PRATHAM

IIT BOMBAY STUDENT SATELLITE

Conceptual Design Report

On Board Computing

By

Vishnu Sresht

Ashay Awate

Omkar Wagh

Pratik Chaudhari

Antariksh Bothale

Chiraag Juvekar

Prashant Sachdeva

Department of Aerospace Engineering,

Indian Institute of Technology, Bombay

July, 2008

Preface

The IIT Bombay Student Satellite is an initiative of the students of the Indian Institute of Technology, Bombay to make a satellite all by themselves. With this goal in mind, over forty students from various departments have been working on this project over eight months till date.

The Onboard Computer has been designed to handle the various tasks imposed on it by all the other subsystems in the due course of the operation of the satellite. Its main purpose is the acquisition (from a number of onboard sensors), processing and storage of data. It is responsible for the control of the satellite via the control of the actuators (magnetorquers) in accordance with the control laws developed by the controls team. It also encodes stored data suitably for transmission by downlink to earth.

The Onboard Computer will physically consist of a single microcontroller with associated internal and external memories. It is also linked (via several I^2C and UART busses to the components (sensors, actuators and transmitters) of other sub-systems in the satellite. The microcontroller will be programmed to implement individual predetermined tasks that are called in a non pre-emptive fashion by a simple scheduler.

In this report, the On Board Computing team has documented the work done towards designing and fabricating the onboard computer.

The IIT Bombay Student Satellite project is a dream of the students of the institute. We hope to see the satellite in orbit by the end of this decade.

Table of Contents

SUBSYTEM REQUIREMENTS

The Onboard computer has been designed to handle all the memory and logic intensive requirements of the other sub-systems of the satellite. This mainly consists of data collection, processing, storage and decision making processes.

Control Requirements:

- Execute the control laws (differ with the stage of operation)
- Interface with the sensors and actuators

Communication Requirements:

Initiating Different modes of data transmission depending on where we are.

- Stage II (not over India) : second (transmission) monopole will be switched off
- Stage IIIa (over India, but not over the ground station) : transmit GPS data
- Stage IIIb (over India and over ground station) : downlink HM data

Health Monitoring

- Monitor the operating conditions of various components
- Log this data into memory at various intervals of time
- This will be done at all stages of operation.

Housekeeping / health data will be sourced from multiple sensors. We have designed for 26 voltage / current sensors routed through 2 (16-channel) ADCs on 2 different I^2C busses. These sensors will be sampled at a frequency of 1 Hz. Orbital data collected from the GPS, magnetometers and sun-sensors will need to be collected for running the control law. These readings will be averaged over a longer period (than required by the control law) and stored for down-linking. The Power microcontroller will be interfaced with the Onboard Computer using an SPI bus. Incase of component failure, data will be relayed to the onboard computer through this channel.

All incoming data will be checked for consistency with the standard operating ranges of the respective component. Incase of exception, the necessary housekeeping task will be performed and the power microcontroller will be informed. All exceptions will be logged for later down-linking.

Readings that lie within normal operating ranges will be time averaged and stored for later down-linking. The collected data is to be transmitted up to 3 times in the time span of one orbit; hence the implementation of redundancy measures like transmission error correction was deemed unnecessary.

HARDWARE

Hardware Description

The hardware for the onboard computer is to consist of 3 main components:

- The AT91M400800 microcontroller
- A one time programmable EPROM memory – AT47BV040
- A bank of NVRAM – BQ4015LYMA-70N

Criteria for selection

The main criteria for selecting the components listed above were their space heritage and resistance to radiation. The onboard computer is possibly the most critical component on the satellite and our major concern was to ensure its functionality in the harsh environs of space. However, keeping with one of the major themes of this project, the use of easily available commercial off the shelf components, space specific processors (like the SPARC line) were not considered. To ensure space radiation hardness, the following constraints were imposed:

- Components with space heritage were given higher priority
- Flash memories were not considered as they are more susceptible to radiation induced latch-ups

Getting familiar with the EB40A development board

The Microcontroller used was the Atmel AT91R4008 (with an ARM7TDMI core). This is pin compatible with the AT91M40800 Microcontroller which we will employ on the final satellite. The only differences between the 2 microcontrollers are in the availability of on-board flash and an External Bus Interface which we will evaluate on the prototype board.

Development Tools and Environment for the Evaluation Board:

The EB40 came with RealView Development Suite (with only a 45 day evaluation period). This suite came with the Angel (serial cable) debugger which was a great aid to development. However, we managed to exhaust the evaluation period by January 2008 and were thus forced to hunt for other alternative development tools. The 2 major considerations were a development system on Linux/Cygwin with arm-gcc or using a licensed version of popular IDE's on Windows. We decided to go ahead with the KickStart version of IAR Studio. The license for this is easily obtainable and the only restriction is a 32 KB limit imposed on the code size.

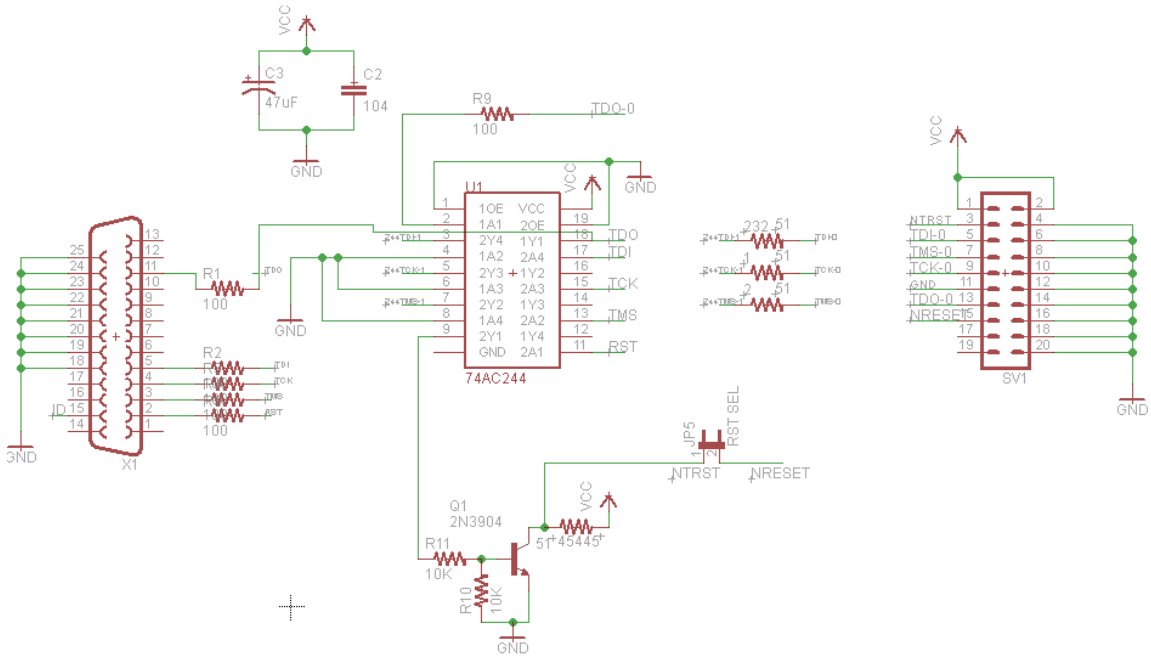
We also needed a programmer with which for the microcontroller. The only option available was the JTAG. We chose to make our own parallel port JTAG instead of buying commercial options. We found that the JTAG-Wiggler worked for the EB40A.

