C&DH

End-of-Quarter

Fall 1999

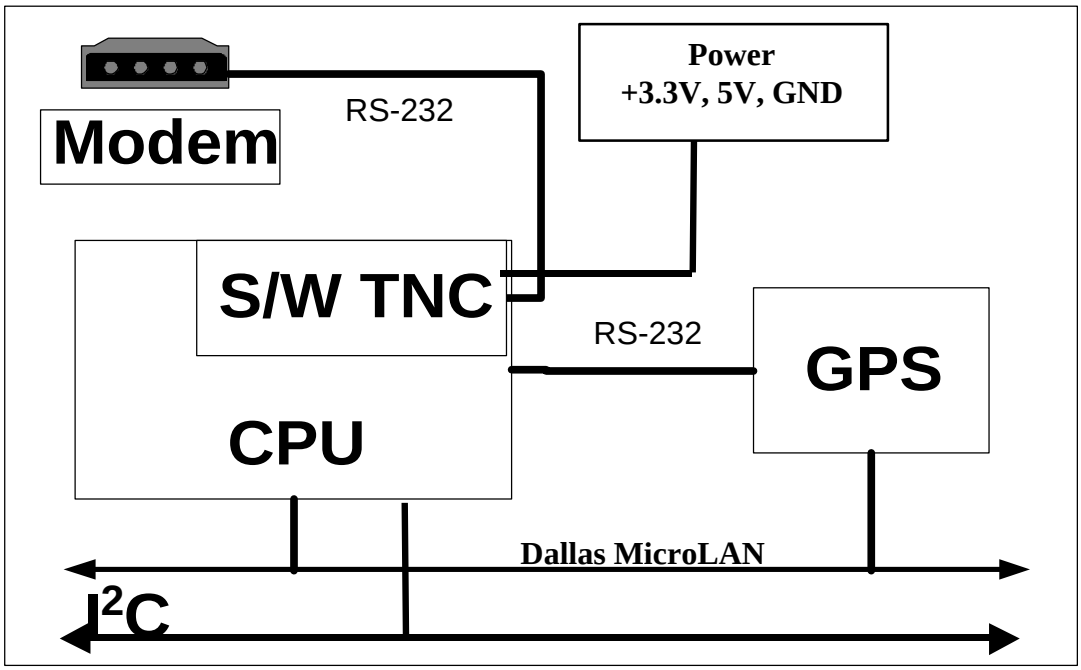
Level 1 : Statement of Objective

Command and data handling is the heart and brain of any spacecraft. This subsystem functions as the director of spacecraft information. The objectives of this system are:

- to decode information sent from the ground or other spacecraft
- to download data to the ground station
- to perform any commands requested of the s/c
- to execute control instructions given to it by the GPS computer for the spacecraft's attitude and position
- to pass information between the various subsystems
- to store data collected by the instruments
- to gather health information and telemetry data

Level 2 : Subsystem Flow Chart

Information is received from the ground station through the Communication Subsystem and passed to the CPU. The SpaceQuest CPU will run a software TNC that will decode the communication. The GPS can pass information to the CPU through a dedicatedRS-232 line. The information may be information to be sent to the modem, data from other satellites, or commands to the AD&C system. Communication to other subsystems will primarily be made through and I2C connection. Temperature data, interrupt capabilities, limited A/D data, and power switching will be made through the Dallas 1-wire MicroLAN.



Level 3 : Interface Documentation

CPU Hardware Documentation

Interface Documentation

The CPU must talk serially to all of the subsystems. Most subsystems will be connected to the I^2C bus, so the CPU must be able to send commands to and receive data from the I2C bus. The CPU should have a more direct line to the downlink and crosslink Communications, so it will have an RS-232 serial interface with the modem. The CPU will also be able to take certain telemetry readings and reset the power on any of the subsystems using the Dallas 1-wire bus, providing extra bandwidth as well as redundancy. The GPS subsystem will either be connected via the I^2C bus or another RS-232 serial interface.

Function Description

The central processing done by the command and data handling subsystem on Emerald will be handled by the BekTek Operating system running on the SpaceQuest FCV-53 flight computer.

