at 80Mhz, but the numbers had to be fudged somewhat to allow delays caused by function calling, Hub availability and so on).

12) Printing Functions

The printing functions are used to output values in human- or machine-readable format as required. Most of these have been derived from those provided in Parallax's API, with minor rewrites to privilege speed at the cost of code size and to use the dynamic math library where appropriate; these functions return a pointer to a string that can then be output either from the radio modem or the TV by using their respective string output functions; the “plus” and “minus” characters can be specified separately to facilitate use in navigation (for example, when printing coordinates, north and east are plus while south and west are minus).

The functions to convert an ASCII floating point or integer values also reside here – a peculiarity is that these return not only the desired value, but (via a by-address parameter) the number of characters that the converted number used up. This is particularly useful when extracting values from NMEA and NMEA-like strings: when this parameter is fed the initial address for the string, it will return the address at which to start looking for the next numerical value; multiple instances of this can easily be put into a for loop allowing for fast and readable NMEA or tab-return parsing code.

The conversion function into the 5-byte binary telemetry format is also counted as a printing function; since this is not a standard format, it requires a full explanation. A common problem with sending binary data serially is that it's possible for values to be misinterpreted as control characters (a cluster of zero bytes being the most prominent or common case); various methods are used to prevent this, from forcing repeat of control characters to using fixed-width binary strings.

My solution consists in treating transmitted serial bytes as ASCII characters if their MSB is low (the NAVCOM AI uses standard ASCII characters only, so this is absolutely not a loss) and as binary values if the MSB is high. Since the Parallax Propeller is a 32-bit machine and since we use a floating point format that also conveniently fits in 32 bits, the most common binary byte

combinations will be multiples of four; the MSB's of each of the four bytes are saved as bits 0 through 3 of a fifth byte, then set to 1. Bit 4 of the extra byte is set if the original value was a floating point number, and cleared if it was an integer number (there being no way to tell a 4-byte float from a 4-byte int by looking at the value itself); bytes 5 and 6 of the extra byte are used for a simple checksum by summing the values of the four original bytes and taking the modulus-4 value; finally, the MSB of the fifth byte is set to 1 to mark it as belonging to a binary packet. The five bytes are then queued for sending through the serial modem normally through the string-output function (it is possible to display them on the TV terminal, although it is rather pointless to do so), or stored.

```
76543210 76543210 76543210 76543210 becomes
T6543210 T6543210 T6543210 T6543210 TccF7777
```

where T is true, c is checksum and F is float status.

With the reverse process, the NAVCOM AI Console (or any other base-station application that may be written) can reliably reconstruct binary data when it appears in a telemetry string; since the encoding is extremely simple, binary values can even be decoded by hand during debugging.

13) Command Interpreter

The command interpreter can be called by any function with a fixed argument (most notably, it is used by the scheduler to initialize or reinitialize processes) or by the scheduler when a full command is received from the radio modem. For a full reference of commands, a command list is available.

14) Route handler

The route handler is called whenever the distance between the AI-controlled vehicle and its currently selected waypoint is less than the specified arrival distance. A route is specified by indicating a starting waypoint and an ending waypoint – the route becomes enabled (if it's tagged as active) once one of the waypoints in it is selected as the waypoint to reach. The route handler simply changes the waypoint to the next one in the route, skipping eventual invalid waypoints and turning the route off once the last waypoint is reached; this simple subroutine emulates a functionality found in many GPS sets. An AI application can override the router handler entirely.

As of the current revision, the NAVCOM AI OS only supports one route because none of the developed applications require more than one, although adding more is fairly trivial.

15) TV Signal Driver

The TV driver uses a signal generator written by Parallax in assembly; this routine is given a byte array representing a 40x14 ASCII display, and generates the appropriate composite signal to display it as white text on a black or blue

screen. The string-out function associated with the TV terminal simply “prints” to this array using the functions described previously to handle numeric values. The TV driver has been slightly modified to generate a signal that is either baseband (composite output on a RCA plug) or modulated (60Mhz, broadcast channel 3) thus allowing connections of an amplifier to the NAVCOM AI computer's TV-out pin for broadcast use; the TV output is mostly useful for debugging and for using the NAVCOM AI as a navigation aid in a human-piloted vehicle, and is generally kept turned off during autonomous navigation.

A version of the NAVCOM AI OS that has no TV out capabilities is available if the application developer requires more main memory and an extra cog to be accessible.

A future revision of the NAVCOM AI OS may add genlock capabilities to the TV signal generator, allowing the text to be superimposed to the output from a vehicle-mounted camera; however, this is not planned for the next revision.

16 onward) Applications

An AI application is a program riding on top of the OS that controls a specific vehicle or platform, using the OS-provided API for data acquisition and servo movement; specific AI applications have been developed for sailboats and airplanes. More information is provided in the section detailing the ELAINE application.

III: Command List / Programming Guide

Syntax:

All commands must start with the @ key: since this requires pressing a shift key and a key far away from it, this should decrease the risk of accidental keystrokes.

Maximum command length is 64 characters.

All commands are terminated by either the return key or the semicolon key (for concatenated commands)

Generally, if there is no space between command and argument, adjacency is necessary; if there is a space, any number of spaces may exist.

A **-** or **+** character can be put immediately in front of a number (no spaces) to specify its sign; default is +.

A number that is outside the specified range for that parameter will be replaced by either the minimum or maximum value allowed.

All commands have a timing estimate indicated; the value shows for how long the AI will process the command (and not be able to send telemetry or update servo positions); “near-instantaneous” means that no telemetry or physics frames will be skipped.

Notation for commands is:

Bold - Type exactly as written.
i[min...max] denotes an integer value.
f[min...max] denotes a floating-point value; the decimal point is represented by the **.** character. Floating point notation for values that do not have a decimal point is not necessary (1 means the same as 1.0)

Navigation Commands

@!

"Man overboard" command. This goes to the next free waypoint, and saves the current coordinates to it, then sets said waypoint as the destination. This pauses overrides any routes; the route must be resumed manually. All nautical GPS consoles have this feature somewhere, for safety reasons -- this is

also why it's designed to be easy to type in quickly. The operation is near-instantaneous.

@WP i[1...50]
Selects waypoint 1-50. If the waypoint was empty (e.g. it did not already point to a set of coordinates), the current coordinates will be stored as that waypoint's coordinates, such will be stored; also, if the AI is inactive (RC mode) the coordinates will be stored as well. Either way, the AI will try to steer towards the selected waypoint. The operation is near-instantaneous.

@WS i[1...50]
As above, but always set coordinates regardless of control situation. The operation is near-instantaneous.

@WC i[coord lat] i[coord lon]
Assigns specified coordinates to the currently selected waypoint, overriding any preexisting coordinates; the coordinates are entered in 1/10000th of an arc minute [Next revision will accept degrees and minutes, or deg/min/sec]. If only one coordinate is entered, the entire command is discarded. The operation is near-instantaneous.

@WD
Displays the current coordinates in 1/10000th of an arc minute in the form of a **@WC** command that should be entered into the NAVCOM AI as a command in order to save the waypoint; most terminals allow for copying and pasting of the **@WD** command's output in order to use it as a command in itself later on.

@RS i[1...50]
Sets waypoint 1-50 as the route start. The operation is near-instantaneous.

@RE i[1...50]
Sets waypoint 1-50 as the route end. The operation is near-instantaneous.

@RB
Begins route traversal; once on a route, the AI will traverse it (incrementing the waypoint number for the route start is less than that for the route end, decrementing otherwise) and change waypoints once ArrivalDistance is reached. If the end waypoint number is lower than the start waypoint number, the route will be traversed backwards. This command must be used after a @! command is entered in order to resume the route. The operation is near-instantaneous.

@RC

Cancels route traversal and clears the route (waypoint coordinates are not cleared). The operation is near-instantaneous.

@RD i[1.0 ... 999.0]
Sets the arrival distance from waypoint, in meters, at which the waypoint is considered "touched"; touching a waypoint allows the route pointer to advance. Default is 10 meters. The operation is near-instantaneous.

AI-specific Commands

@AIi[0...255]

Specifies which AI application function to execute each time step:
0 - None
1 - Sailboat
2 - Power boat
3 - Airplane
4 - Car
5 - Airplane 2
6 - Airplane simulation
5...255 Generic, with the number passed as a parameter.

This allows a single NAVCOM AI to be able to handle multiple vehicle types without needing reprogramming. application developers should be aware of the fact that a global variable AIchange is set during the first interpolation cycle after an @AI command is run; this can be used within an IF block to initialize that AI function's settings. The operation itself is near-instantaneous, but said initialization block (which will be run in the next physics frame) is likely to take up to a second depending on the AI application.

@AISi[0...255]

Changes the AI state for that particular AI -- AI state is a global variable that is used by a given AI application as a way to keep track of its memory. This is essentially a debugging command and should never be used during normal operation. The operation is near-instantaneous, but – again – a state change may trigger time-consuming routines within the AI application.

Kill-and-reset Commands

@KA

Reboots the NAVCOM AI entirely, forcing reinitialization. The operation takes takes between 2 and 5 seconds depending on satellite fix quality, number of sensors and display options.

@KZ
Forces the AI to shut down; it cannot be rebooted via software and must be power-cycled. The operation takes approximately 0.1 seconds to complete.

@KS
Reboots the NAVCOM AI's sensor parser; this command may also execute spontaneously if the sensor parser becomes stalled. The operation takes approximately 0.3 seconds.

@KG
Reboots the NAVCOM AI's GPS parser; GPS data will not be updated for approximately 2 seconds, but the operation itself is near-instantaneous.

@KV
Reboots the NAVCOM AI's servo pulse generator. The operation takes approximately 0.2 seconds.

Telemetry Commands

@TVB
Sends the TV signal to the broadcast output, at 60Mhz (approx. channel 3), to be amplified externally; the interpolations-per-second limit is set to the lower value (currently 60). The operation takes approximately 0.1 seconds, although the signal will take about 0.5 seconds to appear.

@TVC
Sends the TV signal to the composite output, to be modulated and amplified externally; the interpolations-per-second limit is set to the lower value (currently 60). The operation takes approximately 0.1 seconds, although the signal will take about 0.5 seconds to appear.

@TVO
Stops the TV signal generator. This frees up a processor to be used as a FPU, and increases overall processing speed; the interpolations-per-second limit is set to the higher value (currently 75). The operation is near-instantaneous.

@TSN i[0...4]
Sets the serial telemetry data to be sent 0 to 4 times per seconds, starting from the next second.

@TSN X
Sets the serial telemetry data to be sent in the AI frame following a physics frame, effectively tying the telemetry
frequency to the interpolation frequency -- the command **@SI 10 ;TSN X** will effectively request for telemetry 10 times a second. This is not recommended for use when more than 20 interpolations per second are performed.

@TSN
Toggles the serial telemetry on or off. If a **@TSN 0** command had been received earlier and **@TSN** is received, telemetry is turned on at a value of 1; otherwise telemetry will resume with the last entered value if it was last
stopped with this command.

@TSS [serial string]
Determines what information is sent with each navigation packet; a lowercase letter will indicate transmission in ASCII format, while an uppercase letter will indicate transmission in the 40-bit binary format described in the OS guide. While ASCII format has the obvious advantage of being human-readable from a standard terminal, it has to be clamped between 999.9 and -99.9 and thus cannot be used to transmit coordinates.
The data items available are:

Letter	Item
n	Current waypoint (ASCII only)
A/a	Altitude in meters
D/d	Distance to waypoint in meters
S/s	Speed in meters per second
I	Latitude of current position (binary only)
J	Longitude of current position (binary only)
K	Latitude of waypoint (binary only)
K	Longitude of waypoint (binary only)
X/x	Contents of X variable (see memory map variables below)
Y/y	Contents of Y variable (see memory map variables below)
Z/z	Contents of Z variable (see memory map variables below)

H/h	Current heading
B/b	Current bearing to waypoint
T/t	Current tracking to waypoint (by default, equal to bearing)
W/w	Current wind direction
I	Bitwise “virtual LEDs” sent by AI functions (ASCII only)

Letter order is not important. For example, **@TSS hbdsIJKLn** will cause the telemetry string to be composed of heading, bearing, distance, speed, coordinate pairs for position and destination and waypoint number – a reasonable data packet to figure out where an AI-controlled vehicle is and where it's going.

@TSP

Sends a single serial telemetry string containing the current data items as a response to this command, effectively interrogating the AI on its navigation and position status. [This command is not implemented yet]

@TSP [serial string]

Sends a single serial telemetry string containing the data items specified, as a response to this command, effectively interrogating the AI on its navigation and position status. [This command is not implemented yet]

Settings Commands

@SBf[6.5...9.0]
@SBi[6500...9000]

Sets the threshold for the low battery condition; the battery level variable will be a ratio between the current battery level (in volts) and the number specified here. If a value higher than 1000 is entered, it is assumed to be in millivolts and scaled accordingly.

@SE

Toggles echo on or off for the serial terminal, and notifies the user of the change. Default is echo off; if using a NAVCOM AI-specific console with a serial transmit buffer, it is recommended that echo be left off.

@SIi[1...60] (with TV on)
@SIi[1...75] (with TV off)

Sets the number of interpolations per second; 1 syncs with the GPS, while values higher than that will try to use available sensors to interpolate GPS data. If no sensors are connected, dead reckoning will be used. The operation timer delta variable is set at 1 / number of interpolations, thus giving a time step for use in calculations; the operation timer variable itself is updated during each interpolation step. The operation is near-instantaneous, but the new interpolation value will only be used at the beginning of the next second according to GPS time.

@Fi[1...4] f[-2.0...2.0]
Sets a fudge factor for servo 1/4's swing; values of magnitude greater than 1 will still respect the servo's mechanical limits, but allow for faster response in case of small adjustments, while values of magnitudes less than 1 will allow for more precise response. Negative fudge factors can be used to correct issues such as a rudder servo having been installed pointing in a direction other than expected.

@Fv[A...Z] f[-1000000.0...1000000.0]
Overrides a variable value for this interpolation cycle; if a sensor that is associated with a variable does not exist or is not functional, the variable remains (for example, it's possible to manually set wind direction on a car platform, or send altitude data to an airplane in level fight whose altimeter has malfunctioned).

Control Commands
@CK f[-2.0...2.0]
Adjusts the correlation between bearing and heading when cross-interpolating between the compass or gyro sensor and the GPS; a value of 0.0 turns the cross-interpolation off. The default value is 0.5.

@CT f[0...750]
Adjusts the global trim setting for servos; arrow keys (on some terminals and on purpose-built console applications) can be used to trim servo channels 1 and 2; the CT value determines how big each trim step, in microseconds of pulse, is.

Expression Parser Commands

@? [expression]
Sends a Reverse Polish Notation encoded expression to the parser; the result will be displayed on the serial terminal. This can also be used to "query" the status of a variable, for example, entering **@? A** will return the current altitude. Variable-less expressions are permitted: this allows for using the AI module as a RPN scientific calculator, which is occasionally useful on a boat or airplane. The operation takes between 0.1 and 0.6 seconds.

@EX [expression]

@EX

Sends a Reverse Polish Notation encoded expression to the parser; the result will be displayed on the TV screen if active, and saved in the memory map -- since this expression is run before all the others, it can be used as a parameter for other expressions. Unlike the previous command, this will execute the expression (and return the result) at every AI update cycle -- this is useful for testing an expression when generating a "wild" value for a servo, even temporarily, would be undesirable. The @EX command by itself erases the expression from the screen and from memory. When video is not used, the result of this expression is mapped at memory location 264, which corresponds to the letter I -- a useful mnemonic is that the expression is "Impatient" since it will be resolved first. The operation takes between 0.1 and 0.6 seconds.