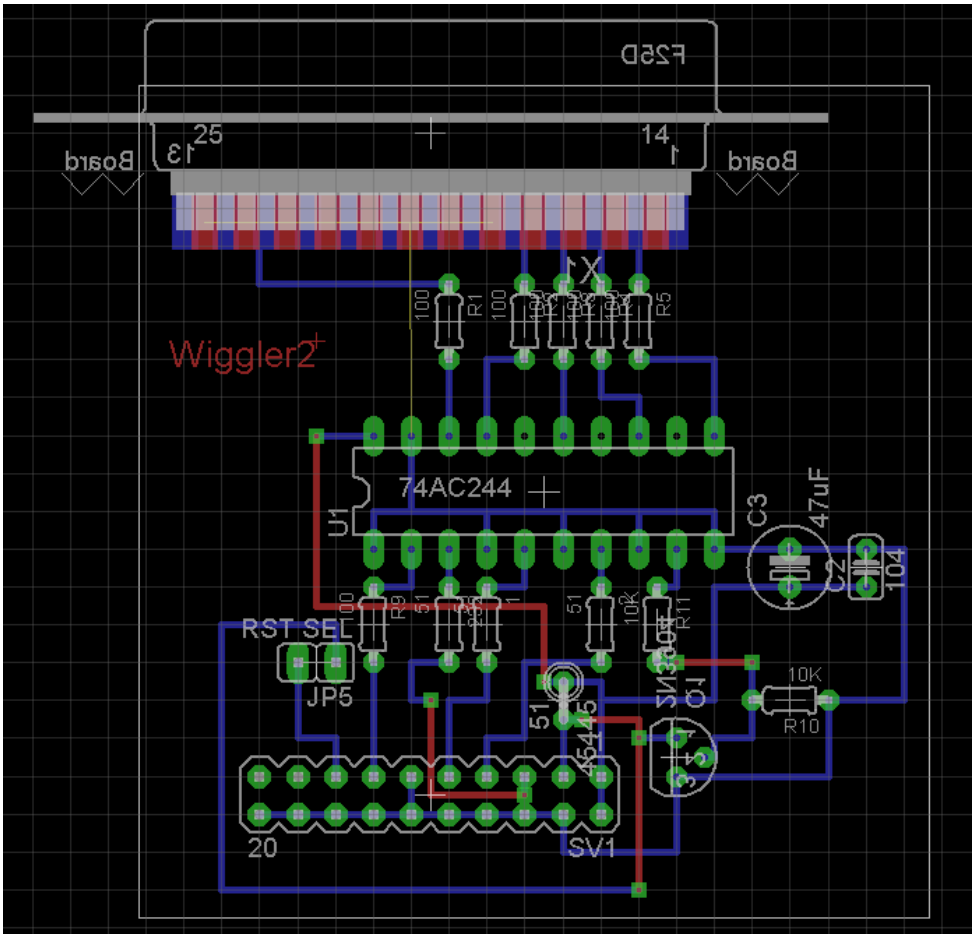
We made this on a prototyping board and hence did not bother to get a correct PCB. Wiggler connects to the PC parallel port with the DB25 adapter. We found changing the parallel port type in BIOS settings to EPP useful.

The software (H-JTAG that is freely available on the internet at http://hjtag.blogspot.com) was used to setup a debug server. The following configuration settings (obtained from one of the posts in the HJTAG forums) were used:

- nTRST Pin2 D0
- nSRST Pin6 D4
- nTRST output inverted

A good way to check that the debug-server is up and running is to see if you can detect and reset the microcontroller from the HJTAG user interface.





Wiggler

It turns out that on the schematic pin number 11 on the JTAG side is connected to the GND. On the EB40A however, this pin is connected to the TCK pin (clock for JTAG). So while using the EB40A board, simply disconnect the pin11 of the JTAG. The jumper in the schematic can safely be ignored for most of the microcontrollers. It is however advised to keep the jumper in case your microcontroller has different lines for nRST and nRESET. On most microcontrollers, these lines are shorted on the die itself.

IAR Workbench

This piece of information was obtained from the SparkFun electronics site. We configured an IAR project for use with the wiggler and HJTAG. We choose RDI as the Debugger. On the development board, we needed to make set the jumpers in the correct positions.

Testing of the various Peripherals:

Parallel Input I/O Controller: This microcontroller has 32 GPI/O pins. All of them have configurable hardware interrupts. The first program that we wrote on this was to switch on a single led. The Atmel Arm7 has a special implementation for all the peripheral control registers. Separate registers are used to

set and clear the same pin, making the programmer's task very easy. The next (obvious) step was to blink the led. We then moved on to creating various patterns using all the led's at our disposal. The input channels come with optional input filters. These will prove to be quite beneficial in the satellite environment.

Timers and Counters:

The ATX family has 3 channel 16-bit timers with 2 channel I/O pins per timer. We will be using these timers mainly to generate hardware ticks for the OS scheduler. We tested these timers and our ability to use them by sequencing a series of LEDs to blink at different frequencies and in a cyclic manner.

Timers are not very well simulated while using the JTAG debugger. The timer counter cannot be accurately monitored using the JTAG.