The SpaceQuest flight computer has three NEC 72001 GULL Serial Communications Controllers (SCCs) that are configured to provide six transmit channels and six receive serial channels. These serial channels will be configured to provide all of the communications necessary for the I^2C bus, the Dallas 1-wire bus, the RS-232 interfaces, and the debug ports.

The BekTek Operating System has a Software TNC that runs in conjunction with some of the ICs built into the SpaceQuest Flight Computer. This allows for compatibility with the Communications Subsystem.

The BekTek Operating System will be obtained with a development environment to speed the process of creating the software necessary to do task scheduling, data storage, subsystem command, serial communications, and any other tasks needed for the success of the mission. We will also obtain an RRIC board that plugs into a PC and allows fast debugging and testing of code written for the BekTek Operating System running on the SpaceQuest board.

Much of the debugging and testing of the serial communication interfaces will be done using a PC running a Visual Basic program designed to debug serial communications.

For component datasheets see the appropriate section in this document.

Inter Subsystem Data Bus

Objective

Allow modular subsystems
"object oriented" hardware design

Flow Chart

<Block Diagram goes here>

Interface Documentation

All systems must fully comply with I2C specification
Higher level protocol specified below
Hardware interface <see Interface Document>

Requirements

Requirements

Connect all of the subsystems

Goals

Reduce Wiring

Modular

"Object Oriented Development"

Scalability

Standardized SQUIRT Bus

"Plug and play" subsystems

Alternatives Considered

We decided to narrow the search to synchronous serial protocols for simplicity, reduced wiring, and speed. In a **serial** protocol, bits of data is sent sequentially over a small number of wires. Contrast to parallel, where multiple bits are sent simultaneously, one bit per wire. Serial connections have inherently fewer wires, but tend to be slower than parallel. There are two main types of serial connection: synchronous and asynchronous, which are distinguished by whether or not timing information is sent along with the data. In a **synchronous** protocol, a clock signal is sent on an additional wire to coordinate timing. In an asynchronous protocol, each end keeps its on clock running at an agreed upon rate. In general a synchronous protocol will have higher transfer rates and be simpler to implement. A synchronous device only has to wait for a clock pulse before looking at data, instead of having to constantly be aware of using a slightly different internal time-base, which may require additional samples, or at least a slower signal.

Existing Protocols:

Rather than re-invent the wheel, we wanted to use an existing communication protocol. We limited this search to simple protocols common in embedded systems. One key feature all of the alternatives had was at least some direct micro-controller support. We considered:
Controller Area Network (CAN): message based, complex, not well supported, used extensively in automotive.

Inter Integrated Circuit (I^2C): "2-wire," multi-master, requires intelligent receivers, used in A/V equipement and "smart" batteries, part of JPL X2000 bus.

Serial Peripheral Interface (SPI): widely used, separate address lines, simple

We omitted, any more sophisticated protocols such as USB, in an effort to limit complexity. We wanted to avoid needing external hardware (not built-in the microcontroller), at least for the distributed nodes (subsystems).

Design History

Early Design SPI-MM/I^{2}C

Originally we liked I2C for its scalability and simplicity, but were afraid that it was too complicated and would require too much processing at each Node. We liked SPI since it was simple enough to be implemented with little more than a shift register and magnitude comparitor yet also was built into the 68332. The biggest disadvantage was that it used external chip select lines and was therefor limited to 16 devices. The CAN bus, was omitted to limited built-in hardware support. So, our solution was to combine them into SPI-MM/I2C:

Shared clock and data lines for I2C and SPI

Used **SPI** for fast, simple Full-Duplex with little specialized hardware (up to 1Mbit/sec)

"**Multi-Mastering**" (MM) SPI, (use of wired-AND for signal lines)

I^{2}C for "unlimited" number of smart subsystems at up to 400kbit/sec (fast mode)

4 address lines (16 addresses)

One address reserved for I^{2}C

One address for broadcast

With this design everything fit in a DB9. A later sub-iteration added an additional line (bringing the total to 9) for selecting I2C or SPI