@Ei[1...4]

Sends the result of the active expression, which has been entered with the @EX command, to a servo between 1 and 4. This does not erase the expression, so it can be entered for multiple servos if desired. A result of 0 means that the servo will be centered, while -1.0 and 1.0 will "slam" the servo one way or the other. Values higher than +-1.0 are treated as +-1.0 by the servo pulse generator. The operation is near-instantaneous.

@Ei[1...4] [expression]

As above, but the expression is entered directly. It is very important that no space be between the E and the servo number, and at least one space be between said number and the expression. The operation takes between 0.1 and 0.6 seconds.

@ET

Sends the result of the active expression, which has been entered with the @EX command, to the tracking calculator; this can be used to force the AI to maintain a specific heading. The default is @ET P which correlates the tracking with the bearing to the next waypoint -- for example, if the AI is desired to attempt to circle the waypoint at a certain distance, P 90 +<CR> or P 90 - <CR> should be entered -- this will cause the AI-controlled vehicle to move at right angles to its destination. The operation is near-instantaneous.

@ET [expression]

As above, but the expression is entered directly. The operation takes between 0.1 and 0.6 seconds.

Expression Operators

Op	Comments	Args	Example	Result
+	Addition of the preceding two values.	2	**3 7 +**	10
-	Subtraction of the preceding two values.	2	**10 3 -**	7
_	Flipped subtraction of the preceding two values.	2	**3 10 _**	7
*	Multiplication of the preceding two values.	2	**2 5 ***	10
/	Division of the preceding two values.	2	**3 2 /**	1.5
Op	Comments	Args	Example	Result
%	Modulus of the preceding two values.	2	**11 5 %**	1
****	Flipped division of the preceding two values.	2	**5 10 **	2
^	Raise to Nth power.	2	**10 2 ^**	100
!	Negate, equivalent to multiply by -1. This is particularly useful for variables, as it's not allowed to put a minus sign in front of them; by the same token, it is best to enter a negative number as F! than as -F. In addition, while Negate works the same way as multiplying by -1.0, it executes somewhat faster since it's a single token rather than two.	1	**100 !**	-100
|	Absolute value.	1	**-10 |** **10 |**	10 10
:	Stack swap: This swaps the last two numbers entered, and can be used to "flip" an operation as above.	2	**10 2 : ** **3 2 : _**	0.2 -1
$	Symmetric square root: this is an "expanded" square root that works with negative numbers. For example, 9 $ returns 3, and -9 $ returns -3. The reason for this is to help prevent errors, and allow use of the square root for rudders and steering systems which often need to be symmetric..	1	**-9 $** **9 $**	-3 3

)	Sine in degrees;Note that this is NOT a parenthesis, the symbol is used because of its shape making a good mnemonic	1	**90)** **45)**	1 0.707
(	Cosine in degrees; works as above.	1	**90 (** **45 (**	0 0.707
Op	Comments	Args	Example	Result
]	Square Sine: This generates a square wave that is synchronized with the sine function, from -1 to +1; the function works the same as sine, but any positive value returned by the sine is promoted to 1.0 and any negative]value to -1.0. This is mostly used in conjunction with the operation timer variable in order to have a servo follow an alternating motion.	1	**90]** **179.99]** **180.01]**	1 1 -1
[	Square Cosine: This generates a square wave that is synchronized with the cosine function, from -1 to +1 as above. This is mostly used in conjunction with the operation timer variable in order to have a servo follow an alternating motion – the two will be 90 degrees apart.	1	**90 [** **179.99 [** **180.01 [**	0 -1 -1
>	Greater-than comparison: This returns 1.0 if the first value is larger than the second and 0.0 otherwise. Obviously, this is mostly used with variables.	2	**20 10 >** **10 20 >** **-1 -2 >**	1 0 1
<	Less-than comparison: This returns 1.0 if the first value is smaller than the second and 0.0 otherwise. Obviously, this is mostly used with variables.	2	**20 10 <** **10 20 <** **-1 -2 <**	0 1 0
=	Equals-to comparison: This returns 1.0 if the first value is equals to the second within the nearest integer, and 0.0 otherwise. Obviously, this is mostly used with variables.	2	**1.1 1.9 =** **1.1 2.0 =** **1.1 2.2 =**	1 1 0
Op	Comments	Args	Example	Result
&	Choice: If the first argument is greater than zero, return the second argument, otherwise return the	3	**1 10 20 &** **0 10 20 &**	10 20

	third – this is a simple version of an IF...THEN block. This is generally used with variables, especially for the first argument.		**-1 10 20 &**	20
}	Clamp: This is mostly used for rudders, and generates a "dead zone" around zero in order to avoid wasting energy and time by performing small adjustments that have no practical effect. If the absolute value of the first value is larger than the second, return the first value, otherwise return zero.	2	**31 30 }** **-31 30 }** **20 30 }** **-30 30 }**	31 -31 0 0
{	"Twirl Function", explained below.	4	**X A B C {**	

The "twirl" function is actually a useful shortcut for a type of function that can generate a large number of response curves depending on its parameters. The twirl function is defined as

$$t(x) := a \cdot sign(x) \cdot x^2 + b \cdot x + c \cdot sign(x) \cdot \sqrt{|x|}$$

and has been experimentally determined to be very useful for rudders and steering systems – the twirl function is provided as a handy shortcut for use.

Variables

Navigation variables are stored in a centralized memory map; up to 26 variables, one per letter of the alphabet, can be addressed in expression. By default, the variables are assigned in the following manner:

Altitude in meters above sea level -- this is NOT altitude over the ground level!
Battery level as a ratio of low battery threshold (for example, if the battery is at 9.0v and the alarm is at 6.0v, the battery level will display 1.5)
Compass bearing, or the direction the vehicle is aimed at; if a compass sensor is not present, this will be left zeroed. A gyroscopic sensor may also be used.
Distance in meters from the selected waypoint.
ETA, estimated time to arrival in best-case-scenario conditions.
[F is currently unused, and can be used by an AI application]
[G is currently unused, and can be used by an AI application]
Heading, or the direction the vehicle is moving toward; due to crosswind or crosscurrent, this is not necessarily the same as bearing.

I is the result for the eXtra expression and can be used as an argument in other expressions; in addition, if the TV output is not used the result of a EX command can be seen by running a **@? I** command.
J is the last value for the output to servo 1; this can be used to calculate dynamic responses.
K is the last value for the output to servo 2; this can be used to calculate dynamic responses.
L is the last value for the output to servo 3; this can be used to calculate dynamic responses.
M is the last value for the output to servo 4; this can be used to calculate dynamic responses.
N is used as the first derivative of turn amount, taken from the last GPS reading; it is mainly used to calibrate rudders or steering systems. Since this is taken from the GPS only, it is not the same as **u** unless the interpolator is off.
Operation timer, from mission start, in seconds. The parameter **o** returns the AI's update period.
P is the best calculated heading to follow in order to reach a waypoint; the tracking equation is set to **P** by default (an **@ET P** command is executed at initialization). Overriding this variable is strongly recommended against.
[Q is currently unused, and can be used by an AI application]
Range, or arrival distance. Range be set with the **@SR** command even when not mapped to a letter.
Speed in meters per second.
Tracking, or the direction at which the vehicle should aim (either to follow a forced-tracking directive, or to reach a waypoint). By default, the result of the **@ET** expression is mapped here.
t**U**rn amount, or the adjusted difference between tracking and heading. When the AI is inactive, this can be seen as a direction to the user (e.g. "Turn 32.4 degrees left!"). Left is negative.
Velocity of wind as detected by the vehicle; this is mostly used for sailboats and as such, returns the APPARENT wind speed, as in, the wind speed summed vectorially with the vehicle speed.
Wind direction as detected by the vehicle; this is mostly used for sailboats and as such, returns the APPARENT wind direction, as in, the wind direction summed angularly with the vehicle's bearing. If a compass or gyro sensor is present, this value is interpolated with that value to allow for a measure of functionality should the wind direction sensor malfunction or become unavailable.
X is left unassigned by default; the navigation parameter that becomes associated with this letter can be sent down as part of a telemetry string (see the **@TSS** command).
Y is left unassigned by default; the navigation parameter that becomes associated with this letter can be sent down as part of a telemetry string (see the **@TSS** command).

Z is left unassigned by default; the navigation parameter that becomes associated with this letter can be sent down as part of a telemetry string (see the **@TSS** command).

Note that a lowercase variable means "how much has the value for this variable changed during the last interpolation interval", effectively being the delta for its uppercase variable; again, since lowercase **o** is the interpolation time interval itself, from 1.0 to 0.013 seconds, this allows a simple way to use the expression parser to solve differential expressions; a parameter's derivative can be obtained by entering [**a**...**z**] **o /** and can be recognized as the familiar Leibniz notation d[a...z]/dt.

Variable Commands

@V[A...Z] i[0...324]

The variables described can be assigned to point to different navigation parameters than the ones preselected. Note that since the memory map is byte-addressable but contains 32-bit values, the number entered must be a multiple of 4 – if a number that is not a multiple of 4, the next lowest multiple of 4 is used instead (so for example, typing **@VA230** would assign the variable A to memory location 228). Memory map locations 0 to 103 are occupied by the variables **A** to **Z** themselves; a full listing of the memory map location can be found in the table below. Unless indicated otherwise, all values are stored as floating point; if a value is stored as an integer, it is strongly recommended that it not be mapped to a letter variable except for debugging purposes.

By default, letter variables are mapped to offset 232 onwards – their functionality is explained above.

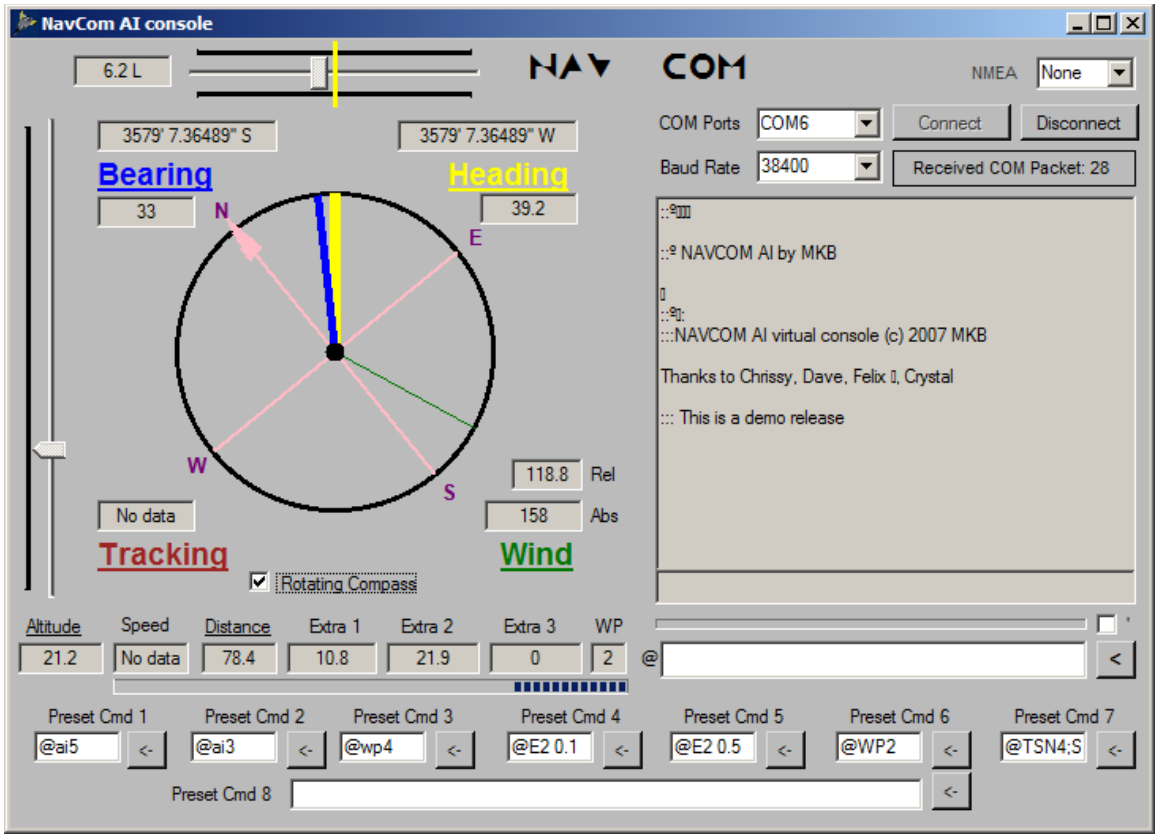
@V[A...Z]**~**

This command has a variable dereferenced (pointing to itself); this causes it to no longer being auto-updated. This can be used by putting the variable in a formula, and use the @F[var] command modify the formula without having to retype it in its entirety.

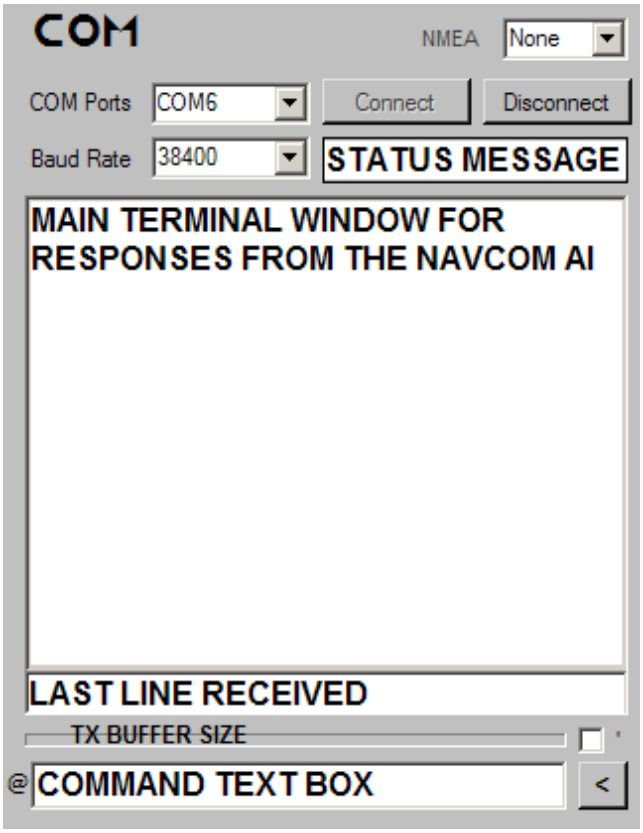
IV: Virtual Console

The NAVCOM AI virtual console is a Windows-based application designed to provide a graphical representation of data received via serial telemetry; while the NAVCOM AI platform is able to work with any serial terminal, an enhanced terminal with graphics capabilities allows for more intuitive access to information by a human pilot or a user monitoring the AI-controlled vehicle remotely.

The basic layout of the NAVCOM AI console.



In the interest of simplicity and ease of access, the virtual console runs in one form; any Windows computer with a SVGA-capable display or higher can show the basic layout correctly (the preset commands will not appear at 640x480 resolution, but the application is otherwise usable on older VGA displays.).