Advanced Interrupt Controller:

The arm core has only 2 interrupt channels – the nIRQ (for normal interrupts) and the high priority nFIQ. The whole peripheral interrupt controller of the microcontroller is built upon just these two lines. It supports 8-level customizable priority, maskable, vectored interrupts. The vector addresses of the interrupt handlers can be stored and hence called from specific AIC registers leading to lower latency timings. The microcontroller also has 3 external hardware interrupts in addition to a software interrupt. The spurious interrupt handling capability will be quite useful against stray noise setting off interrupts.

The complex interrupt handling in the ARM microcontroller results in lots of difficulties for the new programmer. Every project will have some linker instructions regarding the mapping of exception handlers and memory . These will be present in the makefile for arm-gcc projects and the assembly startup code in IAR workbench. Specifically the file called Cstartup.S does the job of remapping the vectors and memory. After a power reset, it executes these routines and points the program counter to the main() function of the code.

Writing an Interrupt Service Routine:

It is necessary to explicitly push and pop status and CPU registers onto the stack at the beginning and at the end of every execution of the interrupt routine. Another method that will reduce code size will be to have asm functions inside Cstartup.S which can be called by another asm handler currently mapped to the particular vector address. The job of this handler is to just push the registers, change the privileged or the supervisor mode of the CPU, call the the C Interrupt Service Routine, pop registers in this order. This is the most involved portion of AIC on ARM.

There are a few application notes on Atmel site that are really useful while implementing the above mentioned routines.

One important point to note would be that the interrupts are not very well simulated while using the JTAG debugger. Accessing certain CPU registers while debugging results in extraneous interrupts being thrown. The microcontroller datasheet provides pointers to avoid such situations. It is thus a good practice to burn the program onto and execute from the Flash.

Communication Interfaces:

Universal Asynchronous/Synchronous Receiver and Transmitter:

This microcontroller has 2 independent UART channels. These can be configured to accept vector addresses of transmit and receive buffers. Communication is initiated as soon as you write these buffer addresses and their respective sizes. Receive and transmit interrupts give overall control over the process though it leaves the CPU free for other tasks while communication. There is also a loopback interface in both the directions that we used for testing and debugging.

The UART was implemented in loop back mode as well as a proper bidirectional receive and transmit system. This consisted of sending data from the hyper-terminal (in Windows XP) or similar programs, checking for patterns like ESC, Enter etc. and then sending some data back to the computer. This was done both in a continuous mode and with interrupts.

The receive and transmit buffers can be given as vector addresses to the module and this simplifies the task a lot. We can then read the buffers at our leisure.

Two wire interface

This microcontroller does not have a hardware implementation of the two-wire interface. We had to implement it by bit-banging.

I^2C implementation via bit-banging:

It was not very clear whether we will be using I2C or not, however we decided to implement it in order to give the other teams the freedom to choose sensors employing this interface. This implementation works at the standard I2C frequency (100 KHz). This frequency is obtained by toggling the SCK pin by the software. The delay loops have been calibrated to the MCK frequency of 66MHz and hence it will be necessary to calibrate them for some other frequency. The current version of the code also does not take care of interrupts. It is assumed that the microcontroller talks to the sensors on the bus after stopping all the other tasks. This goes well with our implementation of the round-robin scheduler. We are currently not looking at interrupts and hence the communication cannot be affected by them.

The I^2C protocol employs two lines called the SDA and the SCK lines. The SDA line is the data line and it will have to be bidirectional so that the slave can write data as well. The SCK line is the clock for the whole system. It is the master who has to provide the clock. Both these lines have the default state of 1 and hence have to pulled up by the hardware. Data on SDA can change only when the SCK is low. The system reads the value of the SDA before the falling edge of the SCK. In the current implementation, we read the SDA just before we write a zero to the SCK pin.

The protocol consists of three parts, the START bit, DATA bytes and the STOP bit. A high to low transition on the SDA will result in the START bit being transmitted. A STOP bit is a low to high transition on the SDA. Transmission starts as soon as the slave receives the START bit. There are acknowledgement bits in this protocol to ensure that the connection with the slave is not lost. All clock cycles after the START bit

are taken as data bytes in sets of 8. After each data byte, there will have to be an ACK bit by the slave. The master just has to read this bit.

There will be some more things when we try to interface different devices. Some of them include the ADDRESS byte, COMMAND byte(s), and DATA byte(s). Every device on the bus has a particular address. The master has to send this address appended with the R/W bit as the LSB right after sending the START signal. The device with the particular address responds by ACK and receives the next data till the STOP bit.

Here is a timing diagram depicting the states of the SDA and SCL lines during communication.

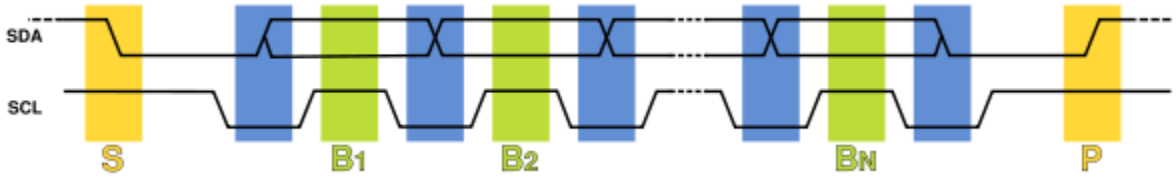


Diagram from Wikipedia

For even detailed explanation of the protocol refer to the Two-wire interface section of the datasheet for Atmel ATMEGA16 microcontroller.

The External Bus interface is yet to be tested. However, the currently used evaluation board boots off an external flash memory. Hence, we have a prototype of the boot-loader required.

Prototype board: Component selection

Microcontroller:

The microcontroller used will be the Atmel AT91M40800. This is the same microcontroller that will used for the final fight model.

Memory:

NVRAM: Texas Instruments BQ4015LY (4Mbit)

This is a CMOS SRAM that can retain data for upto 10 years after power-down. It will be used on the final flight model to store housekeeping data. It will also store data regarding the status of the microcontroller at every stage of operation.

Program memory: Atmel AT49BV040B (4Mbit) flash

This will be used to store the code. The program counter will be initialized to this memory by the boot loader. This is a flash memory and will not be used on the main flight board. It is being used during the development phase for rapid and cheap prototyping.