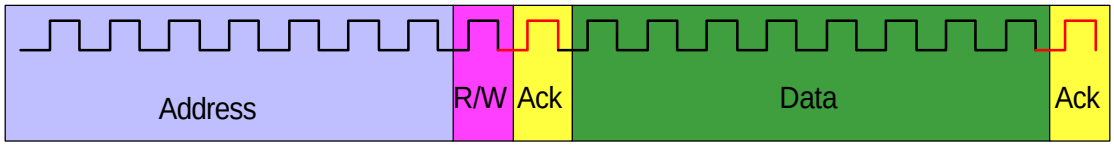
Current Design (I2C with separate Dallas I-wire)

The biggest disadvantage of the SPI-MM/I2C was its complexity, but we were willing to live with it to support simple non-intelligent subsystems. When we realized the wide assortment of simple devices available for I2C, (digital I/O, A/D converters, EEPROM, microprocessors, etc.) we decided drop the added complexity of the SPI multi-master and just use I2C.
We were also attracted to the simplicity of the Dallas 1-wire system, especially the temperature sensors. Since the I2C system only uses only 4 wires (SDA, SCL, plus two shields) there was still room in a DB9 to support the Dallas bus, which only requires a power/data line (plus a common ground, which is supplied by the power system). In addition to temperature telemetry, the Dallas system will be used to provide subsystem on/off signals with the available serially addressable switch.

Functional Description

Low level: I2C

I2C is a synchronous serial protocol that uses only 2 wires, one for data (SDA) and one for clock (SCL). It also requires a common ground, as supplied through the power system. It operates at 100 kbps (kilo *bits* per second) in standard mode. Faster modes (400kbps and 3.4 Mbps) are also specified, but fewer devices support these modes. The protocol is specified to a fairly high level from reading and writing to multi-master support and arbitration.
Both lines are connected wired-AND to allow for arbitrary numbers of devices and to support "hot swapping," adding/removing devices without interrupting the bus. Communication is always initiated by a master, which also drives the clock. Each I2C message consists of an arbitrary number of 9 bit "words." These words are 8 bits of information (supplied by the current "transmitter") plus an acknowledge (from the "receiver). The first word of the message sets the address (7bits) and the communication direction (Read or Write). In read mode, after the first acknowledge, the slave begins transmitting and the master becomes the "receiver." A pictorial representation of a simple I2C message is shown below:



For more information and details, see the I2C specification or “the I2C bus and how to use it” both put out by Phillips and included as part of this documentation

High Level Messaging

The protocol is fairly simple. Currently all devices on the I2C bus except the CPU are slaves. Because they are slaves they can only respond when data is requested of them. To send a command to a slave a command message is sent. A command message also precedes a slave read since the slave must be told what data to make available in its read buffer. The command message is as follows.