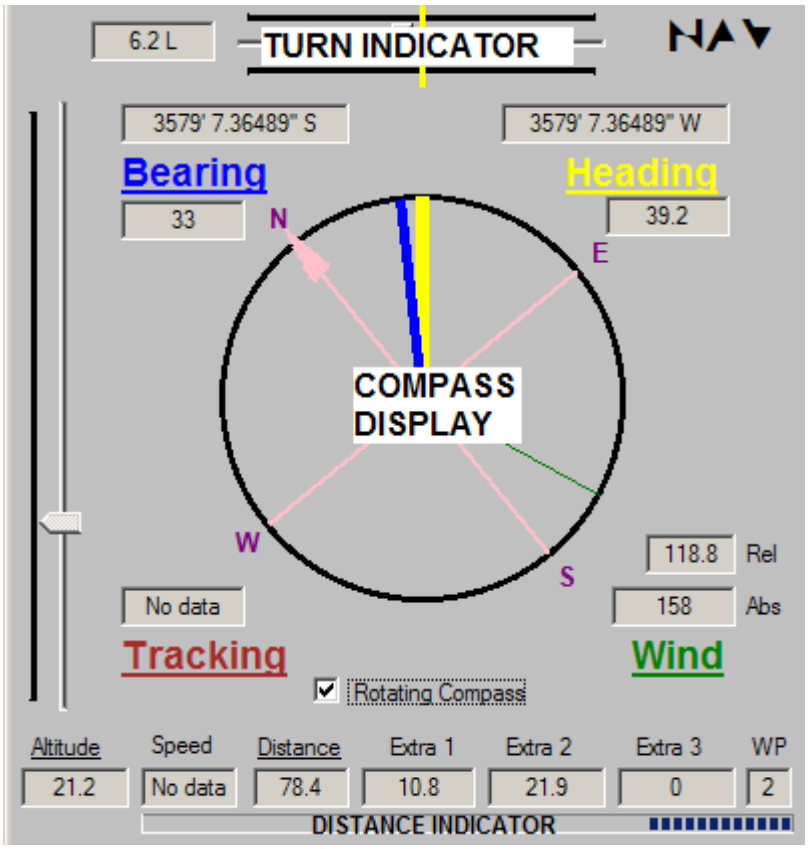


Communications Panel

The rightmost side of the application, marked COM, is essentially a general-purpose serial terminal optimized for use with the NAVCOM AI platform; the main read-only text box contains the AI's responses to user commands, and the smaller read-only box right underneath contains the last characters received (for efficiency, the console only parses an incoming data packet after receiving a carriage return). Right under is the current size of the used transmission buffer, set to 64 characters to match the NAVCOM AI's reception buffer – the console ensures that a maximum of 64 characters are sent at a time, and allows a user to monitor command length. The checkbox next to the buffer usage indicator allows bypassing of the buffer entirely, and sending characters one by one as would happen with a standard serial terminal.

The upper right corner of the window contains connection settings and a status message, which will comment on the last user action or the last data packet received from the NAVCOM AI; the connection settings are fairly straightforward and involve COM port and baud rate selection, plus a connect and disconnect button (the COM port is set to operate at 8 data bits, no parity, 1 stop bit); the NMEA option box controls moving-map output and will be discussed in the moving-map section.

Navigation Panel



The leftmost part of the screen displays navigation information, emulating as much as possible the layout of a standard H.S.I. (horizontal situation indicator) avionics instrument. Foremost is the compass display, with an underlying compass rose indicating cardinal directions and four “clock hands” giving navigation information as follows:

HEADING: The current heading of the vehicle, or its direction of travel.

BEARING: The bearing to the vehicle's target, or its intended direction of travel.

TRACKING: As above, but to the next waypoint (due to obstacle avoidance or tacking routines, this may be different from current target).

Tracking will not be displayed if equal to bearing, unless requested.

WIND: If available, the wind direction; the two numbers express it as a relative angle and absolute angle (both are often required in navigation).

Clicking on each of the four words will enable or disable display for each navigation parameter – this is different than requesting the AI to send

information, and is best used to turn off parameters that are deemed to clutter the display at any given moment. In general, the appropriate parameter's clock hand will turn on as soon as the relevant data is received by the AI, and stay on unless clicked upon. The numerical displays will remain on even if the graphics are not being drawn. All indicators are accurate to 0.1 degrees and display information in the nautical standard (0-359.9 degs)

The checkbox immediately underneath the compass determines if the compass rose rotates (thus keeping the HEADING clock hand pointing at the top of the display; this is recommended for manual navigation) or keeps in a fixed position with North on top (this is recommended for monitoring, as it makes it easier to follow the vehicle's progress on a map).

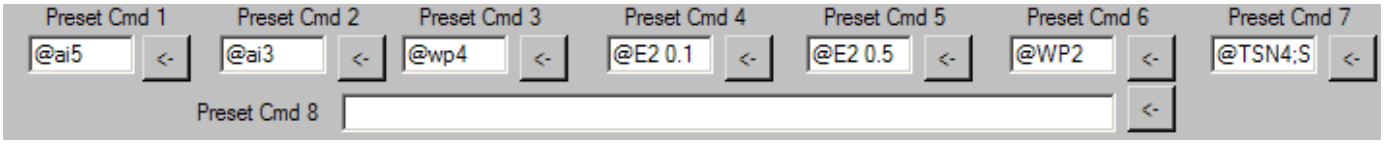
To the left side of the compass is an altitude indicator; as with the clock hands, this will appear once altitude data is received, and can be turned off by clicking on the Altitude label underneath – altitude data will still be displayed numerically. The indicator is accurate to 0.1 meters.

Underneath the compass display are numerical indicators for altitude (right below the altitude slider), ground speed, distance, three extra data items that will report the values of the NAVCOM AI variables X, Y and Z respectively (see the command list for more information), and number of next waypoint. Note that the Distance label can be clicked to disable or enable the graphical distance indicator – when distance is less than 100 meters, the horizontal bar underneath this row of numerical indicators will gradually fill until the waypoint is reached; this allows for quick, at-a-glance progress monitoring in the final approach phase. Unlike other graphics, the distance indicator must be enabled manually – it will remain inert until distance is less than 100 meters.

Above the BEARING and HEADING label are two numerical indicators for latitude (leftmost) and longitude (rightmost), in standard degree/minute format; these indicators will activate upon receiving coordinates from the NAVCOM AI.

In the top part of the window's NAV side is the turn indicator: a text box will give a turn suggestion made of a number and a letter such that – for example – 6.2 L will mean “Turn 6.2 degrees left”. The moving slider next to this message box covers 60 degrees left and right of the desired orientation, thus giving a higher resolution than the compass below – matching the slider to the yellow line is equivalent to matching the yellow clock hand to the blue clock hand, and indicates that the vehicle is navigating towards its intended target (If the vehicle is off course more than 60 degrees, the slider will be “slammed” to the right or left side depending on which way the vehicle should turn). The turn indicator display will only appear when HEADING and BEARING information is being received, and can be turned off by clicking on its message box.

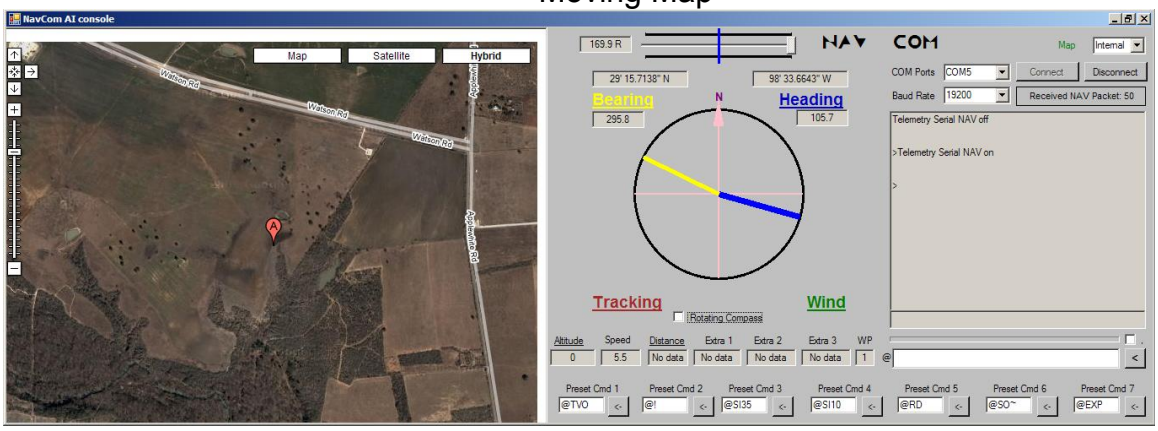
Macros



On the bottom of the application windows are eight “macro” command buffers that can be used to send frequently-used commands by clicking on the associated button to the right of the text box; these buffers can be pasted to as required by using the standard Windows key combination (CTRL-V). The 8th text box is intended for use with sending a commonly-occurring set of coordinates to the AI via the WC command (thus, it's possible to copy the output of a WD command directly in to this text box for future use).

Due to the availability of many options for moving-map software, both commercial and open source, it was determined to be impractical for the NAVCOM AI console to reimplement these functions from scratch. Instead, two options are provided for the use of a moving map – link to the Google Maps service, and NMEA output on a virtual serial port that can be redirected to an external moving map software.

Moving Map



The default option is “None” -- no output will be given. If “Internal” is selected, an internal web server is started on TCP port 3705. Clicking on “Map” will expand the NAVCOM AI window to show a browser window running a Java applet that retrieves map data from Google Maps and paints a marker on the AI-controlled vehicle's position; alternatively, a web browser of the user's choice can be pointed to http://localhost:3705 and display the same information. If the computer's Internet connection provides an external IP and does not block that port, it becomes possible for other Net users to monitor the AI-controlled vehicle's

position in near-real time. The Java applet updates the vehicle's position every second and redraws the map (recentering it) every ten seconds.

If a COM port is selected, the $GPGGA and $GPRMC NMEA sentences (two common sentences, at least one of which will be recognized by any current moving-map software) will be output at the standard NMEA baud rate (4800 baud) every second on said COM port and transmit position, heading and velocity data; in effect, this causes the PC running this application to emulate a standard GPS unit – using the provided virtual COM port pair, this information can be sent to any moving map software.

Using a physical COM port and the appropriate cable, it is even possible to send the output to any other computer or any other NMEA compatible device such as data loggers or differential-GPS base stations – since the NAVCOM AI understands the $GPRMC sentence, it's even possible to send the output to a second NAVCOM AI unit! (Note that this is an extremely roundabout way of doing so: two NAVCOM AI units can be put into communication directly via their expansion ports, or indirectly via their radio modems).

Console Tutorial

Using the virtual console is extremely straightforward for anyone who has ever operated a serial terminal package, and very easy to learn for users that have not; either way, it is advisable to practice using the terminal a few times before launching an AI-controlled vehicle.

First, determine what COM port the radio modem has been connected to; USB radio modems that emulate a COM port will generally notify the user of this information upon connection; if there are any difficulties, use the setup application that came with the radio modem; if the radio modem is embedded in a cell phone, it may be necessary to dial the cell phone connected to the NAVCOM AI and configure it for serial port emulation. (IMPORTANT: Be sure to set up the remote phone as a modem before launching the vehicle!). Once these steps are completed, select the appropriate COM port and click on the Connect button – the default baud rate of 38400 will be preselected for you.

Next, turn on the NAVCOM AI if you haven't done so yet; the copyright and initialization messages should appear in the main terminal window. Hitting ENTER after clicking on the command text box will result in an error message (“Commands must start with the @ symbol”) from the AI – this shows that communications are working; note that the @ symbol is put at the beginning of each line automatically by the console, although if you are used to typing it in yourself no ill effects will result. If communications are not working, check your COM port settings and be sure that they match those on the NAVCOM AI's radio modem.

Once communication has been established, try to enter a few commands and watch the response – note that every time the NAVCOM AI produces a response, you will be notified by the status box as to how many bytes were in it.

If you are practicing with the NAVCOM AI before launch (and if you are reading this, you probably should not have it running a vehicle yet!), put the GPS in demo mode and set it to a heading and speed of your choosing. Then, click once on the compass – this sends the @TSN command to the AI, having it start to send telemetry data once per second; what data will be sent depends on the AI type selected or the last @TSS command sent, but the defaults will give basic navigation data and coordinates.

The relevant items should turn on in the NAV section of the application window; change the heading in the GPS demo option screen, and note how the compass and turn indicators follow – the second delay is due to the fact that the GPS is sending data once every one or two seconds, and since the actual vehicle is not moving other sensors are not returning meaningful data. Click on the compass again to disable telemetry; the last displayed data remain in the text boxes and graphics, to allow for a screen print.

Depending on the AI application, five “virtual lights” might appear above the COM window – this is a way for AI application programmers to alert users of a particular situation that needs to be shown more visibly than simple text (for example, an obstacle within sensor range); these lights will turn on or off depending on the application, and their meaning varies with applications. A quick beep may also be produced.

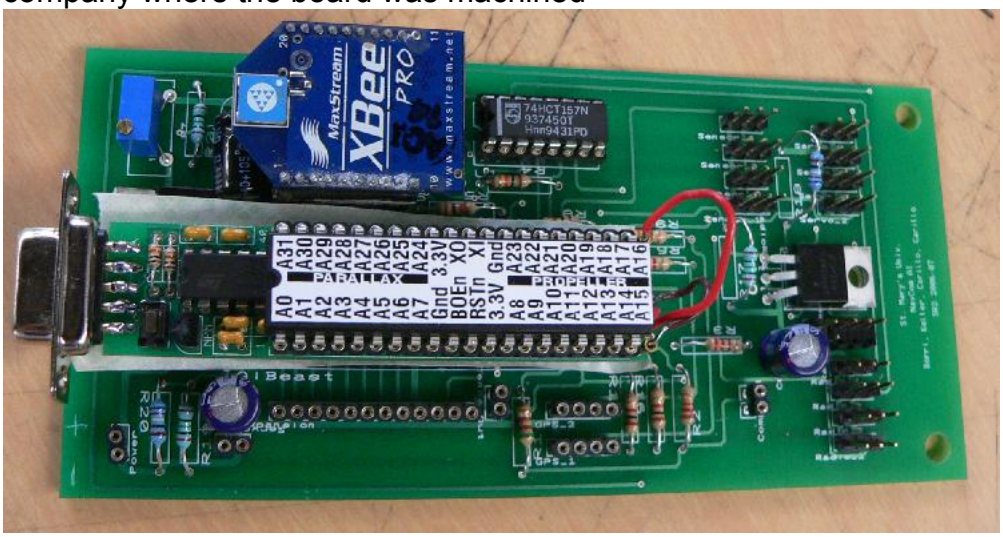
Now, enter the @TSN4 command; note how the display is updated faster, and the status window reports a NAV packet received more often. NOTE: When using a satellite phone, or a TACS (pre-2002 for most models) cell phone as the radio modem, do not set TSN above 2 as to not cause lost packets due to excessive traffic.

Feel free to experiment with AI commands and note their effect; for example, set a waypoint (in the AI using @ commands – GPS waypoints are not communicated to the AI) and try to navigate toward it using the compass and turn indicator; when getting close, click on the Distance label and note the arrival gage filling up.

Once you have familiarized yourself with basic operation, select “Internal” from the NMEA menu and click on “Map” to display the moving map (be sure to have a connection to the Internet before doing so; a nonexistent or misconfigured connection will result in a white window, in which case the map must be turned off and turned back on when a connection is established) or select a NMEA output port and set your moving map software to communicate with it. This is all you need to know to use this console.