Program memory: Atmel OTP EPROM AT27BV040 (4Mbit)

This is a one time programmable memory and will be used for code storage on the flight model. All the specifications and interfaces are similar to the flash memory described above. EPROM is used as it is more resistant against bit flips and latching due to radiation.

Debugging:

- Serial interface: MAX232 for level shifting.
- A number of LEDs connected to spare GPI/O pins for easy debugging during runtime. However, as the microcontroller GPI/O pins cannot provide enough current to drive the LEDs and hence we have to use a buffer (74245) between them.

Power:

On the prototype boards, we will use Texas Instrument's PTH78000W module (9V to 3.3V). This will not be there on the final board as we will be powered by the Power sub-system.

Circuit Design

The EB40A board user guide comes with the schematic of the circuit. There are also a number of "circuit design notes" available on the Atmel site for ARM micro-processors. They describe some very good practices for decoupling on the board and also help in choosing the values of certain capacitors and resistors (crystal and PLL). We also referred to the Texas Instruments application note SZZA009: PCB Design Guidelines for Reduced EMI

This highly instructive document contains exhaustive information on reducing noise due to power rails, crystal noise. The satellite environment is not a very good one with extensive routing of the high frequency signals and hence it would be of great importance if we can protect ourselves from this noise. There are techniques for implementing ground planes that this application note brings to light. Our

circuit is quite complex and so far only some basic techniques have been implemented on the prototype board.

An unlicensed version of Cadsoft Eagle has been used for circuit design.

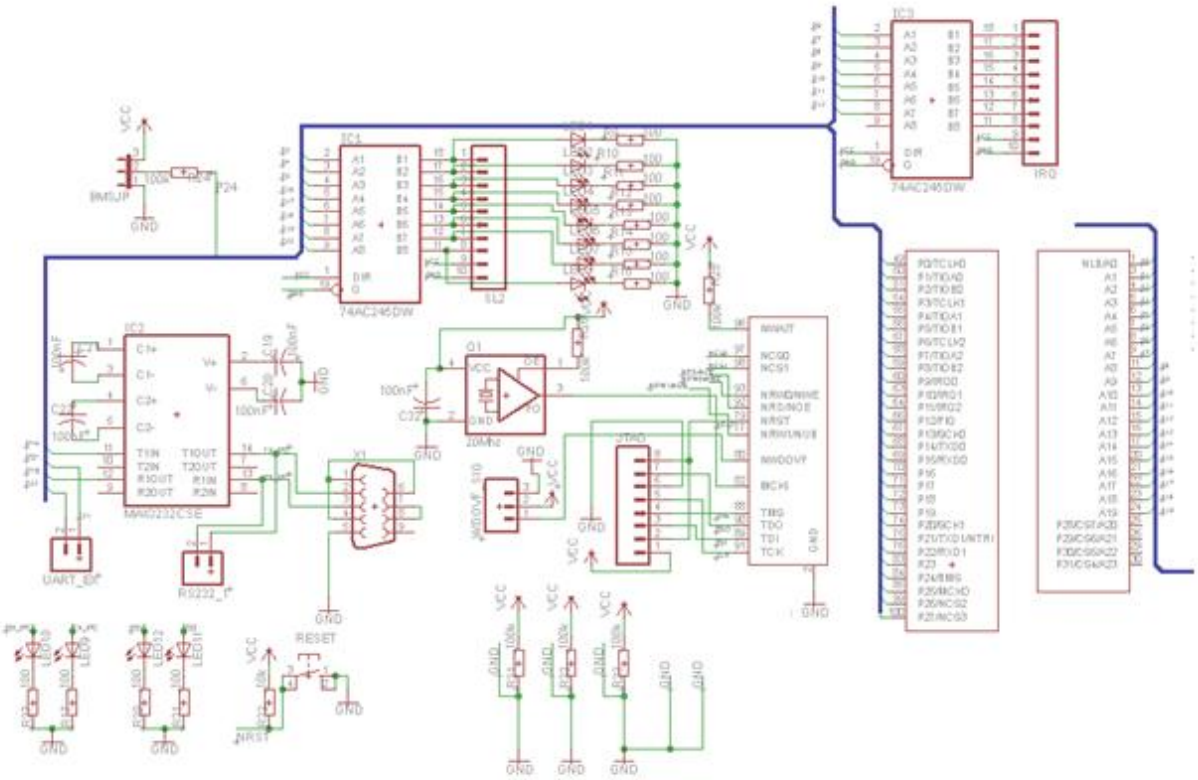
Bootup:

There is a BMS (Boot mode select) pin which enables us to boot from the external memory on NCS0.

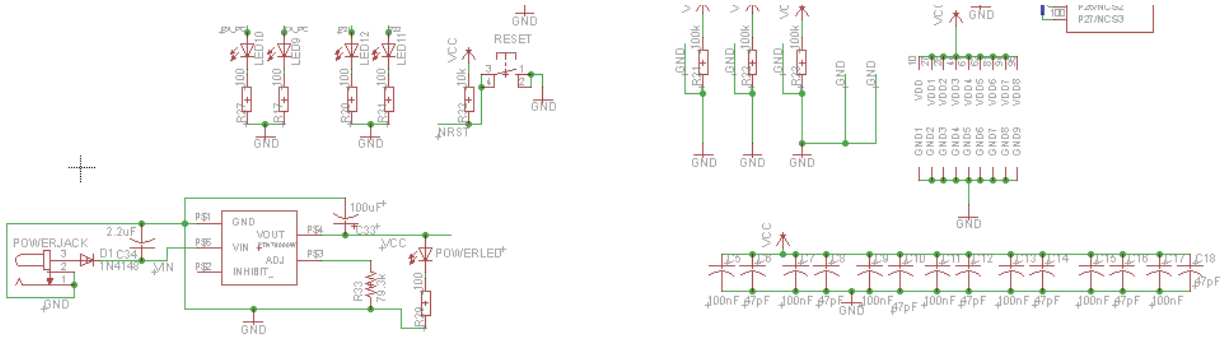
External memory:

The non-availability of flash on this microcontroller ensures that we need to have an external non-volatile memory for code storage on NCS0. However, the layout was not very good due to the awkward pin configuration on the memory ICs. Hence, the board has a FRC connector with the address and data lines taken out to be joined with a separate memory board. This also enables us to try different configurations of memory from different manufacturers on the same board without desoldering.