<FROM><COMMAND><DATA 0>…<DATA F><CHECKSUM>

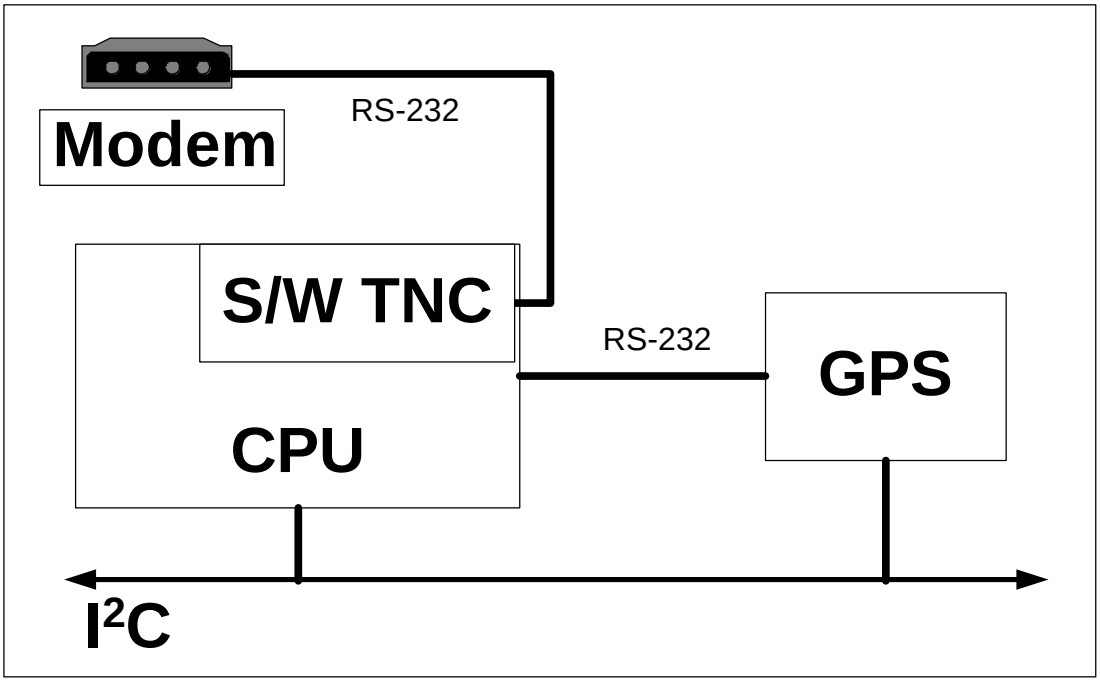
In order to know where the request is coming from, a FROM byte starts the message. Following is the command. Each device will have a command dictionary generated by the device’s subsystem. Following the command 16 bytes are available for parameters to the command. Finally a checksum that is a 8-bit rolling count of all bytes including the from is added at the end.

When reading from a device the CPU or Master must issue a read. In order to get specific data the slave must be told which data to make available by issuing a command as described above. The packet read from the slave begins with the number of bytes available. The MSb in the LENGTH byte signifies whether or not another packet after this one will be available. Currently 16 bytes may be read from the slave, but in the future packets of 126 bytes may be received.

<LENGTH>< DATA 0>…<DATA 8F><CHECKSUM>

In future version multi-mastering may be implemented.

Level 4 : Prioritized Requirements



Prioritized Requirements

Attitude Determination & Control

None

Command & Data Handling

Provide scheduling of tasks as necessary (4)

Communications and GPS

Must be compatible with CPU hardware (5)

Power

200 mW (5)

CPU should be powered at all times. (3)

Clean, low-noise signal, should not cause reset or glitches in any operational mode (5)

Dallas 1-wire commandable switches for subsystems (2)

Latch-Up protection. (5)

Structure and Mechanism

Mass: 0.75 (5)

Size: 140 mm x 165 mm x 2" (5)

Thermal

Maintain system within operational temperature range (5)

Dallas one-wire temperature sensors (2)

Systems

High level commandable subsystems (5)

I2C interface to subsystems (5)

Diagnostics & Testing

Satellite External Debug Port for I2C bus (DB9, pinouts as per Electrical Interface Document) (5)

Satellite External Debug Port for RS232 (stderr) messages. (DB9, pinouts matching standard serial port) (5)