V. The NAVCOM AI Board

The NAVCOM AI 1.1 board shown below was designed by me and Chrissy Keller, incrementally improving my hand-wired first prototype (1.0). The schematic was made in Multisim from the original drawings, and the board was designed in Ultiboard. The Gerber files thus generated were sent to the PCB Express company where the board was machined



NAVCOM AI 1.1 board, turned to the side; the “top” is considered to be, following the onboard text, where the DE-9 connector is.

This board includes three regulators, the Propeller chip (our main processing unit which rides on its own daughterboard with regulators, capacitors, crystal and EEPROM), the XBee Pro radio modem, two Picaxe-08M microcontrollers, and a multiplexer that performs double duty as a level shifter, in addition to all the necessary passive components such as resistors and capacitors; the precision trim potentiometer in the upper right corner is used to calibrate the input for the Picaxe responsible to discern battery level.

a. Regulators

The electronics on the NAVCOM AI require different voltages. The primary supply voltage (generally a 7.2V or 9.6V Ni-MH battery) needs to be regulated down to 5V for the Picaxes, radio, sensors and multiplexer, to 3.3V for the Propeller CPU and the radio modem (and optionally the GPS), and to between 5.5 and 6 volts for the servos.
I credit Christine Keller, Felix Carrillo and Crystal Carrillo for the decision to use the FAN1084, which is a low-dropout, three-terminal regulator with a 4.5A output current capability – the prototype used a LM317 5A adjustable regulator for the servos and five U7805s in parallel for everything else; this version (1.1) uses three different power buses in order to reduce crosstalk and voltage jitter, and to partition the risk of catastrophic breakdown due to overcurrent. There are a total

of four regulators on the AI board, the FAN1084s indicated above and a low-dropout 2.5A 3.3V regulator on the Propeller daughterboard that handles the low-voltage loads – this regulator is fed from its own FAN1084 in order to reduce possible heat buildup by performing the regulation in two stage, and to provide a clean DC voltage at input for added safety, while the other two FAN1084s handle the servos and the other electronics respectively.

b. The Propeller

The Propeller chip is a 3.3V system designed to provide high-speed processing for embedded systems while maintaining low current consumption and a small physical footprint. In addition to being fast, the Propeller provides flexibility and power through its eight processors, called cogs, that can perform simultaneous, independent, or cooperative tasks, all while maintaining a relatively simple architecture. The Propeller sits on a daughterboard that connects to the main board by a 40-pin DIP socket; the daughterboard contains a regulator, backup EEPROM memory, 5Mhz crystal, reset button and a simple three-transistor level shifter for RS232 serial communications for the programming port.

c. The XBee Pro

The XBee Pro is a RS232 radio modem with has an operating frequency of 2.4 GHz and sixteen software-selectable channels, as well as a 256 character buffer. The XBee has an indoor/urban range up to 100m (~300 ft) and a line-of-sight radio frequency line-of-sight range of up to 1.5 kilometers (slightly less than one mile). The transmitter's power output is 100 mW, and the theoretical maximum data frequency is 250,000 bps. The XBee has a receiver sensitivity of -92 dBm. For the NAVCOM AI, the XBee is configured as a transparent serial modem and its buffer split equally between transmission and reception; an integrated active antenna is used at both transmitting and receiving ends, although provisions were made for an external antenna should the need have arisen. To improve reliability, baud rate is kept at a relatively low 38400bps in both directions – each packet is sent three times and, in case of difference, the two identical packets are used (if all three packets differ, they are all discarded). Since all data transmission is asynchronous serial, any RS232-compatible device may be used – in particular, a GSM cell phone can be easily adapted to use text messaging services (SMS) as AI commands and responses; this would simply require a level shifter, an inverter and possibly a character buffer.

d. Picaxe 08M

The PICaxe08M is a very inexpensive microcontroller that runs a pBasic interpreter in a similar way to the popular BASIC Stamp product line; it uses an internal RC timebase with a digital potentiometer and can be run at frequencies from 32Khz (16 IPS) to 8Mhz (4000 IPS), with a recommended manufacturer rating of 4Mhz (2000 IPS). It has five I/O pins, three of which are programmable

as input, output or 10-bit ADC input. All these features are conveniently packaged in an 8-pin DIP for the price of about $3. Two Picaxes are present on board

The first Picaxe is merely used as an ADC to discern battery level, which is read through a trim potentiometer acting as a voltage divider from the unregulated battery input; this value is scaled and transmitted serially to the Propeller. As this unit respects the NAVCOM AI sensor standards, further details on the software can be found in the Sensors section.

The second Picaxe can be set, using a jumper, to discern pulses coming from either the radio input labeled Channel 1 or the one labeled Channel 5. At this stage, it's important that a human pilot be able to quickly override the AI in case of erratic behavior; therefore, this microcontroller's task is to quickly and reliably allow such overriding. Depending on a jumper setting, the Picaxe will read pulses from either Channel 1 or 5. If Channel 1 is selected, the reception of any pulse in the valid servo range will cause the Picaxe to have the multiplexer pass the RC signals to the servos; this is useful in case only a two- or three-channel servo is available, as is the case with most entry level RC setups. This mode allows the human pilot to effectively use the remote control's power switch as a switch between RC and AI operation – during RC operation, the Channel 1 pulse is sent to the multiplexer normally after being read by the Picaxe (a known problem with the Picaxe is relatively low input impedance, and the multiplexer handles any voltage drops caused by the reading). If Channel 5 is selected, the Picaxe behaves somewhat differently – a servo-type pulse with a duty cycle between 5% and 7.5% will cause control to be transferred to RC, while a pulse with a 7.5% to 10% duty cycle will assign control to the AI. This allows Channel 5, which is otherwise not forwarded to any servo, to be used as a switch to allow quick change in behavior if the human pilot uses a higher-end RC setup. Should no pulse be present, or be outside the valid lengths (with a 20% tolerance), the Picaxe will raise a signal to the Propeller notifying it of an assumed radio failure and allowing the AI to take the necessary measures (for example, a NAVCOM AI controlled airplane might assume that the RC transmitter is out of range, and set itself to fly in circles to allow the pilot to move closer to the plane's location). In either mode, the signal driving the multiplexer is also forwarded to the Propeller so that the AI may know whether it's in control or not; this can be used to, for example, switch from telemetry to navigation mode without any additional user input.

While realistically a single Picaxe could have absolved both roles, it was decided to separate the functions in order to prevent any overvoltages from the voltage divider from damaging the microcontroller and thus possibly disrupting the pulse detection and signal switching functions.

Neither Picaxe appears in the picture above, as both units and the jumper are situated under the XBee Pro.

e. Multiplexer

This component is a standard 74HCT157 noninverting 2-to-1 quad multiplexer, upgraded from the prototype's 74LS157 in order to negate the need for a separate level shifter; it is used to switch the servo input's signals between the radio and the AI and back, according to the pulse width detector described above. The multiplexer receives the servo pulses (period 20ms with a 5% to 10% duty cycle) directly from both the radio and AI, and sends either one to the servo outputs; in the process, the 3.3V signals from the AI are promoted to TTL level (5V).

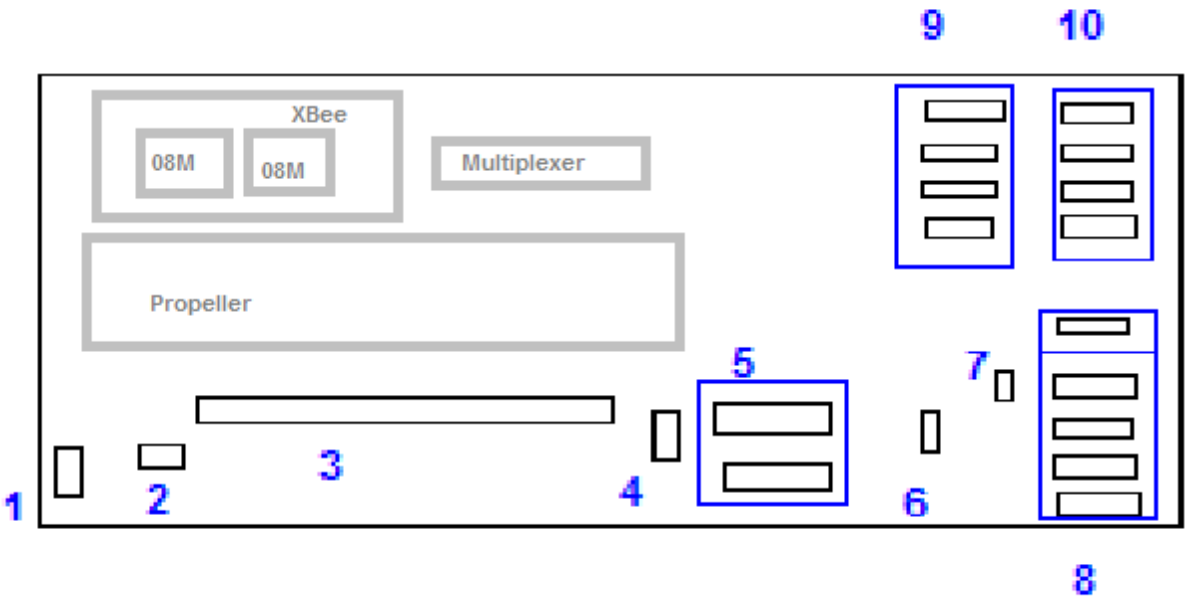
f. Connectors

Most connectors on the NAVCOM AI board follow the de facto standard configuration for servos (three pin, 100mil spacing, signal-power-ground), with the main exception of the GPS which is configured as ground-tx-rx-power in case the GPS unit needs to be set or reset via software. Using this standard allows for easy acquisition of extension cables, crimpable connectors and so on, while protecting both the electronics in the peripherals and the power supplies on the motherboard from overcurrent should the connectors be inserted the wrong way in. For all connectors present on the NAVCOM AI board, the ground wire points towards the top left corner: horizontal connectors have their ground pin on the left, vertical connectors on top.

Sensor pins use the same s-p-g layout, the signal being TTL serial data coming from each sensor into a different pin on the Propeller; a limiting resistor allows use of standard RS232 voltages for externally powered sensors.

The two TV outputs allow connection respectively of an external amplifier or a composite input to video receiving device.

Finally, the expansion slot provides application developers to direct access to the first 10 pins of the Propeller, a ground pin, and a 5V line connected to the regulator feeding the low-voltage regulator – it is thus possible to connect two NAVCOM AI boards through a high speed synchronous 8-bit bus (the two extra lines being used for DTR and RTS signals), even with one motherboard powering the other as long as the slave board doesn't need to support servos.



Power connector (2 pins, power and ground)

5) 4.8~6V servo power input if the internal servo regulator is not used (2 pins, ground and power)
6) Expansion slot or double-up connector (12 pins, 8 data lines – 5V power – ground – 2 flow control lines), can be used as general purpose expansion
7) Modulated TV output to signal amplifier (2 pins, ground and signal)
8) GPS connectors (ground – tx – rx – 3.3V power)
9) Baseband TV output to video composite input (2 pins, signal and ground)
10) Power output to RC receiver (2 pins, 5V power and ground)
11) Inputs from RC receiver; channel 5 is on top, slightly apart from the others (3 pins, ground – 5V power out to radio – signal in)
12) Sensor connectors, 4 on top (3 pins, ground – 5V power out – signal in)
13) Servo output connectors, 4 on top (3 pins, ground – 5V power out – signal out)

g. Power usage and electrical interfacing information

Outputs

FAN1084 – 4.5A each, configured for 5V output; servo unit can be heat sinked

7803 – 5A, 3.3V output

74HCT157 – 30mA sink/source per pin, 100mA total, 0-5V

Composite outpit – 0~2.8V analog signal, baseband

Broadcast output – 0~3.3V analog signal, CH3 (60Mhz), modified sine wave

Inputs
Sensors in – 3.3 to 12VDC signal high, -12 to 1VDC low, max sink 20mA each, 51KOhm input impedance average
Servo in – TTL, max sink 40mA each, high input impedance

Expansion port
5V power output, do not exceed 2A source
0V ground, do not exceed 2.5A sink
3.3V LTTL data lines, recommend using a 1K resistor if fed TTL signals

Power consumption (worst case scenario)
Propeller – 1.64A measured with all 8 cores running
Picaxes – 80 to 120 mA each at 8Mhz
GPS – 250 to 400mA typical (tested with LS-40EB and Garmin eTrex)

Using a 3300mAh Ni-MH 7.2V battery:
Maximum recorded total for whole board by itself 2.41A (17.25W)
As above, plus 3 self-powered sensors 3.28A (23.6W)
As above, plus 2 standard servos 3.65A, peak (26.28W)
As above but with 1 std, 1 heavy-duty servo 4.11A (29.6W)

A future revision may include a switching power supply for driving the servos, as most of the power is consumed there – the values indicated are worst-case peak currents, measured during 8-hour endurance runs and discounting startup current. With the indicated battery, system life is between 12 to 14 hours, with the CPU beginning to falter with the battery at approximately 6.55 volts as measured by the internal battery voltage checker.

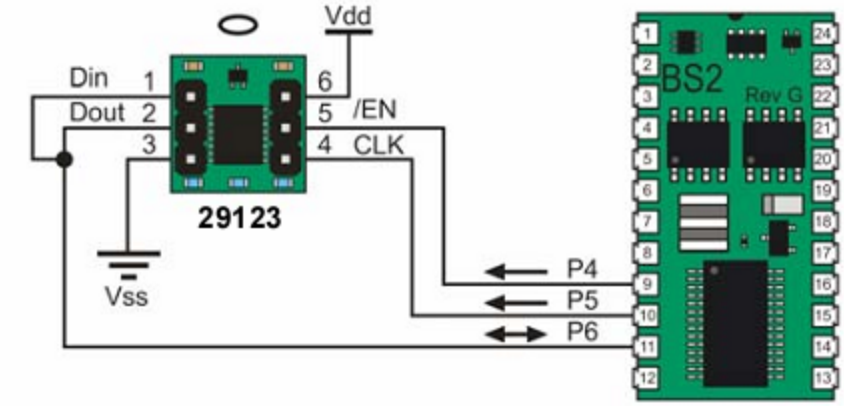
As the FAN1084 is inexpensive and features thermal shutdown, it will likely continue to be used to drive the electronics – while the cascaded linear regulator approach may seem inefficient, it has the beneficial effects of providing very clean DC for critical systems (CPU, GPS and radio modem) and spreading the heat buildup among the two regulators. A future low-power version of the NAVCOM AI board using SOIC components may use one or two DE-SW050 miniature switching voltage regulators to drive the electronics, along with larger bypass capacitors (experiments performed with a 1-Farad HPC capacitor show that it is able to power the entire board by itself for four second, time enough to at least send a distress data packet).

VI: Sensors

Sensors for a NAVCOM AI application, as explained earlier, can be of many types; the NAVCOM AI API allows an application developer to add sensor types quickly and reliably within the framework.

The NAVCOM AI OS requires that sensors transmit data in NMEA format; this generally requires a microcontroller. A compass is a common sensor to add to a NAVCOM AI vehicle due to its ability to provide orientation data with better frequency than a GPS; in a typical compass sensor, a pair of Hall sensors perpendicular to one another would be connected to a microcontroller and measured simultaneously (or near-simultaneously); the microcontroller then has the option of either converting the values to an angle by itself, or sending the raw values to the CPU and have it do the conversion – this “raw” versus “refined” model exists for most sensor types.