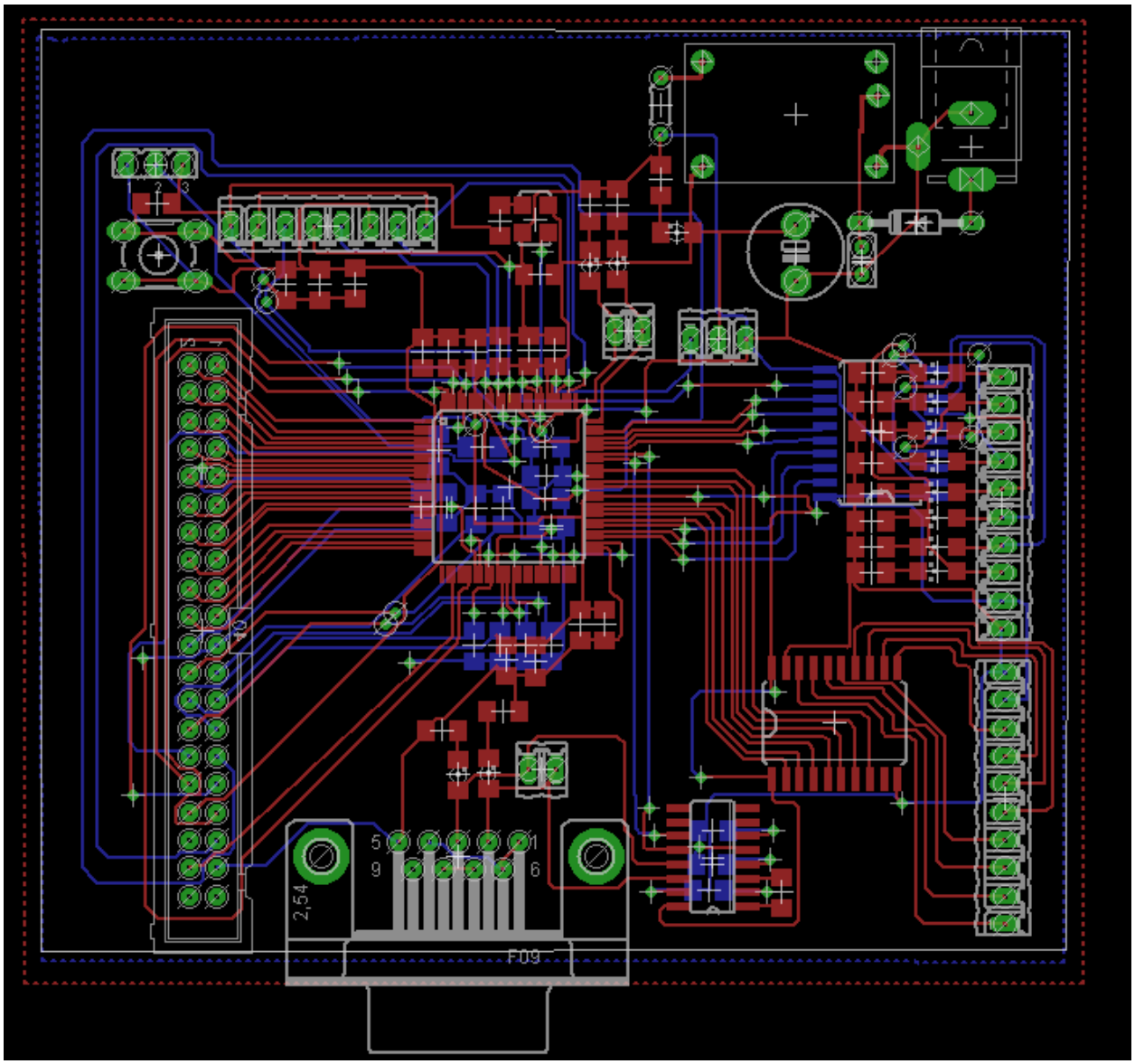
This is the schematic of the microcontroller part of the circuit with its debugging interface and the corresponding berge connectors. Berge connectors have been used extensively in this board to enable fast and easy interfacing with other components.



The power part and the very important decoupling capacitors,



Here is the board layout.



USB JTAG

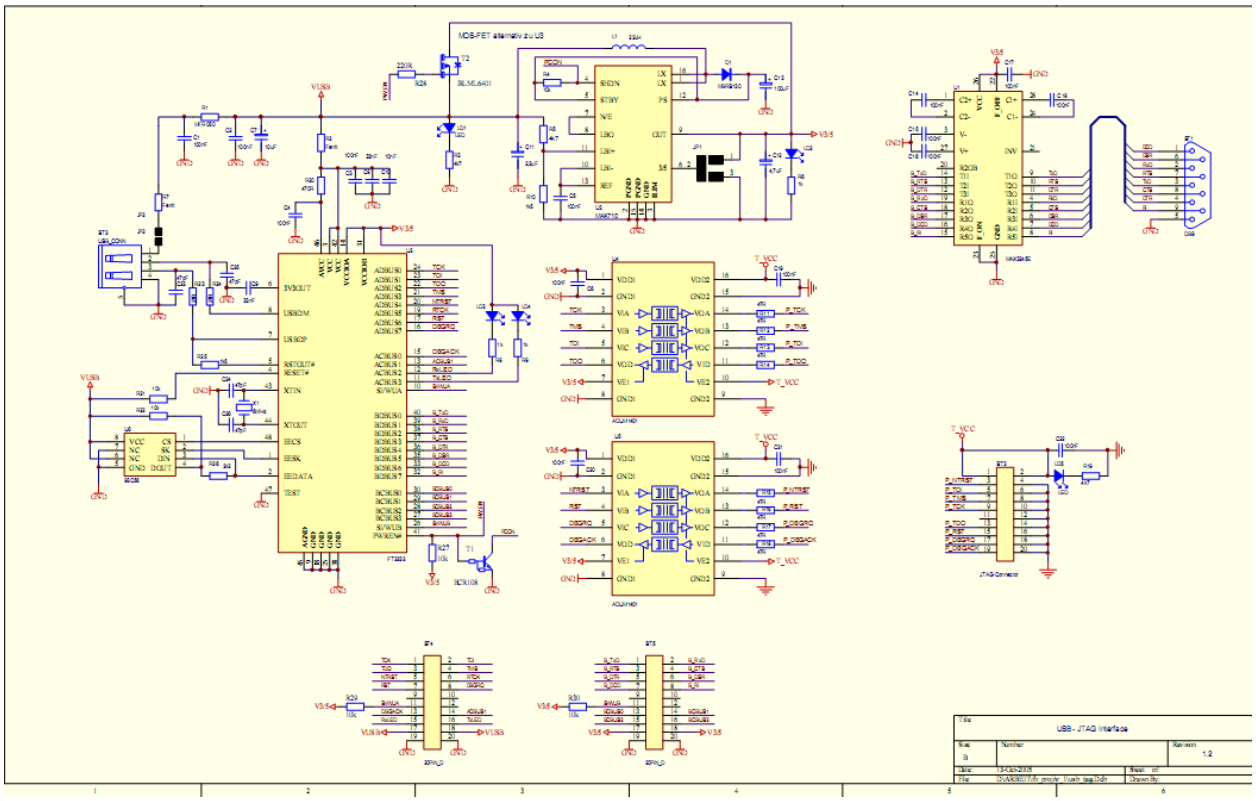
We wanted to have a USB JTAG debugger so that we could get around the dearth of computers with parallel ports. This link has a lot of USB JTAG circuits,