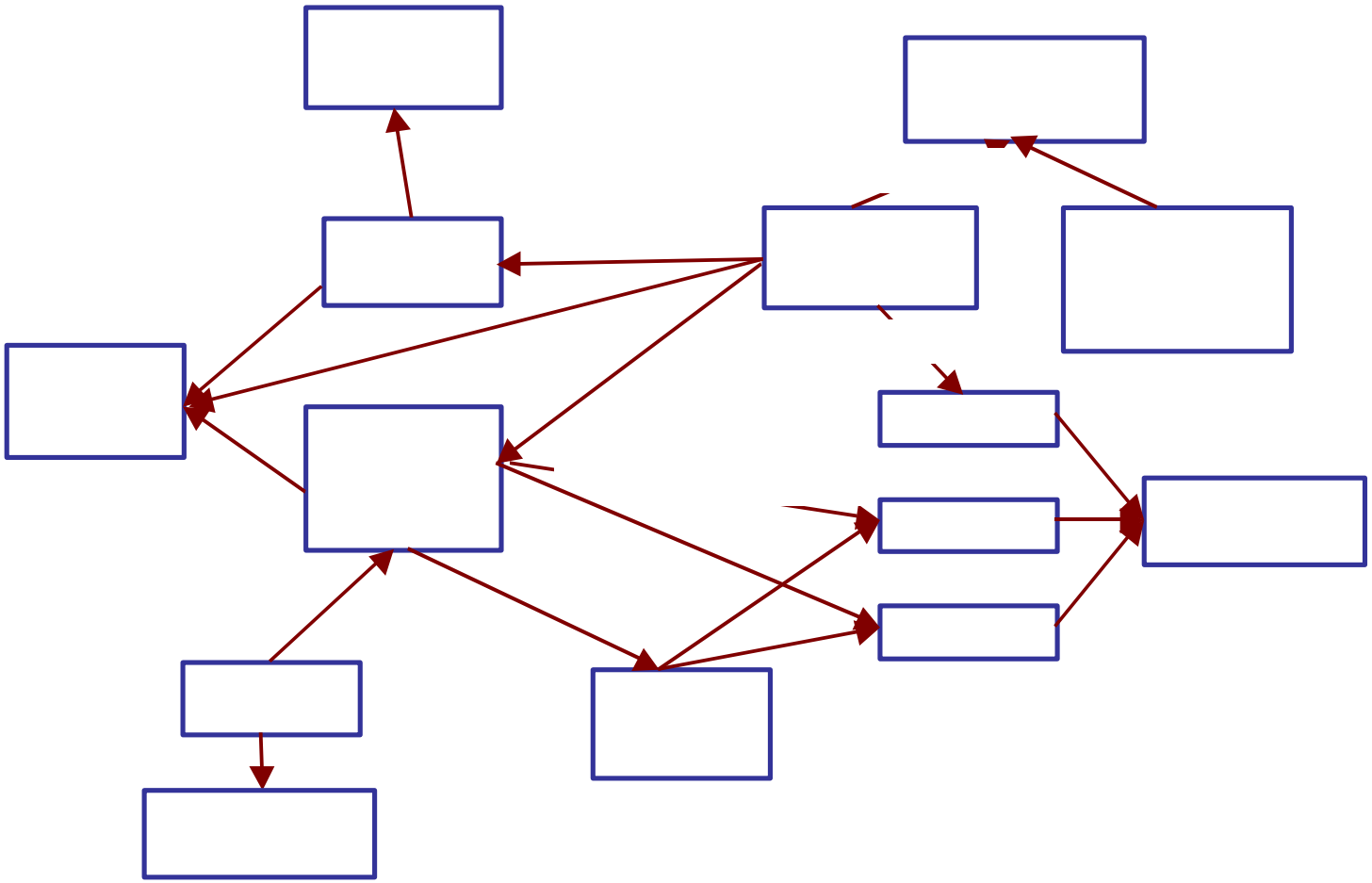
Mission Operations

Not defined as of yet. (?)

* Numbers in parentheses indicate priority. 1 is low, 5 is high.

Level 5 : Functional Description

A. CPU Software Architecture



Note that the above shows software module dependencies, rather than dataflow. For example, if the health monitoring task changes, then it is very likely that changes will have to be made to the beacon task as well.

The usefulness of this chart is made clear by the fact that it helps designers determine priorities. For instance, it is apparent that the Dallas and I2C tasks are depended on more than almost all others. Thus, they are one of our priorities.

B. Microchip PICMicro Microcontrollers - Subsystem Processors

•Why intelligent sub-systems

Ease of integration, Modular Development, Stand-alone testing, high level command interface

•Alternatives Considered

68xx/68xxx more power than most systems need, external peripherals required

COP little industry support

8051 odd architecture, external peripherals required

•PICMicro Controller

single CHIP computer (only need to add clock)

wide family range (8 to 68 pin, built-in A/D, I2C, etc.)

Radiation Research

Standard, Cheap Development

Low Power (<100 mW at max speed – scales with speed, <100μW sleep)

Experience through ME218

A brief introduction to the PICMicros

A very simple, yet quite powerful micro-controller.