An example sensor configuration using a Basic Stamp 2 as a compass.



The NAVCOM AI in default configuration supports altimeter, clinometer, compass, gravimeter (accelerometer for the Z axis), up to three range sensors, and wind vane.

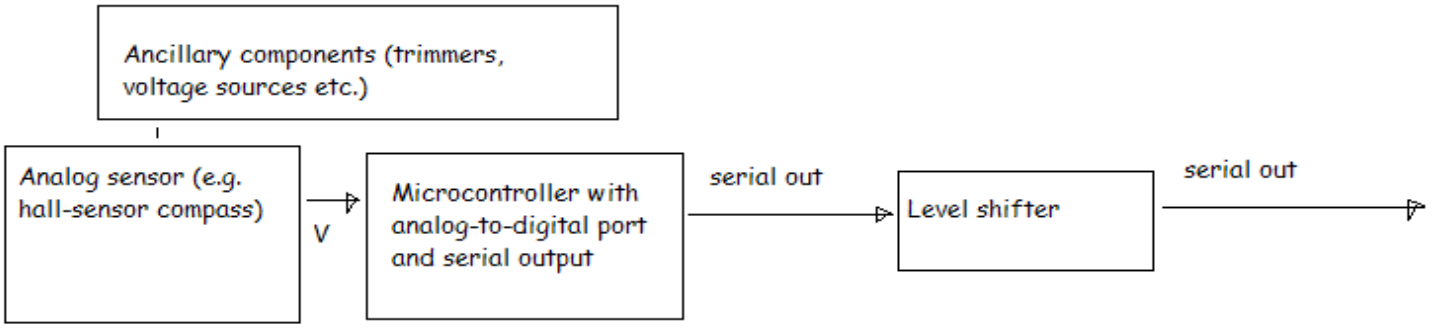
The motherboard contains a battery level sensor, which is itself read as a NMEA device, and a radio signal detector which, due to the necessity for quick reaction time, is connected to the CPU via two digital pins; one pin notifies the CPU whether it or the radio has control, while the other indicates an unexpected interruption of the radio carrier signal. A jumper is provided to enable or disabled this function; when disabled, the radio sensor checks for the presence of a valid servo pulse on channel 1 and reports its presence or absence.

When enabled, the radio sensor checks channel 5 instead, and reports whether the pulse is more or less than 1500 microseconds (allowing channel switches on high-end remote controls to be used to 'arm' or 'stand down' the AI board as a whole), using the second line to indicate an error if no signal or a spurious signal is received on channel 5 while the AI is supposed to be stood down – the AI can be set to interpret this as an unintended loss of contact with

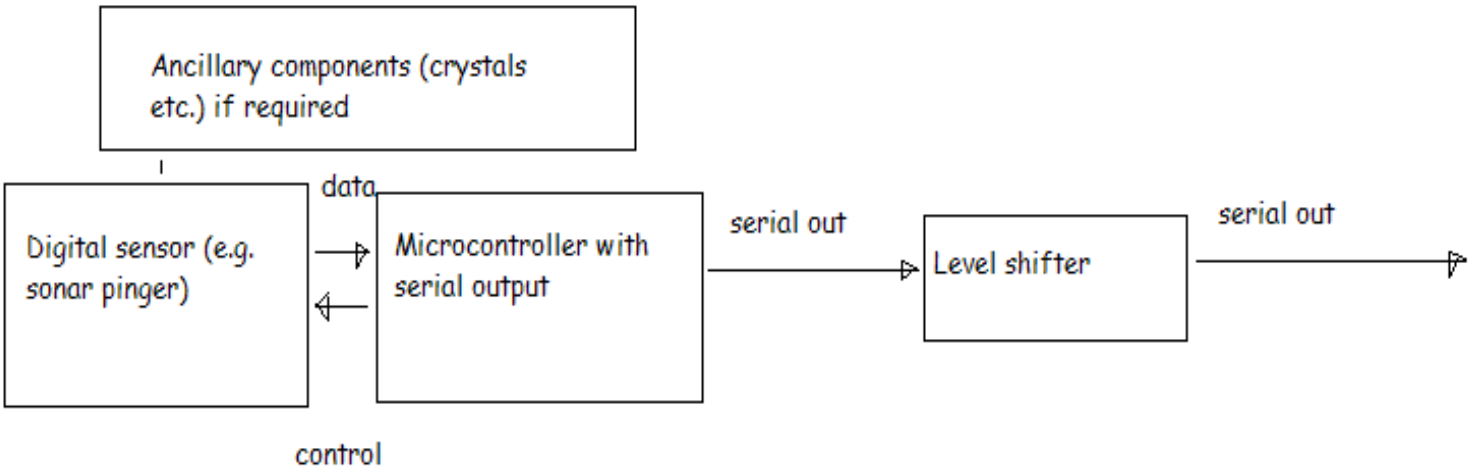
the remote control, and react appropriately (for example, by moving in circles and wait for contact to be reestablished, or by trying to return to the launching point).

Worth noting is that two or more AI boards can be cascaded through the servo interfaces; a slave board would be getting a “radio carrier” signal when the board above it in the chain has control. This can be useful for very complex applications that require more than two boards.

For the two internal sensors, the motherboard uses Picaxe-08M microcontrollers for their low cost, ability to be reprogrammed without special equipment, and integrated 10-bit ADC (used, for example, by the battery sensor). Since the NAVCOM AI OS consumes NMEA information, analog or digital sensors may be used as shown below.



Analog sensor block diagram.



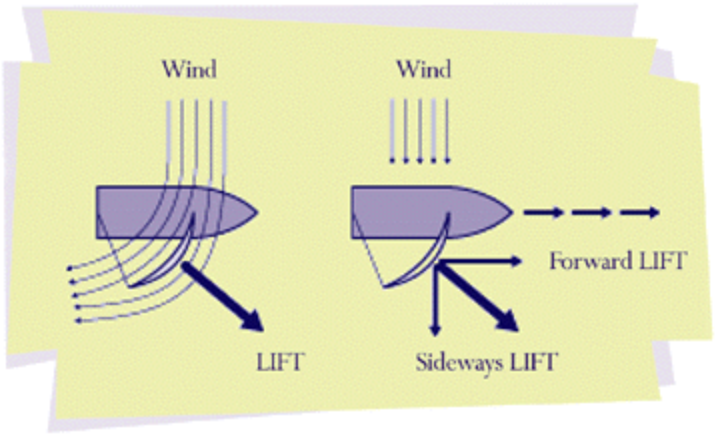
Digital sensor block diagram.

The level shifter may be required by some microcontroller to match the CPU's inverted TTL serial input (0-5v, 9600BPS); an application developer may change the baud rate, but 9600 or below is recommended.

VII: ELAINE: Sailing AI application

ELAINE's AI exists as an application riding on top of the NAVCOM AI OS, and consists of three parts: a sail angle function, a steering function, and a state machine to handle tacking. In addition to these, ELAINE features an auxiliary motor which can be turned on from the sail servo, and a bilge pump with automatic water level detection; for reliability, these systems exist on their own power circuits and are electrically isolated from everything else; in addition, the bilge pump's electronics are doubled – in case of catastrophic failure, ELAINE would still be able to guarantee its survival.

The sail-angle function commands the sail servo to keep an optimal angle between the sails an the hull in order to produce thrust and move the boat forward; when climbing against the wind, the sail acts as a wing and transforms its “lift” into thrust; when the wind is coming from the boat's broadside, the sails act more like funnels, redirecting the wind's direction towards the rear of the boat and propelling the boat forward by reaction. Finally, when traveling with the wind, the sail acts as a parachute, and drags the boat along. In the first mode of operation, drag is actually a major liability, tending to push the boat off course and tilting it on the longitudinal axis (which requires a centerboard and a counterweight to prevent the boat from capsizing); thus, while only one parameter is being changed – the angle of the sail to the hull – it must be controlled with great care for optimal results. In addition, once the boat gains speed, the relative wind with respect to its hull changes in both direction and velocity (the wind's absolute speed and direction must be summed vectorially with the boat's), requiring continuous adjustment even with constant weather condition if optimal speed is to be reached.



The only way to find a workable mathematical formula tying wind direction to sail angle was by trial and error, a process which must be repeated every time the boat's configuration changes significantly; for ELAINE's hull, a valid relation was found to be

$$S := W^2 - \left| \frac{\left(180^2 - T^2\right)}{(180 - T)^2} \right|$$

where S is the servo swing, W is the relative wind direction, and T is a predetermined angle (the tacking angle) which, for ELAINE, was found to be approximately 40 degrees during normal conditions – this means that ELAINE can climb up to the wind as long as the angle between the wind's source and her bow is less than forty degrees.

This equation can be overridden in three cases, in order of priority: if the clinometer detects that ELAINE's mast is bending too far from the vertical, sails will be allowed to slack somewhat in order to reduce stresses on the mast, eliminate the risk of capsizing and allow the compass to keep working properly. If ELAINE's speed is less than that which would be achievable by the onboard motor, the sails are reeled in and the motor is turned on instead – this allows ELAINE to detect being effectively becalmed even with no wind velocity sensor. Finally, if a waypoint is closer than a specified distance, the motor is employed for the final approach due to the need for maneuverability – essentially all modern sailboats dock under power, and even the galleons of old often had to be towed by rowboats to their docks if wind conditions were less than perfect for an approach under sail.

All the numerical constants described above are associated with letter variables, and can be modified on the fly through the XBee radio modem, as follows:

A Tacking angle
G Result
Q Approach distance

Invoking the AI function sets the default tacking angle to 40.0 and assigns the variable G to servo 2 – adding or removing a fudge factor can be accomplished by the command **@E2 G** *fudge* **+** through the XBee.

The steering function must take into account the fact that a sailboat's heading (the direction of movement) is not necessarily related to its bearing (the direction at which the bow is aimed) due to current drift and the sideways pressure exerted by the wind on the hull and sails during most wind conditions; this forced the use of a differential approach can't be readily expressed with a mathematical function.

The NAVCOM AI OS calculates, based on GPS and sensor input, how much to turn to face its intended direction (which normally matches the bearing to the next waypoint); this value is assigned to the **U** variable, and its differential with respect to time to the **u** variable. After initialization or reaching a waypoint, the rudder is centered (thus assigning a value of 1500 to the servo pulse generator); this value is added to or removed from depending on **u** every AI frame, following the formula

$$J_{i+1} := J_i + \left[\left(\frac{d}{dt}U\right)\cdot F\right]$$

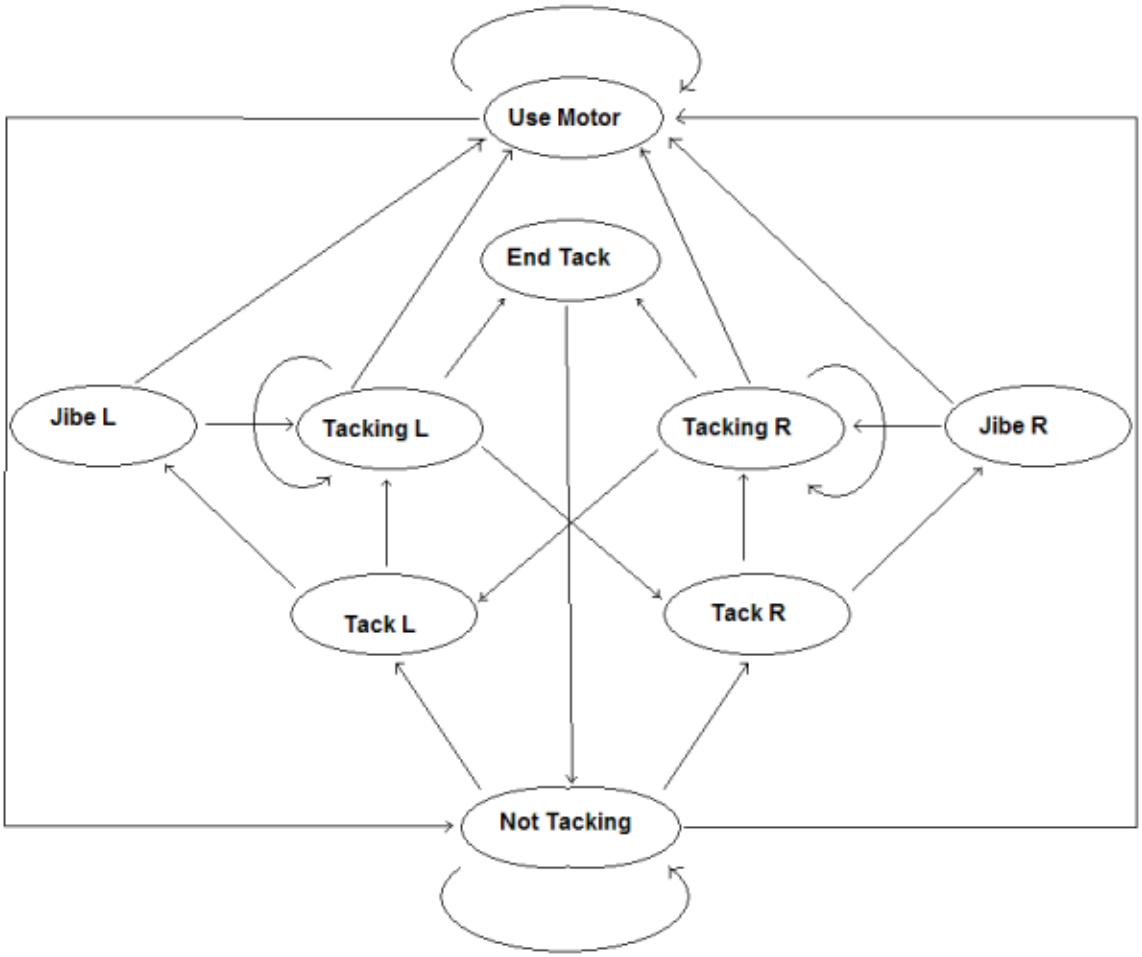
where J is the value of the servo pulse for that cycle – essentially, the AI uses its delta variables to solve a differential equation, with F as a fudge factor.
To prevent overcorrection which would result in “drunken sailing”, the rudder is left to its current position if **U** is less than a specified value; furthermore, the overall value of the servo pulse can be capped at less than full swing by using another variable. As above, numerical values are assigned to letter variables and can be changed via commands, as follows:

V The result of the equation, which is assigned to servo 1 (this can be modified by sending the **@E1 V** (fudge) (operation) command).
JThe value of the rudder angle during the previous cycle
uThe first derivative of U
LThe primary fudge factor
M The allowed swing range of the servo
ZThe “dead angle” as far as steering is concerned

These values would change with hull configuration (and in a smaller measure, weight), and the defaults are significant for ELAINE's particular shape and size; this equation can also be considered a useful generalized rudder equation, and is included in the basic NAVCOM AI package for application developers to use as an example.

The tacking state machine allows more efficient navigation. By having ELAINE simply turn on the motor when its bearing to a waypoint would put her against the wind, it's possible to have basic sailing with just these two functions; tacking maneuvers require a computation layer above this. The NAVCOM AI OS saves the great-circle bearing to the next waypoint in a variable, assigned to the letter **P** by default; likewise, the tracking expression is initialized to this variable (equivalent to the command **@ET P**) to allow for moving to the shortest route. By modifying the tracking expression, ELAINE can be directed to take a different route without having to interrupt the workings of the rudder servo (and thus keeping the modules independent, allowing parallel development and better

reliability). A simple state machine is implemented to allow ELAINE to decide when and how to tack.