<http://www.hs-augsburg.de/~hhoegl/proj/usbjtag/usbjtag.html>

Currently another circuit titled Prototype 2 is being designed WITHOUT the aforesaid modifications for EB40A because the prototype board is slightly different from the EB40A in the JTAG interface.

The board is being made and will mostly be a PDIP circuit except for the FTDI FT2232 part.

Following is the schematic of the circuit,



SOFTWARE

Software Design

The lifetime of the satellite will be divided into several stages of operation. In each stage, the Onboard Computer will be called upon to perform different sets of tasks. Each stage characterizes the state of the state of the satellite. Thus, for a preliminary software design, a state-space was assembled listing out the various states of the satellite and the inputs, outputs and processing tasks associated with each state. This design also includes the paths that the onboard computer will take from one state to another.

Outline of Stages of Operation:

- Stage 1 :- First OBC start-up
    - Phase 1 :- Start up of OBC
        - Boot-loader(remaps, exceptions etc)
        - Sub-system check (Memory, GPS, Magnetometer)
        - Monopole deployment
    - Phase 2 :- De-tumbling State(75 s after boot to allow GPS cold boot.)
        - De-tumbling control law(0.25s deadline)
        - Health monitoring(1s)
        - Data logging(0.5 per min)
        - Watch-dog kick(deadline depending on settings)
- Stage 2 :- Normal with no downlink
    - Initialisations :-
        - Turn off second monopole
    - Execute the Normal control law.(1s deadline)
    - Health Monitoring (1s)
    - Data Logging(0.5 per min)
    - Communication check routine(once per second)
    - Watch-dog kick.(deadline depending on settings)

- Stage 3 :- Normal and down linking
    - Initialisations :-
        - Turn on second monopole
    - Execute the normal control law(1s)
    - Health Monitoring(1s)
    - Data Logging(0.5 per min)
    - Communication check routine(1s)
    - Downlink of data.
        - State1:- Not over GS, transmit standard bit patterns for identification.
        - State2:- Over GS, downlink data.
    - Watch-dog kick.

Emergency modes

- Power Low
    - Store one data packet in NVRAM indicating state of microcontroller. (this is set at the entry point of every state also)
    - Dump internal RAM (8KB) into NVRAM.
    - Dump essential CPU registers into NVRAM.
    - Send signal to power microcontroller to turn off main computer.
- Watch-dog reset
    - Read state data from NVRAM. Enter normal/de-tumbling operation mode.
- Component Failure mode:

    This stage consists of exception handling routines. This stage is entered in case single or multiple sensor or actuator failure(s) are detected. This stage will implement several routines that attempt to resurrect the failed component or initiate tasks that will try to compensate for the loss of data or control. In all cases, data logging will be done documenting the failure and the stage at which it occurred.

    - Magnetometer/Magnetorquer failure :-
        - Communications check routine.

- Down linking of data at appropriate time.
- Health monitoring/Data logging.
- Attempt to restart the magnetometer in the hope that it may start working again.

- GPS failure :-
  - Health monitoring/data logging.
  - Second monopole switched on for x% time.
  - Downlink of data from second monopole.
  - GPS restart attempt.
- Communication circuit failure :-
  - Attempt to restart ciruits.
  - Normal control law operation.
  - Health-monitoring/Data logging
- External NVRAM burnout :-
  - Attempt restarting.
  - Run normal control law.

Some special features:

Whenever a new stage is entered, data will be entered at a special location in the NVRAM. Thus, incase of a reset, the boot-loader can read that memory location and determine the stage of operation at the time of failure. This will prevent unnecessary repetition of certain tasks. It will also enable logging of the error and its circumstance.

A highly reliable, internal Watchdog timer will be used to ensure that a task does not 'hang'. The timer will force a restart of the microcontroller after a preset time interval, unless it is reset ('kicked') in time.

Task Scheduling

Initially (when the payload was a thermopile experiment) there was going to be a lot of data handling. There would have to be error checks on this data and also calculations on how to send it. This seemed

like a processor intensive task. However now the experiment has been simplified (TEC measurements), so our team felt that an Operating System would not be needed. Moreover, there is no need for either preemption or context-switching.

The various tasks that we now have can be handled by a scheduler created from scratch without the overhead of running an OS

All the tasks on our satellite will be deterministic (i.e. they will have a predefined maximum running time), which makes creating a scheduler a bit easier. Tasks will have different frequencies depending upon their importance. There will be a specific time allocated for a task to finish (since it is deterministic) after which the next task will be run. Since we will be dealing with limited amount of well defined tasks, a complicated scheduler is not necessary. One which is more robust is preferred. A simple scheduler will be used to handle the various tasks running on the satellite. The scheduler makes it possible to execute various tasks at different frequencies. The scheduler has a list of tasks along with their execution times, and depending upon this list it decides when to execute and how microcontrollerh time to allocate to each task.