- Only requires power and clock source (actually some have RC built-in, but inaccurate)
- Optimized for dealing with the real world
- simple "RISC" instruction set… Very limited number, but does alot
- Built in RAM, ROM, and special features
- Large and diverse family of chips and features

Our PICMicros: 16C74 and 16C73

- "Mid-range" family: largest, most diverse, 14bit instruction set
- 16C74 is simply a larger (40 pin) chip with more I/O than the '73
- Built-In functionality you might be interested in:

Feature	16C73	16C74	Comments
Async. Serial Comm.	built-in	built-in	TTL level RS232, requires level converter
I2C interface	built-in	built-in	
A/D converter	5	5 (8)	8 bit accuracy
Capture and Compare	2 built-in	2 built-in	also does PWM
max crystal speed	20 MHz	20 MHz	translates into a 5MHz instruction clock
ROM	4096	4096	14bit words
RAM	192	192	Bytes
remaining I/0	1(8)+2= 10	2(8)+5= 21	num 8 bit ports (*8) + random = total note most of these remaining pins used by external SRAM, if included.

PICMicro Software Objectives/Philosophy

- High level interface to hardware devices:
  - I2C including high level protocol
  - RS232 for debugging
  - A/D converter
  - Timer/Time Module
  - External SRAM support
  - Capture/Compare/PWM (TBD)
- Standardized Coding
  - Similar Look and Feel
  - Uniform function/variable/constant capitalization
- Modular Development
  - Each module can be tested stand-alone, yet reconfigured to also be part of a larger program
  - Global variables used only when absolutely necessary
  - Should respect and conserve system resources (time, interrupts, memory)
  - Well documented including use of hardware resources, global variables, and timing information.
  - Co-operative "multitasking" requires short duration functions.
- PICMicro embedded specific
  - Minimal large function calls (printf, etc.)
  - Simple math (integer arithmetic)
  - Look-up tables for more complex calculations
  - Minimal time in interrupts (non-vectored)

Functional Overview

See specific module documentation in appendix.

Overall program strategy:

Use a "super-loop" which executes in a fixed, constant time
Call functions from there
For non-absolutely time critical functions, check flags instead of interupts.

PseudoCode:

```
SetUpModules()
      while (FOREVER)
            if I2C message received
                  run corresponding function
            if RS232 message received
                  run corresponding function
            main loop, repeated function (reconfigurable by I2C/Rs232)
            wait for loop timer to expire
      repeat
```

Microchip PICMicro Microcontrollers

The PIC microcontroller is a robust yet lightweight integrated system useful for meeting the distributed computing needs of the Emerald project. Its RISC instruction set makes programming somewhat simle, and its on board EPROM reduces the overall system complexity. Integrated EPROM also simplifies programming, new code is just written on a blank PIC and plugged into the system. The integrated I2C bus I/O allows for integration in a seamless manner with the main backplane of the spacecraft. The Harvard architecture of the PIC allows its aformentioned RISC instructions to be executed quickly, all but branch instruction execute in one clock cycle (four clock tics, or 5 MHz for a 20 MHz crystal). Finally there are rumors that the PIC is somewhat radiation tolerant. It is said that there were PICs present on the very successful Mars pathfinder expedition.

Shown in the following pages is the schematic and setup for a PIC protoboard, which allows for testing of PIC modules, providing access to all pins, and space for further development. The typical development path has used the PICs integrated serial (TTL level) interface, through a level converter, as a debug port, with the integrated I2C bus used for command and data handling and command. Final system integration will retain the debug port, with the DB-9 connector replaced with a 1/8" stereo jack. A new proto board is best tested with a pre-programmed PIC which is known to work. Stable code can be found in another place in this document. Malfunctionning boards often have power or wiring problems. One other potential difficulty is an incorrectly setup programmer. The programmer must be set for the correct kind of PIC that it programs. Otherwise, it will not function correctly, but will still program (or appear to do so) the chips

General Overview

The PICMicro microcontroller is a robust yet lightweight integrated system useful for meeting the distributed computing needs of the Emerald project. Its RISC instruction set makes programming somewhat simle, and its on board EPROM reduces the overall system complexity. Integrated EPROM also simplifies programming, new code is just written on a blank PICMicro and plugged into the system. The integrated I2C bus I/O allows for integration in a seamless manner with the main backplane of the spacecraft. The Harvard architecture of the PICMicro allows its aformentioned RISC instructions to be executed quickly, all but branch instruction execute in one clock cycle (four clock tics, or 5 MHz for a 20 MHz crystal). Finally there are rumors that the PICMicro is somewhat radiation tolerant. It is said that there were PICs present on the very successful Mars pathfinder expedition.