Tacking finite state machine

The default state is “not tacking”; the decision to tack is made when normal navigation would bring ELAINE to face the wind at less than the tacking angle (the same variable is used for the state machine and sail-angle function); the direction depends, obviously, from which side the wind is coming from. In that case, the state machine sends a **@ET W A -** (left) or **@ET W A +** (right), which

translates as “Set course to the heading that will cause the boat to climb against the wind at the minimum feasible angle”. If this does not happen in a a specified amount of time, the boat will attempt to jibe instead – that is, turn 270 degrees or so as to move with the wind for a few moments and gain sufficient momentum to complete the maneuver.

Once tacking, the boat can either stop the tack if the wind changes and return to its normal status, or change tack after a specified distance has been traveled or a specified time has elapsed, whichever comes first; the AI will attempt to change tack, helping itself by turning the motor on while facing the wind directly (this operation exists on a separate line, and can quickly be commented out during a race) and jibing if necessary. The approach distance is checked by the state machine as it is checked by the sail-angle function, and forces the state machine to remain in a non-tacking state during approach.

A future development will include the ability to decide how often to change tack based on allowed crosstrack error (the size of the “corridor” between waypoints in which the boat zig-zags during tacking), by having the tack change when the direct-route tracking becomes a specified fraction of the tacking angle: having the ratio be 1/1 would cause one tack between the two waypoint, have it at ½ three tacks, ¼ four tacks and so on. This has not been implemented due to the need to stick with a simpler, more reliable algorithm for this prototype.
The letter variables used by the state machine are:

A Tacking angle
Q Approach distance
XMaximum length of each tacking leg in meters
YMaximum duration of each tacking leg in seconds

VII: Test Results

A number of tests were performed during the design process. Deployment of ELAINE requires two people and approximately 15 minutes to pack up the boat and equipment and drive to Woodlawn Lake. Early secondary tests performed without witnesses (requiring approximately 25 minutes for deployment), generally for the purpose of refining fudge factor before the ability to change them on the fly was implemented, were not recorded. Additional information can be found in the development diary.

Early Testing – Summer 2006

Since the development of ELAINE and the NAVCOM AI proceeded in parallel, multiple tests were necessary at every stage of development. Preliminary tests were conducted in the summer of 2006 using a Picaxe-28 microcontroller and a FPU chip, on a RC car to test basic navigation and on ELAINE's hull to test seaworthiness – the relatively small processing ability of the Picaxe, even augmented with a floating point unit, caused the car to either oversteer or understeer with little capability for correction; the form factor for the prototyping board this system was built on, however, proved seaworthy and defined the final dimensions for the NAVCOM AI board.

Seaworthiness / Telemetry Test 1 – August 2006

This test was conducted using the hand-wired NAVCOM AI prototype board, primarily to test its seaworthiness and to see whether the wind information was being relayed to the base station correctly; no servos were connected to the board, and the GPS, while on board, was left disconnected during most of the run.

Overall results were positive, proving that the basic concept was sound – at this time, ELAINE carried two front-mounted sonar sensors; those functioned for a short time before sustaining water damage. The need for a bilge pump was realized.

Navigation Test 1 – October 2006

A simple navigation algorithm was developed and tested out, but proved to work only when the boat has the wind in its back – the relevant information was saved in case the need for developing a powerboat AI arises.

Seaworthiness Test 2 – October 2006

The purpose of this test was to check the seaworthiness of the new professionally milled NAVCOM AI board, and give an initial check to the basic navigation algorithm. Water damage once again prevented use of the sonars, and the navigation algorithm proved insufficient to deal with the mixture of forces

that a sailboat under way is subject to. Some water damage was sustained by the electronics due to an underpowered bilge pump.

Navigation Test 2 – February 2007

A different algorithm was tested and found to be sound in principle, but lacking in practice. What was decided to be the final version of the bilge pump performed satisfactorily.

Car Tests – January / March 2007

A RC car proved of some use during calibration due to the ease of deployment; unfortunately, the low-end nature of its steering system prevented accurate control even when driving it manually. Lessons learned from the car tests included how to interpolate between GPS and compass and how to deal with an uneven power supply (at one point a 1 Farad double-layer capacitor was fitted to the sensor power circuit!) and magnetic interference to the compass from other components; the car was also used to test the reliability of the serial telemetry system.

During this period, tests related to the Ithuriel projectile drop project were run, consisting of distance measurements taken from a moving full-sized truck. These tests allowed identification of a loss-of-sync problem existing with our GPS, and its correction.

Telemetry Test 2 – March 2007

The final version of the telemetry section (console, relative/absolute wind direction, turn indicator) was tested with satisfactorily results, to the point that it was possible for me to steer the boat by telemetry. The sailing AI was activated and allowed to control the sail servo, with rudder still in human hands, and found to be performing satisfactorily albeit slowly; cross-interpolation of wind sensor readings with compass sensor readings improved the response speed noticeably.

Comprehensive Test – April 15, 2007

This test confirmed the ability of ELAINE's state machine to perform tacking maneuvers correctly; the AI was led out into the lake under manual control, and had to return home by itself. After a disappointing run due to oversteering, the correct value for the various fudge factors was determined and programmed in.

The next run saw ELAINE switch to autonomous sailing, turn on itself twice, then head for home executing a tack approximately halfway through, completing its approach 3 meters from the target.

Shortly after returning to autonomous control in the final run, a gust of wind blew away ELAINE's wind sensor; despite having never been thoroughly tested, the error correction routine within the sensor parser kicked in, discarding any further input from the wind sensor and estimating wind direction from the last

valid data and the boat's current bearing. Despite tightening its sails a little too much, ELAINE headed home slowly but surely, completing its approach 1.5 meters from the target.

Comprehensive Test – April 25, 2007

The values derived from previous tests were coded into the AI application, rather than sent as commands; for this demonstration, ELAINE was given a waypoint, manually steered away from it for a short distance (3 meters), told to navigate away from it by modifying the tracking function and told to return to the starting point after having reached a distance of approximately 40 meters.

The test was generally successful; a minor problem came about due to the strong wind during the test, which tilted the boat considerably – although the failsafe functioned properly, the boat spent enough time at a wide angle from vertical that the bilge pump hose exhaust found itself underwater, becoming a siphon. Although ELAINE returned home under its own power and intelligence, the water inside got dangerously close to the electronics. Photographs were taken.

VIII: Summary and Conclusion

The NAVCOM AI has proven its potential as a general-purpose navigation computer, and demonstrated actual commercial potential if used as a telemetry device (during both boat and plane tests, commercial availability was inquired about). While real-world navigation is much more complicated than what the current incarnation of this platform can handle, the ELAINE project demonstrated that automatic sailing does not require prohibitive processing resources.

Further revisions of the NAVCOM AI will include the ability for the NAVCOM AI to make an educated guess as to rudder fudge factors by observing the vehicle's behavior when in telemetry mode; a telemetry-only board is planned, as well as a surface mount version for smaller vehicles. In the longer term, the ability to read (and learn from) incoming servo pulses from the human pilot is planned.

Further revisions of the ELAINE project will include enhanced tacking ability based on crosstrack error and the possible replacement of the current board with a SMD version; ELAINE has proved to be too small for a development platform for future NAVCOM AI revisions, and will likely be donated to the University so that others may use it for their senior designs – particularly interesting would be the addition of solar panels once a smaller control board is engineered. The name ELAINE will be kept for further sailing-related developments in autonomous navigation.

Autonomous robotics is today in the state where personal computing was in the late 1970s: the field is wide open for exploration and opportunities are plentiful. It is entirely possible that efficient automation and renewable energy technologies, combined to produce a class of “electronic animals” built and trained to augment human capabilities, may eventually result in an economic paradigm shift comparable to the advent of the Internet.

Matteo K Borri
April 16, 2007

Appendices

What follows is additional information that may be useful for AI application developers and original equipment manufacturers should they choose the NAVCOM AI in order to implement their own project; the author is going to be available through email for clarifications and updates.

The development diary has been left in a more-or-less unedited state in order to better reflect the development process; a few ideas in it were not developed into the final product, mostly due to time constraint, and may be useful to developers who wish to extend the platform.

Appendix A – Costs

Item	Company	Cost
Electronics		
Parallax Propeller microcontroller	Parallax Inc.	13.99
Picaxe-08M microcontrollers (x5)	Revolution Education	15.00
Garmin eTrex portable GPS unit	Wal-Mart	80.00
MEMSIC 2125 Accelerometer	Parallax Inc.	
Hitachi H55b hall sensor	Parallax Inc.	
Voltage regulators (x5)	JameCo	7.50
Glue logic (multiplexer, transistors etc.)	Saint Mary's University	Available
Electrical		
Standard mini servo for rudder	HiTec	5.95
Large ¼ size servo for sails	HiTec	34.95
1¼-inch DC electric motor (x2)	Radio Shack	6.00
Wiring, battery holder, microswitch	Saint Mary's University	Available
NAVCOM AI Motherboard (milling/pressing)	PCB Express	59.95
Passive components and connectors	Saint Mary's University	Available
3300mAh Ni-MH main battery	Radio Shack	Donated by manager
Mechanical		
Assorted LEGO plastic parts	Toys R Us	11.95
Assorted wooden boards	Lowe's	14.95
Waterproof sensor covers	Walgreen's	Free (old photo film covers)
Support Equipment		
XBee-Pro Demonstration Kit	MaxStream Inc.	29.99
XBee Demonstration Kit	MaxStream Inc.	19.99
TRS-80 Model 100 laptop computer	Found at a garage sale	5.00
		NAVCOM AI Total:
		250.24
		Grand Total:
		305.22
---	---	---
Development time		10 months
Cookies eaten during development		86
Hours of sleep lost		Too many
Boats sunk during development		0

NOTE 1: This only includes the cost of components that are actually present in the final design. As with many prototyping projects, a number of components were damaged and required replacement, and some subsystems were designed and tested but removed from the final implementation.

NOTE 2: ELAINE's hull was self-built from plans over the span of five months and represents a “labor of love”, a factory-assembled complete R/C sailboat of comparable size costing anywhere between $500 and $1200 new.

NOTE 3: The “Company” column indicates the company that the component was purchased from, not the company that manufactures it; for all significant parts (thus excluding generic parts such as passive components) datasheets are available.

APPENDIX B – Development Diary

Summer Progress:

- January through April - The “naval engineering” part is taken care of through the construction of a 1-meter remote controlled sailboat. Additionally, a suitable electric RC car is found and adapted.

- May - Experiments with interfacing a GPS to a PicAxe microcontroller start. While the PicAxe can handle waypoint and basic navigation, it proves to be too slow for the task. PicAxes are redeployed in a secondary role, to provide an unified NMEA-like interface between the sensors and the main processor.

-After much research and trial and error, a microprocessor was decided upon- the Parallax Propeller. After the HCS12 was ruled out because of the difficulty to interface and inability to find a C compiler, Chrissy explored simpler micros like the Basic Stamp 2, CB220, and Basic-X 24. These were too slow and too small to handle our needs and could do no higher level math. So we decided that a Picaxe 28 with a math co-processor would work; preliminary tests showed it could deal with the boat, but couldn't give sufficient precision such as the plane would require.. Then I discovered the Propeller. This could do the higher level math, so there was no need for a co-processor. Also, it is much faster that the PicAxe.

- June - Tests using the car as a platform. Basic navigation principles hold and a decent accuracy is obtained via interpolation.

- July - The Propeller language (SPIN) was learned, and a navigation-oriented math library written for the microcontroller

- August - First sailing tests, first with the Picaxe and FPU then with the Propeller. The latter proves to be notably superior.
-
- A prototype motherboard for the AI was designed and built. It handles 4 sensors, 4 servos, a battery level detector, a radio switch and a radio modem. This was later expanded with a daughterboard socket and the ability to broadcast a TV signal.
-

Weekly Progress- Fall 2006:
(weeks dated by the Friday of each week)

September 1

- Minor maintenance done on the hull (re-enameling of exposed wooden parts mostly).
- Replaced sail arm servo with a model that has less speed but much greater torque.
- Updated math library for better precision (for speed, we use fixed-point arithmetic rather than floating-point).

September 8

- Was too busy trying to make rent to get much done. Started design on the final PCB layout.

September 15

- Did data recovery for Dr. Ibaroudene.
- Continued work on PCB layout.
- Came up with generalized formula for servo movements compared to sensor inputs -- it's too slow at the moment though.

servovalue1 = constant + eq1A.sv1 + eq1B.sv2 + eq1C.sv3 + eq1D.sv4 + eq1E.diff_ht + eq1F.diff_bh + eq1G.dist + eq1.prevvalue

where:

constant is the base value, for example, in case of a rudder the base value would be 150 (center).

eq is a third degree equation (this allows for using a cubic-spline interpolation), which I need to write a faster implementation for; the coefficients can be changed on the fly via the serial communication channel. If I learn enough about neural networks, I can
come up with a way to "learn" to adjust them -- I have 2 processors free, so 1 could be doing this. Note that the coefficients can be stored as a sparse matrix, as most of them will be zero (e.g. in case of a sensor having nothing to do with a servo, for example
the sail servo should only deal with wind)

sv is a sensor value

diff_ht is a normalized (least circle arc) difference between heading and tracking

diff_bt is a normalized (least circle arc) difference between bearing and tracking

dist is the distance to the next waypoint

This should give me a solid software framework for training (either by supplying the coefficients after seeing how the AI does in a test run, or by having it detect under/overcompensation and change in consequence) the AI module, after which God willing the software side is done.

Hardware side, I must build a small VHF transmitter for the TV signal, make the sensors waterproof, and possibly have the AI board printed professionally.

October 6

Rebuilt sonar after water damage in test run; added anti-splash barier.

Currently rewriting math libraies to take advantage of the 2 unused processors.

Processor allocation is now:

0 - Initialization and main program
1 - Servo waveform driver
2 - Serial port driver
3 - TV signal driver
4 - GPS parser
5 - Sensor parser
6 - Math processor 1 (basic math)
7 - Math processor 2 (trig)

Currently helping Chrissy out with programming in Spin.

Decided that an extra compass sensor is needed to check against while performing sharp turns; sensor has been ordered. Investigated accelerometer for this purpose, but a compass sensor is a more realistic bet.

October 13 - 20 - 27

Designed, ordered, received and populated the new mainboard -- thanks to Chrissy for the help with the voltage regulators. Testing of said board continues. Layout and schematic are on file (Multisim-Ultiboard 9 used, board fabricated by ExpressPCB). This took quite a long time.

Built control terminal out of a radio transceiver and a TRS-80 laptop (vintage 1983) -- better than modern laptop in that batteries last longer and display is easier to read outdoors, and we only need textmode anyway. It looks rather cool.

Rewrote math library AGAIN for what I hope is the last time -- the 2 spare processors are allocated dynamically as coprocessors, first-come-first-serve. This is acceptable because doing things this way is still 2x-15x faster than having each processor do its own math, especially with trig.

Built and tested compass sensor module; still need to decide where to mount it on the boat. Begun to write parser for the compass.

Moved all sensor xmit speed from 4600 to 9600 baud, since they can handle it with no errors. Now every serial comm in the AI module is at 9600.

Decided to use temporary waypoints for the tacking AI, rather than overriding the main
waypoint-track function -- this generates cleaner code.