Basic scheduler design

We plan to implement a cyclic scheduler. All the tasks will be stored in a circular queue. Each task will be tagged with its maximum execution time as well as a flag. Tasks that have a higher priority will have more instances in that queue. The scheduling function will iterate over the queue. It will execute every task that has its flag set. It will also allocate a time slice equal to the maximum execution time of that task.

Work done on the scheduler

Initially we tried to create a scheduler from scratch and program it on to an ATMEGA16. This effort has met with limited success. So an Operating System (SeOS) was programmed onto an ATMEGA16. Then it was stripped down by removing all the unnecessary code leaving only the scheduler and a few basic functions. We ran this bare OS with multiple tasks having different priorities. Then a basic scheduler was first implemented in C++ and then ported to an ATMEGA 16 microprocessor.

Unresolved design issues:

- Full details of structure should be used to store the task list
- What sort of features should be implemented to for redundancy?
- Efficient methods to introduce new tasks and move around the ones already present (if needed).
- Methods to kill a hung task independent of the watchdog timer.

Communication Protocol

Introduction:

A communication protocol is essentially a reliable format for sending data across a communication channel. Here, we are referring to the communication across the radio channel from the satellite to the ground station.

Criterion for selection:

1. Reliability: It is important the communication protocol we use should be able to deliver data with a certain level of accuracy.
2. Standardization: It is necessary to use a relatively standardized protocol for this purpose, as we expect other colleges to receive signals from our satellite.
3. Documentation: A well documented protocol is always better as we can sort out any small problems which may otherwise go unnoticed.

Available Protocols:

Some of the protocols that other satellites have used are:

- SSP - Simple Serial Protocol:
    - It is a simple protocol intended for master-slave communication on single-master serial buses, especially within small spacecraft.
    - A useful implementation is possible on a wide variety of hardware, including small microcontrollers with very limited resources.
    - However, when it comes to satellite communication, its strict request-response structure and lack of a packet serial number make high throughput difficult to achieve over such links.
    - Conclusion: Even though it is easy to implement and very flexible, it is not suitable for satellite communication due to its intrinsic nature.

- AX.25
    - It is a packet radio protocol, developed for use by the Amateur radio community.
    - This is very microcontrollerh the standard protocol used by most nanosatellites and microsatellites. (Refer to table/document with page number)
    - AX.25 is very well documented.
    - It is fairly easy to implement.(Refer to implementation)

- Implements CRCs for error detection, but has no error correction.
- Supports low baud rates.
- Conclusion: It very microcontrollerh fits our requirements (in spite of the relatively low baud rate), except that it does not implement any error correction. The collected data is to be transmitted up to 3 times in the time span of one orbit, hence error correction was deemed unnecessary.

- SRLL – Simple Radio Link Layer:
  - Experimental protocol developed by TITech (Tokyo Institute of Technology).
  - Uses fixed length packets and frame detection by 32 bit PN code.
  - Implements FEC (Forward Error Correction): 1 byte for every 2 bytes.
  - It is fairly easy to implement and has been successfully implemented in CUTE -1 by TITech.
  - Conclusion: It fits most of the requirements for us, but the lack of documentation is a big deterrent.
- Custom Protocol:
  - Might make matters far simpler.
  - Error Correction could be implemented.
  - Tried by very few microsatellites/nanosatellites successfully.
  - Makes it tougher for other ground stations to receive and interpret data from our satellite.
  - Would require extensive experimentation before usage and documentation after implementation.

Conclusion:

Even though new protocols like SRLL or custom-made protocols seem attractive for many reasons, esp. their error correction capability, for our first satellite we would rather like to implement a standard protocol that is well documented. Hence AX.25 is seemingly the best option and we are going to use it for satellite to Ground station communication.

References:

1. http://atl.calpoly.edu/~bklofas/Presentations/DevelopersWorkshop2008/CommSurvey-Bryan_Klofas.pdf : Communication Protocol Survey
2. http://lss.mes.titech.ac.jp/ssp/cubesat/srll/srll.htm
3. www.tapr.org/pdf/AX25.2.2.pdf : AX.25 Link Access Protocol for Amateur Packet Radio. Version 2.2

Table 1: Summary of spacecraft transceivers.

Satellite	Object	Radio	Frequency	License	Power	TNC	Protocol	Baud Rate/Modulation	Amount Downloaded	Antenna
AAU1	27846	Wood & Douglas SX450	437.475 MHz	Amateur	500 mW	MX909	AX.25 on Mobitex	9600 Baud GMSK	1 kB[1]	dipole
CanX-1	27847	Melexis	437.880 MHz	Amateur	500 mW		Custom	1200 baud MSK	0[2]	crossed dipoles
Cute-1 (CO-55)	27844	Maki Denki (Beacon)	436.8375 MHz	Amateur	100 mW	PIC16LC73A	CW	50 WPM	N/A	monopole
		Alinco DJ-C4 (Data)	437.470 MHz	Amateur	350 mW	MX614	AX.25	1200 baud AFSK	?[3]	monopole
DTUsat-1	27842	RFMD RF2905	437.475 MHz	Amateur	400 mW		AX.25	2400 baud FSK	0[2]	canted turnstile
QuakeSat-1	27845	Tekk KS-960	436.675 MHz	Amateur	2 W	BayPac BP-96A	AX.25[4]	9600 baud FSK	423 MB	turnstile
XI-IV (CO-57)	27848	Nishi RF Lab (Beacon)	436.8475 MHz	Amateur	80 mW	PIC16C716	CW	50 WPM	N/A	dipole
		Nishi RF Lab (Data)	437.490 MHz	Amateur	1 W	PIC16C622	AX.25	1200 baud AFSK	?[3]	dipole
XI-V (CO-58)	28895	Nishi RF Lab (Beacon)	437.465 MHz	Amateur	80 mW	PIC16C716	CW	50 WPM	N/A	dipole
		Nishi RF Lab (Data)	437.345 MHz	Amateur	1 W	PIC16C622	AX.25	1200 baud AFSK	?[3]	dipole
NCube-2	28897[5]		437.505 MHz	Amateur			AX.25	1200 baud AFSK	0[2]	monopole
UWE-1	28892		437.505 MHz	Amateur	1 W	Integrated[6]	AX.25	1200/9600 baud AFSK	?[3]	dipole
Cute-1.7+APD	28941	Telemetry Beacon	437.385 MHz	Amateur	100 mW	H8S/2328[7]	CW	50 WPM	N/A	dipole
(CO-56)		Alinco DJ-C5	437.505 MHz	Amateur	300 mW	CMX589A	AX.25/SRLL	1200 AFSK/9600 GMSK	0	dipole
GeneSat-1	29655	Stensat (Beacon)	437.067 MHz	Amateur	500 mW		AX.25	1200 baud AFSK	N/A	monopole
		Microhard MHX-2400	2.4 GHz	ISM	1 W	Integrated[6]	Proprietary		500 kB	patch
CSTB1	31122	Commercial[8]	400.0375 MHz	Experimental	<1 W	Integrated[6]	Proprietary	1200 baud AFSK	6.77 MB[9]	dipole
AeroCube-2	31133	Commercial[8]	902-928 MHz	ISM	2 W	Integrated[6]	Proprietary	38.4 kbaud	500 kB[1]	patch
CP4	31132	TI CC1000	437.325 MHz	Amateur	1 W	PIC18LF6720	AX.25	1200 baud FSK	320 kB[1]	dipole
Libertad-1	31128	Stensat	437.405 MHz	Amateur	400 mW		AX.25	1200 baud AFSK	0[10]	monopole
CAPE1	31130	TI CC1020	435.245 MHz	Amateur	1 W	PIC16LF452	AX.25	9600 baud FSK	0[11]	dipole
CP3	31129	TI CC1000	436.845 MHz	Experimental	1 W	PIC18LF6720	AX.25	1200 baud FSK	1.6 MB[9]	dipole
MAST[12]	31126	Microhard MHX-2400	2.4 GHz	ISM	1 W	Intgrated[6]	Proprietary	15 kbps	>2 MB[1]	monopole