Shown in the following pages is the schematic and setup for a PICMicro protoboard, which allows for testing of
PIC modules, providing access to all pins, and space for further development. The typical development path has used the PICMicros integrated serial (TTL level) interface, through a level converter, as a debug port, with the integrated I2C bus used for command and data handling and command. Final system integration will retain the debug port, with the DB-9 connector replaced with a 1/8" stereo jack. A new proto board is best tested with a pre-programmed PIC which is known to work. Stable code can be found in another place in this document. Malfunctionning boards often have power or wiring problems. One other potential difficulty is an incorrectly setup programmer. The programmer must be set for the correct kind of PICMicro to be programmed; otherwise, it will not function correctly, but will still program (or appear to do so) the chips.

I2C Bus

An in-depth discussion of the I2C protocol can be found in the various datasheets at the end of this document. A discussion of the higher-level protocol we designed for instruction passing appears elsewhere in the document.

Dallas 1-Wire Bus

Datasheets regarding the Dallas one-wire bus are found at the end of this document.

SRAM Interface

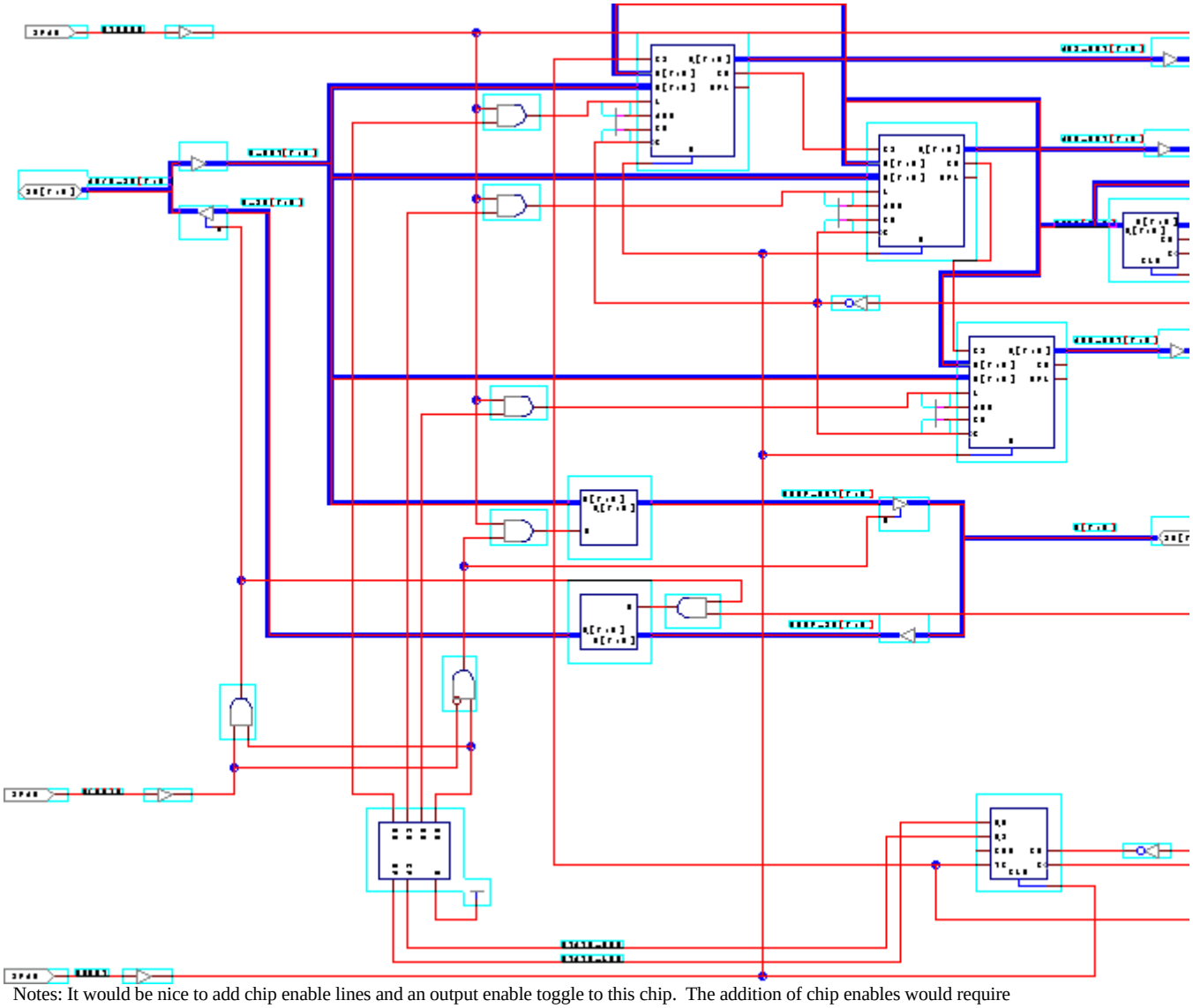
One of the short comings of the PICMicro microcontrollers is the small amount of random access memory (or data storage capability) that they posses. Within the family of PIC we are using (16C7x), the largest amount of RAM present on any device is no more than 200 bytes. This is a far cry from what is necessary for experimental data storage or telemetry. As a result, a method for interfacing with external SRAM's was designed.