Debugging "plug and play" sensor parser -- it still freezes the main processor on init if no sensors are connected. Interestingly, the dedicated processor goes along just fine. Not sure why.

Processor allocation is now:

0 - Initialization and main program
1 - Serial port driver
2 - Servo waveform driver
3 - TV signal driver
4 - Sensor parser
5 - GPS parser
6 - Math processor 1 (dynamic)
7 - Math processor 2 (dynamic)

November 2

Tested new math library for precision and speed, and made sure that a function cannot just take all the FPU time -- simple round-robin based on the common internal clock. Started adding the new math code to the actual program -- I'm about half there. Wrote simple fixed point arithmetic display routine for TV telemetry. Figured out how to save some memory with my strings.

Found & ordered decent-looking servo plugs and cables so I can make them the right length, found & ordered parts for TV amplifier, reduced instances of dropped serial characters on COM port via adjusting the transmit buffer. Checked operation of "reflex" systems, i.e. motor and bilge pump -- bilge pump may need rewiring. Cleaned up wiring in general.

Tested battery endurance (~ 8.5 hours with no servos, wow!).

Still waiting for letter for Stanford :(((

Processor allocation hasn't changed.

November 3

(This was all done on Friday -- the memory management change is signficant, so it needs
to be documented by itself)

Changed memory management in order to simplify it -- specific data now reside at fixed addresses which are accessible by all CPUs. A lock will be added to make sure there are no overwrites in case of multiple sensors of the same type.

The AI will constantly operate in tracking-matching mode, with the tracking vector being either derived from current and waypont position, or overridden either manually or by the sailing AI in case of adverse wind or obstacles. The override value can be a function of the calculated value, e.g. for use in a zig-zag maneuver or to go around a moving obstacle (in case of the model, ducks or swans). This generates a "steer by" value which will be used to drive a rudder servo.

Writing math libraries is less scary than I thought it would be.

Processor allocation hasn't changed and should now be considered stable, since it works.

November 4

Shaved a few microseconds off trigonometry execution.
Trig functions now understand degrees natively (i.e. don't have to go back and forth with radians)
Started adding 3D support.

Redesigning bilge pump assembly -- the one I have is too big. What to do about it?

November 11

New bilge pump seems to have problems, so I built a third one -- this one is held in place with velcro and is less of a snug fit. Now using a 74HC14 for triggering the FET which drives the motor. Bilge pump is now independent of the AI.

November 18

Considering writing a simple BASIC-like language for control of the AI instead of having AI functions. Read up on PicoBASIC and tried to write an interpreter for the Propeller with little success.

December 4

Delivered presentation with reasonable success. Still working on documentation (wrote the first draft of the AI command list).

Weekly Progress - Spring 2007:
(tasks written the day they got finished)

January 19

Discovered that over break the sail servo had a seizure and rebuilt it -- will still order a new servo and keep the rebuilt one as a spare.

January 22

Partial rewrite of the GPS parser after car tests determined that sudden accelration threw the timing out of whack due to the GPS not responding when it should have. Now using full duplex serial for both GPS and xbee.

January 30

Added speed settings to math library (lock/unlock and forceslow/allowfast). Performing a lock during interpolation allows up to 80 interpolations per second! Clamped at 75 with TV off and 60 with TV off to be safe.

February 1

At the request of the airplane team, began work on a telemetry console that could display data graphically in case the TV output is unusable. This will be a windows application written in visual C or visual basic.

February 10

Made a subtle modification to the plane's AI function that would have caused a major bug if the plane is being piloted by hand and not exactly aligned with the target (target drop zone is now a line rather than a circle). Hopefully Chrissy won't notice because I don't want to tell her that her logic was wrong.

February 12-15

Major software rewrite -- the memory map is now allocated entirely statically. While this is ugly from a software engineering standpoint, it's also faster and takes up less memory. Wrote "execute command" routine to allow premade commands.

February 12

First version of the console seems to communicate OK with the AI, but it loses data packets on occasion. Visual basic was used, which may be part of the problem -- it's too high-level and I cannot communicate with the serial port the way I want to.

February 20

Finished the console -- I had to basically write my own serial port buffer driver, but now it works properly and displays information as it should. Added embryonal moving-map support (piggybacking on

google maps). Steve Cerwin showed me a HSI instrument and asked me to copy the look of it. I was asked to add "virtual lights" by Chrissy.

February 22

Moved a large number of commands to the ExecuteCommand format, which has been streamlined. Considering writing my own scripting language for the NAVCOM AI instead of using expressions. Quick AI test.

February 26

Released what I hope is the final version of the console, it can also generate GPGGA strings for use by external moving map software, if coupled with a virtual serial port pair. Steve is happy with the design of it, and it works very well for the boat too (turn off altitude and turn on wind speed). Virtual lights added and functional -- now I can go back to real work. TV output is now off by default as it seems that the console does the job just fine.

March 5

Ran first telemetry test with mixed result -- water comes into the boat and shorts
everything; bilge pump not effective. Sonars too exposed to water. Telemetry worked
until the NAVCOM AI got flooded -- no damage, fortunately. Wind sensor vane was totaled during transport. The boat is too small for this sort of work.

March 6

Added WC and WD command and revamped WS commands to allow entering waypoints manually, this should make life easier for both boat and plane. Came up with theoretically effective sail-position algorithm which will be tested during the next outing. Begun design on tacking algorithm.

Decided against a scripting language. However, support was added to map any letter variable to any navigation parameter on the fly (or to nothing at all if so required). This can be used to quickly change fudge factors in formulas. Added support for running a finite state machine inside the main AI function since it will required for tacking.

March 9

Rebuilt bilge pump assembly to my satisfaction -- I am no longer using a 74HC14 for
water level detection, the FET is being driven directly with no adverse effects. Wind vane was rebuilt -- new vane is slightly less precise, but more accurate and a lot more robust for the same weight.

Rewrote the sensor parser almost completely to take advantage of being able to read serial pin voltages directly and using that to allow on-the-fly adding and removing of sensors; this should simplify things considerably if a sensor momentarily comes loose. Side effect: The system takes a lot less time to boot, from ~7 seconds to ~4 seconds depending on connections.

March 10 - March 19

Spent all week working on the IEEE website. Got very mad at Sebastian for not doing his job. If he goes home with IEEE webmaster on his official transcript for this semester and I don't, I will consider that as permission to prank someone over it for contempt of reality.

March 20

Spent a few weeks rigging up a proper servo controller for the car and ran a few tests;
telemetry works effectively, but controlling the car proved unrealistic due to uneven
steering response and the fact that the electric motor acting as its prime mover generates
enough of a magnetic field while running that the compass becomes hopelessly confused
even if mounted at the other end of the car. Good response from sonar sensors, however,
the car will be abandoned as there is no time for it and it's not giving me useful data
for the boat. Next time I'll get a RC car from somewhere other than Radio Shack and
costing more than $30.

March 23

Built guides inside the hull for the mainsail and jib lines, to avoid said lines getting tangled
in wiring. Removed front-mounted sonar. Waterproofed bilge pump electronics. Considering using smaller battery.

Added "clamp" and "twirl" operations to the expression parser after noticing that they made the car slightly more controllable. Realized the possibility of setting variables through the AI function and feeding that variable to the servos as an expression, which should allow for quick fudge-factoring. Added trim commands, for same reason.

March 27

Ran second telemetry test and first sail servo test, both with partial success. Standard
analog transmitter is of poor quality, however serial telemetry has excellent range. Sails
work as intended, which is a big relief.

Built waterproof enclosures for compass and front-mounted sonar (which now doubles
as a clinometer); this allows the compass to sit on deck farther away from any other
electronics, and thus get better magnetic exposure. Figured out how to put all the batteries inside, but a smaller battery would still be useful to have.

March 30

Built a small "demo box" for the AI to display at the research symposium; people were generally interested and one of the organizers gave me a $25 food coupon for some reason. Yay free food!
Posted spoof page on IEEE server which will be removed tomorrow.

April 3

Determined structure of tacking-algorithm state machine and devised formula to determine numbers of tacks for a given maximum crosstrack error between waypoints in such a way that the previous waypoint doesn't need to be memorized. Started coding for it after successful simulation within Vegastrike videogame engine; the logic is sound, all I need to do is make sure the rudder responds appropriately.

April 6

Paused development to help with the airplane project; plane test did not happen because

of crosswinds. Hacked into a website's records to figure out a phone number for an alternate takeoff site, but it also was in the wrong orientation. Notified website owner of breach & of the fact that his phone number was not accessible normally, so could he please put it on the page. Was thanked for it.

April 11
Navigation test. Mechanics OK (bilge pump is a power hog, but it's on its own battery, and it works) , telemetry OK, basic navigation works in principle but is very inefficient – on occasion the boat will move in circles. Must refine compass interpolation and steering function. Tacking logic OK, but only tested while piloting manually. ELAINE seems to be attracted to a particular tree; in respect of this, may set my end waypoint there.

April 13
Plane test went OK, in the sense that my part in it worked perfectly and I have kept my obligations. Further refined the steering function and state machine.

April 15
Navigation test allowed me to refine my fudge factors to correct oversteering; after one unsuccessful run, the boat performed very well in two runs, returnig home without problems. During the second run, a gust of wind removed the wind vane from topmast; the NAVCOM AI correctly extrapolated an estimated wind direction from the last data received and its own position, and returned home.

April 16
A new wind vane was built and is currently being calibrated. Minor water damage found within the AI board was repaired; during last test, the centerboard was snagged and sustained some cosmetic damage at the root that will be repaired as time permits.
Finished writeup for the ELAINE project, pending subsequent tests.

April 18
Demo shown to Dr. Aminian; boat performed considerably well in simple two-waypoint navigation. Video was not taken because there was little advance warning for this demo (originally scheduled on Apr 20, performed today due to unexpectedly good weather).

April 25
Similar demo shown to my family; boat performed solidly in stronger wind, and inclination sensor proved its usefulness in avoiding capsizing.

APPENDIX C: NAVCOM AI TUTORIAL – SAMPLE USAGE

Upon startup, the AI will initialize itself and wait for a valid GPS fix from GPS #1. It's possible (and advisable) to activate the AI without any external components connected in order to try out a few commands. In this state, the AI is unable to start the main AI function, but can still execute commands – including the RPN calculator – and update servo positions.
A good place to start is to familiarize with Reverse Polish Notation: try out the calculator emulation function by entering commands such as

@? 2 2 +
@? 10 2 * 5 +
@? 10 2 / 5 + 2 ^
@? 9 $
@? -9 $
@? 9 - $

and examine the results.

Now connect a servo to the servo output 1.
Enter the commands

@E1 1
@E1 0
@E1 -1

and note the servo's movement; experiment with fractions.
Most non-embedded GPS units, such as the Garmin eTrex, have a demo mode; turn the GPS on, and put it in demo mode. Notice that the AI acknowledges a valid satellite fix (it is unaware of the fact that it's simulated, and will accept the data as long as they are in the correct format).
A message stating that sat signal quality is OK should appear within the next second; the system is now fully functional. If you have a TV available, consider setting it to broadcast channel 3 and turning on the TV telemetry with the **@TVB** command – having the TV antenna near the motherboard should be sufficient for getting a readable signal. If you have a RCA composite cable, you can connect it to the TV and the AI board in the “composite out” pins and use the **@TVC** command instead; either way, a screen full of textual information should appear.

A quick way to establish a waypoint is to use the **@!** command; this saves the current position in the next available waypoint, then sets it as the current waypoint – this is a “man overboard” feature common in most nautical GPS systems. Enter that command, and wait some time to allow the GPS's demo mode to “walk away” from the waypoint – if you do not have a TV connected, you can use the **@? D** command to get distance information; the ditstance should be steadily increasing. **@? S** will return speed; if you have a TV connected, note how the information acquired his way matches that on the screen.

Once the distance has increased to a high enough value, try some simulated maneuvering: if you are using the NAVCOM AI console, click once on the compass rose and use the Demo panel of your GPS to generate a simulate heading and speed – note that any compass or gyro sensors that are connected to the AI board will not be returning meaningful data, so there will be a 1sec. delay between entering a heading on the GPS and seeing it on the console.

Click on the Distance label to show the graphical distance readout, then enter the **@TSS hbsdnlJ** command to signify that you want telemetry to transmit heading, bearing, speed, distance, nav point number and the current coordinate pair; finally, enter **@TSN 1** to limit telemetry to once per second (since simulated GPS is being used and other sensors are not active, this is the best precision available right now, and more frequent telemetry would just repeat the same data). If you are using a standard terminal, note how all telemetry items save for coordinates are human-readable; the decimal point is implicit before the last digit.

Now enter the **@!** command and note how the nav point number and distance change with the next telemetry packet – distance is now very low but increasing, and a new nav point has been selected. This command can be entered quickly even from a standard terminal, and is designed to represent “man overboard” -- that is, mark the current position and endeavor to return to it, so as to mount a rescue attempt; this is a standard feature of most maritime GPS units, and incidentally provides a handy way of doing simple point-to-point navigation. Wait a little to allow the GPS to virtually move away from the saved coordinates, then using the telemetry information (whether in text or graphic form) try to navigate back to that point and watch the display react.

If you have a standard servo available, connect it to the Servo 4 output at the bottom right corner of the AI board, and enter the command **@E4 U 180 /** -- then resume navigating within the GPS's demo mode; note how the servo will “steer” depending on what the difference between the current and ideal bearing to the waypoint is! If you want to be sure of this, mark the same waypoint on the GPS and on the AI and compare the results.

A practical way to do so is to issue the **@WD** command and copy its output into a clipboard (if using the console, we recommend the Macro 8 text area); this generates a **@WC** command to assign a specified coordinate pair to the current waypoint. Set a GPS waypoint and enter **@WD** at the same time, then the resulting **@WC** command after your demo has gotten away from the waypoint; turn on telemetry (**@TSN1** or use the **@TVC** or **@TVB** to get a TV display for composite output and channel 3 respectively) and note how the AI and GPS recommend you steer – then watch the servo do just that.

Now, enter the **@ET P 90 +** command, and try to follow the AI's directions: it will try to drive a counterclockwise circle around the waypoint, keeping at the same distance. The tracking equation defaults to **P**, the straight-line bearing to waypoint; by having the AI try to keep its heading at a 90 degree angle, you have defined a circle with the current distance as a radius. If you would like to spiral towards a target, replace the 90 with a smaller number; if you would like to invert direction, use a negative number 90 degrees or less; if you want to spiral away, use a number (positive or negative) 90 to 180. If you want the AI to move AWAY from a waypoint, just set the tracking equation to **@ET 0 P -** -- this can be useful in application when some waypoints can be designed at hazardous locations. *Feel free to experiment!*

Appendix D – Quick Nautical Terms Reference

Beam: The width of a ship.

Boom: A spar extending from a mast to hold the outstretched bottom of a sail.

Bow: The forward part of a ship.

Close Reach: A sailing mode in which the hull is moving at less than 90 degrees to the direction wind comes from. In this mode, the sail acts as an airfoil.

Jib: A sail situated in front of a mast, generally triangular and fastened at the two extremes of its leading edge.

Tacking:

Mast: A long wooden or metal pole or spar, usually vertical, on the deck or keel of a ship, that supports sails.

Centerboard: A metal or wooden slab housed in a casing or trunk along the centerline of a sailboat.

Tacking: The maneuver by which a sailing boat or yacht turns its bow through the wind so that the wind changes from one side to the other.