[1] Spacecraft worked on orbit but is now dead.

[2] Satellite never heard from in space.

[3] Japanese data downloaded is unknown as of 8 April 2008.

[4] Used a modified Pacsat protocol on top of AX.25. Source code available upon request.

[5] This object separated from SSETI Express months later and is presumed to be NCube-2.

[6] The radio module accepts serial data and uses an internal TNC.

[7] This is the main satellite processor.

[8] The exact model number is unknown.

[9] As of 8 April 2008.

[10] No uplink commands received by spacecraft.

[11] The CAPE1 team knew the receiver was dead before integration but had no time to fix it.

[12] One identical radio per satellite section.

Survey of All Student Satellites

Source: http://atl.calpoly.edu/~bklofas/Presentations/DevelopersWorkshop2008/CommSurvey-Bryan_Klofas.pdf

BUDGETS

Memory Budget

Uses:-
1.Data Logging.
2. Temporary storage to be used for averaging.
3. Code Space.

Data Logging
The total amount of memory that will be required per day for data-logging :-

Sr. Description	Size	Frequency
1. Voltage+Current	8 bits x 26	1/min for all
2. Temperature	(4+6) x 8b	
3. Attitude	3 x 16 bits	
4. Position	25 x 8 bits	Once every 10 mins from GPS
5. Time	32 bit	Once every 10 mins from GPS

Total amount of data generated in one day 64 KB in 24 hours.

2. Temporary storage
Data to be collected once every minute to be averaged over longer (as yet undecided interval).

From the communication side:
We know that the data rate is 1.2 kbps so that gives us a downlink capability of 61 KB assuming a 7 min transmission suitable pass.

3. Program Memory

The restriction from the micro-controller is that we need to store the code on a non-volatile memory that can be accessed used the EBI peripheral. We plan to use a 4 Mbit eprom as program space. This

might also contain various pre-coded tables needed for lookup during the execution of the control law.

4 .Code Variables
This would naturally have to be on a RAM memory. This will probably be the internal SRAM (8 kB .. should be more than enough for code variables)

Our Memory plan:-
1 bank of NVRAM 512MB BQ4015LY used for the data logging operations .
1 EPROM for the code memory (storage of the program code)

Pin Budget

Total GPIO		32
Interfaces		
Magnetometer	2 UART (RX + TX)	2
GPS	1 UART (RX)	1
Debugging	2 Pins implementing custom protocol	2
Comm.	1 UART (TX)	1
ADCs	SPI/I2C yet to be found out	2x3 or 2x2
Magnetorquers		
3 timers + 6 direction		9
Power out Interrupt	IRQ	1
Boot mode select		1

Timing Estimate

SunSensor + Gyroscope

6 SS + 3 Gyro = 9 Sensors (Taken as 10 in further calculation)

Data to be taken via ADCs at 10 Hz 10*10 = **100 cycles**

4 Floating Point Multiplications per sensor 4*10 = 40 operations = 40*200 cyc/op = **8000 cycles**

Total cycles = 8100.

Time required = 8100/20E+6 = 405 µs

Magnetometer – Will work on UART. Interfacing will be done without processor

Baud rate = 9600 bps

Taking 5 bytes per reading and 10 readings per second = 5*8*10 = 400 bits

Time for Magnetometer reading = 400 bits @ 9600 bps = 42 ms

4 Float Multiplications at 10 Hz = 4*10*200 cycles Overhead = **8000 cycles**

Total processor time = 8000 @ 20 MHz = **400 µs**

GPS – Will work on UART. Interfacing will be done without processor

Baud rate = 4800 bps

Data transferred per sec = 64 bytes * 5 /sec = 320 Bps

Time needed = 320*8/4800 = 533 ms

Total data input sans processor = 533 ms || 42 ms (Parallel)

Total processing time for data time = 805 µs

Power Budget

Component	State	Voltage	Current	Power (max)
NVSRAM bq4015LYMA-70N	Active	3.3	50 mA	165mW
	Power Down	<2.90	1 µA	
AT91M40800	Reset	3.3 V	-	0.1 mW/mHz
	Active	3.3 V	-	4.63 mW/mHz
	Idle	3.3 V	-	2.06 mW/mHz
AT27BV040	Read	3.3 V	10 mA	33 mV
	Standby	3.3 V	20µA	-

CRITICAL ISSUES

As of now, there are several issues that can be treated as points of failure.

- Hardware Failure
    - Faults in the main Microcontroller and code memory (PROM)
    - Fault in communication bus interfacing with power Microcontroller
- Sensor Failure
    - GPS, magnetometer or sun-sensor communication or configuration failure.

These failures cannot be tolerated and as of now, robust error prevention techniques and redundancy measures have not been designed.