Under power: Said of a sailboat that carries an auxiliary engine, when the engine is being used for propulsion.

Under way: Said of any ship that is actively sailing under its own direction, as opposed to being docked or tugged by another ship.

More information can be found online at:
http://sitesalive.com/ca9697/misc/glossary.htm

Appendix E – Schematics

A full scale version of the NAVCOM AI schematics and production drawings, in various formats to ensure compatibility, is available at

http://67.15.245.144/portfolio/navcom_ai/schematics.zip

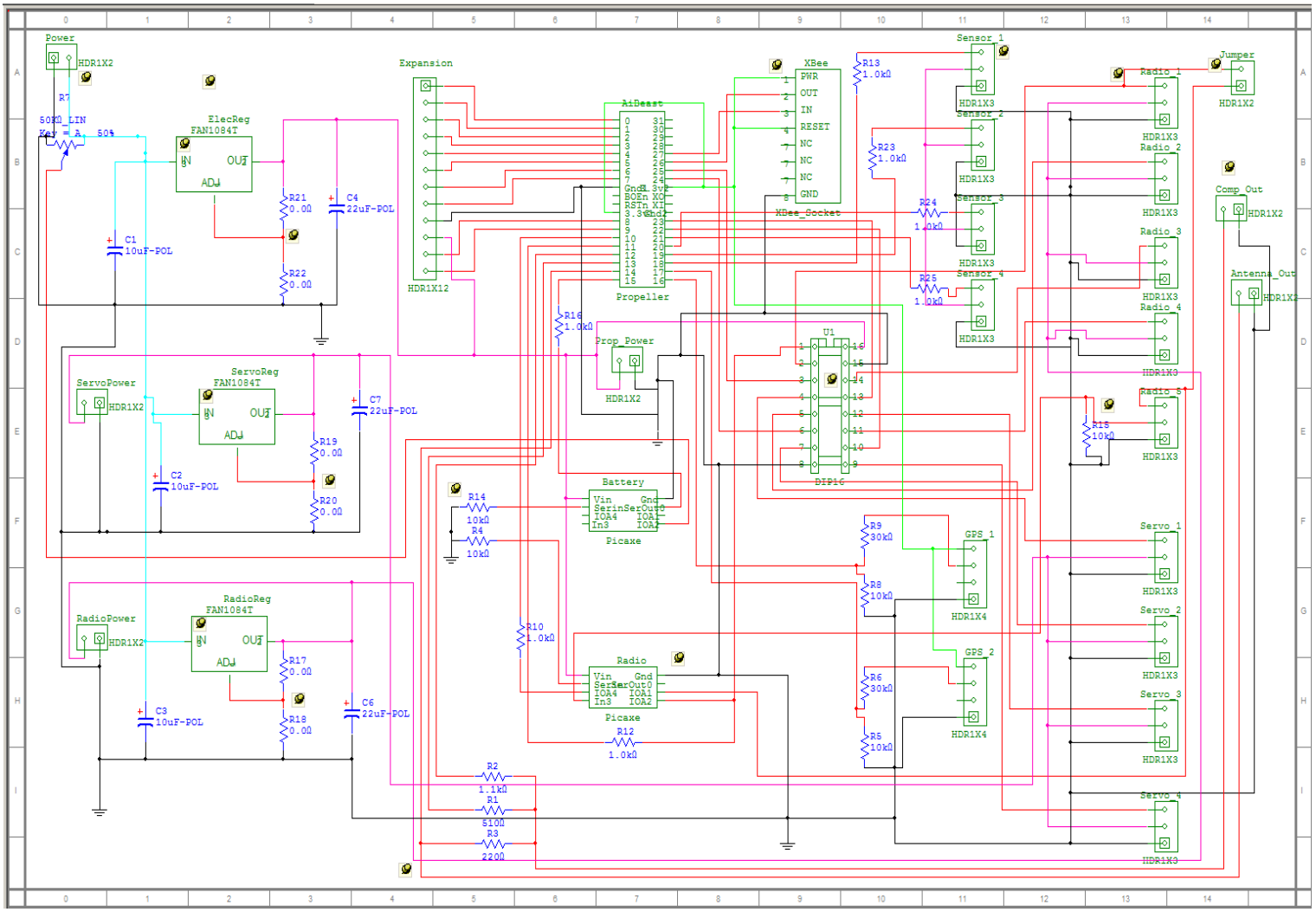
and may be used to manufacture copies of the device subsequent approval of the copyright holders.

Schematics will be provided in different formats (.net, Gerber, .pcb, .dxf etc.) upon request. The copyright of any improvement or modification for educational purpose reverts to Saint Mary's University.

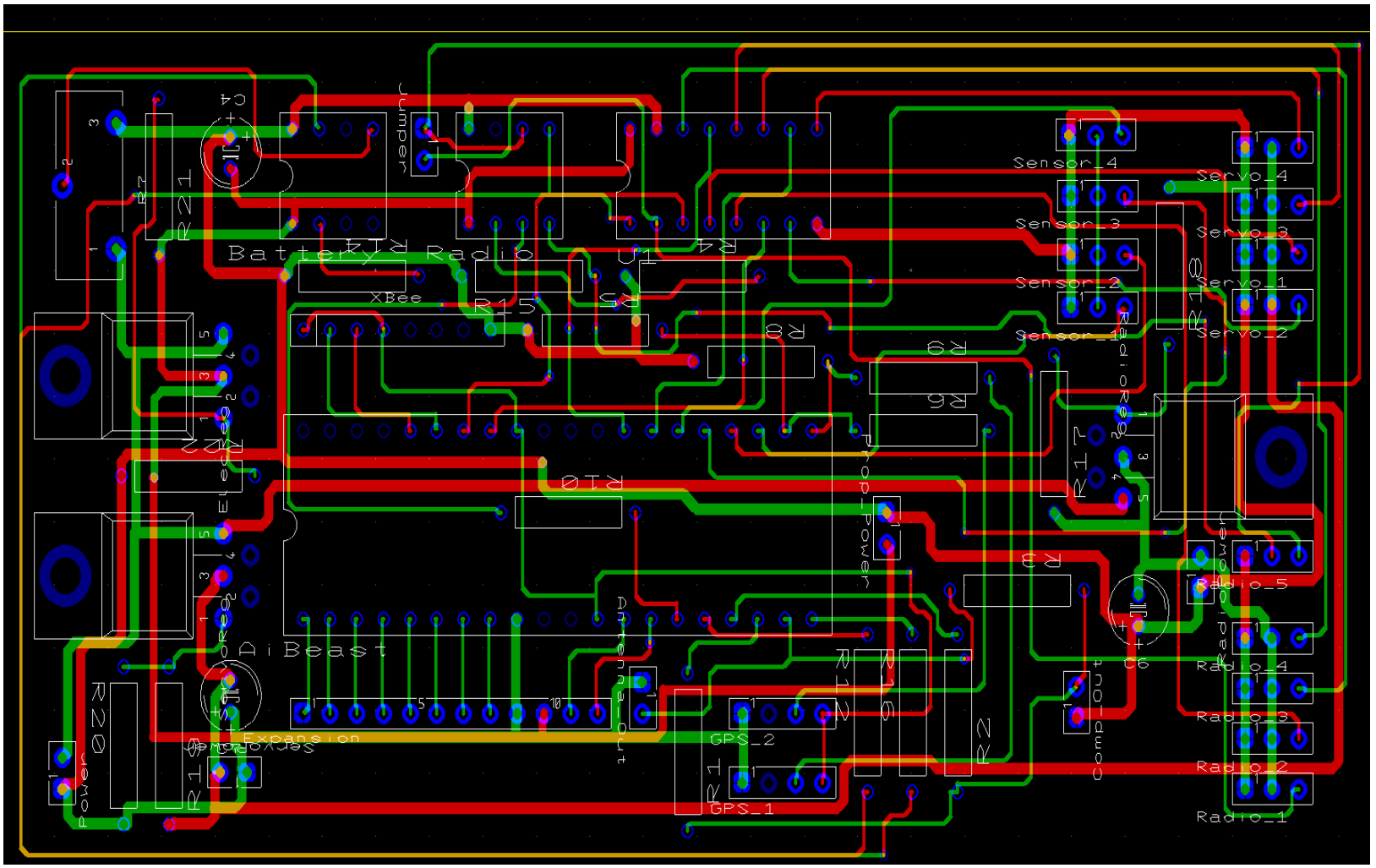
Low-resolution printouts of the schematic and working drawing follow, along with a short explanation where necessary.

The most current schematics set as of May 1st, 2007 is included in the CD that accompanies this report.

a. NAVCOM AI Motherboard



The NAVCOM AI motherboard acts as a support platform for the Propeller CPU and radio modem, and generates appropriate voltages for the operation of sensors and servos. This schematic was generated with Multisim 9.



This working drawing was generated and routed with Ultiboard 9; the physical dimensions of the device are exactly 2 by 5 inches, with a height of 1 inch at the tallest point.

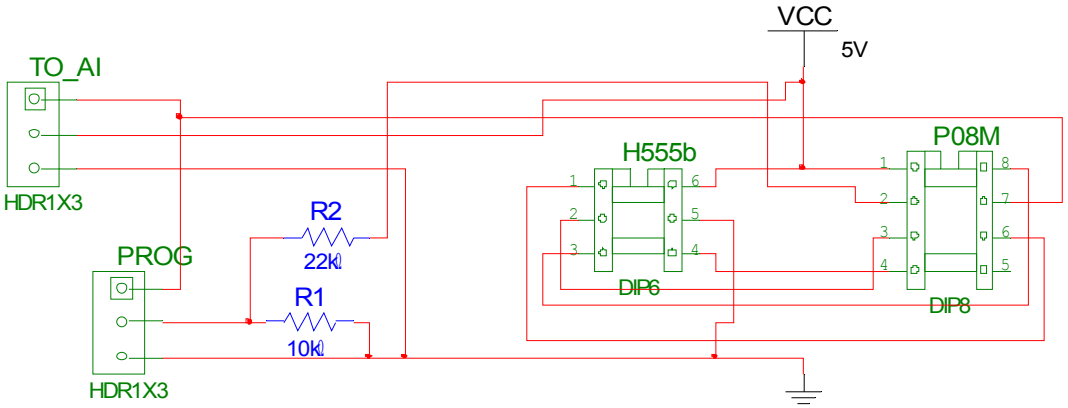
It is possible to replace all components but the radio modem and voltage regulators with SMD equivalents; shielding of the CPU (by means of a grounded conductive shell, such as a layer of aluminum foil) is recommended in order to increase the radio modem's effective range. If an amplifier is used for the TV signal – either composite or broadcast -- a minimum distance of two feet must be kept between the amplifier and the GPS – in this case, it's strongly recommended that the motherboard be kept close to the amplifier.

b. Sensor modules

Sensor schematics vary; the three external sensors used in the ELAINE project have been hand-wired and use the following schematics. It is worth noting that the wind

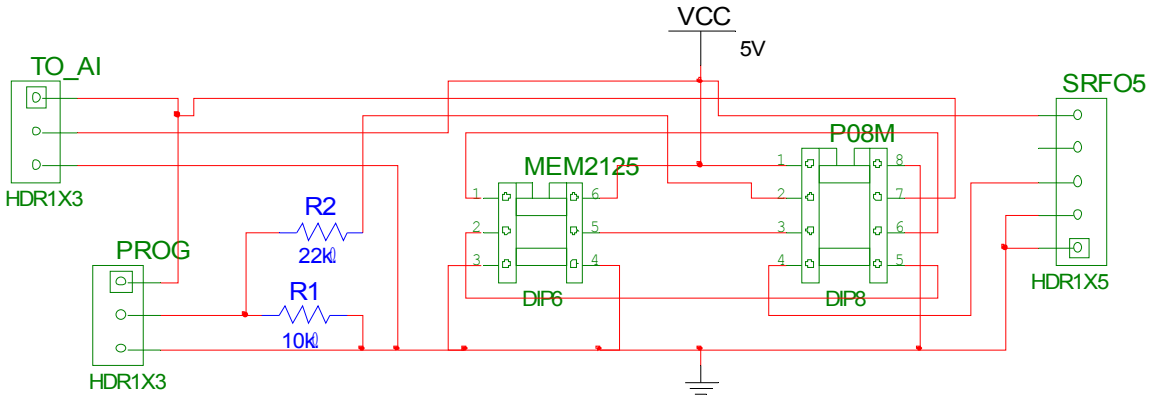
vane and compass are electronically equivalent – both sensors use a Hall-effect device pair, with the wind sensor sporting a free-wheeling weather vane carrying a neodymium magnet; the compass sensor is placed at a sufficient distance from other electronics (and the wind sensor) to avoid magnetic interference.

Compass / wind vane sensor



The header marked “TO AI” connects via a servo wire to the NAVCOM AI motherboard; the header named PROG can be used with a serial cable to reprogram the Picaxe-08M without having to remove it from its protective casing.

Clinometer + obstacle detector



The clinometer is a variation of the above, with added capability to read a sonar range finder; this sensor is located on ELAINE's prow, and informs the NAVCOM AI of the hull's pitch, roll, and (indirectly) wind velocity measured by the roll axis inclination. The range finder is an optional component whose presence allows the Picaxe to send an extra value to the NAVCOM AI – obstacle detection will trigger evasive maneuvering. This sensor pairing was decided upon for reasons of weight and space; since the PICAXE could handle the extra sensor, there was no point in adding a second microcontroller.

c. Independent systems

The auxiliary motor within ELAINE is triggered by the sail servo arm retracting completely and touching a pushbutton; while this arrangement may appear primitive, it allows for the motor (the single most power-consuming component in the entire vehicle) to be electrically insulated from everything else. An added benefit is that during manual operation ELAINE can be controlled with a simple, inexpensive two-channel transmitter.

The bilge pump uses a simple water level detector built from a LS1084 voltage regulator (the same regulator used on the motherboard); since the device is CMOS based, merely allowing a current path between two points on the pump that are close to its bottom is enough to start the motor. The choice of component was simply due to availability, given that some had to be bought for the motherboard anyway.

Appendix F – Datasheets

The datasheets for every active component used in the ELAINE project are available at

http://67.15.245.144/portfolio/navcom_ai/datasheets.zip

and are copyrighted by the respective manufacturers.

Worth noting is that the Picaxe-08M is a PIC12F683-I/P microcontroller loaded with proprietary bootstrap code and a pBASIC interpreter; since speed was not a concern for NMEA sensors, the modified microcontrollers were preferred to the originals for ease of development.

The most current datasheet set as of May 1st, 2007 is included in the CD that accompanies this report.

Appendix G – Source Code

The complete source code for the NAVCOM AI, containing AI applications for the Plane Drop project and the ELAINE project, is available at

http://67.15.245.144/portfolio/navcom_ai/source.zip

and is to be distributed under the General Public License. For the purpose of modifications and additions pertaining to educational projects, Saint Mary's University is assumed to be the copyright holder. Feedback, bug reports and additions are appreciated and strongly encouraged.

Older versions will be made available at
http://67.15.245.144/portfolio/navcom_ai/source_archive/
as time permits.

The author discourages generating a hard copy of the source code unless necessary for development purposes, on account of the Spin language using significant indentation which is hard to reproduce on paper.

Sample source code for various types of sensors is provided in the source code archive; all sensors described there use the Picaxe-08M microcontroller and have been tested for full compatibility with the NAVCOM AI.

The most current source code as of May 1st, 2007 is included in the CD that accompanies this report.