The basic problem is the limited number of I/O pins available on the PICs. When the A bus is dedicated to analog-to-digital conversion and the C bus to communication (RS232, I2C, etc.), this leaves only the 19 pins of the B, D, and E busses. allowing for and 8 bit wide data bus, and 2 lines for R/W (read/write selection) and OE (output enable), this leaves 9 bits for address, or 512 bytes. The requirement for the VLF experiment (to use it as an example) is on the order of several megabytes.

Thus, a circuit has been designed to mulitplex address and data onto one 8 bit bus. The resulting logic (when implemented on an FPGA) requires 3 control lines, and 8 data/address lines (probably the D and E busses), and can address as much memory as desired. Then general structure for this design is three 8 bit latches loaded one at a time for address, and a feed through data bus. This serialiazation of the bus necessitates that one of the control pins act as a clock/strobe to mark each piece of data as it exits the PIC. For increased throughput, the address latch increments on successive memory accesses, so that it needs only be fed once for each block of data. This necessitates a second one of the aforementioned control lines, a reset pin, which initializes the address latch to be ready to accept data again. Current implementation requires an output enable line from the PIC: a better implementation should not require this. The final control pin tells the direction of the data bus (read or write).

The schematic has been tested (and works) in beta form, and is shown below.

Bus Multiplexer



Notes: It would be nice to add chip enable lines and an output enable toggle to this chip. The addition of chip enables would require tying a decoder to the address bus. An output enable output would require more complex logic, perhaps the strobe line could work as a OE as well, or maybe inverting the OE line would work. Not sure about that.

EMERALD: Electrical Interface Document

First Edition: 4-1-99

Standard Connectors

Between subsystem boxes we plan to use D sub-miniature standard density connectors, aka DB connectors. they are available in 9, 15, 25, 37 and even 50 pins. Hopefully we can stick with the smaller ones. Just to be consistent, we will use the pins for describing gender.

Standard Subsystem interface:

Each subsystem gets 2 DB9's, one for power and one for serial communication and a stereo plug for RS232 debugging..

DB9 Pinout

Pinouts:

pin	power	serial
1	+5V	I2C: SDA
2	+5V	I2C: SCL
3	+8/12V	interrupt request
4	unregulated	Dallas Power/Data
5	unregulated	
6	clean ground	SDA shield
7	clean ground	SCL shield
8	dirty ground	Dallas Gnd
9	dirty ground	

Wires:

To make our lives easier we also have standard colors for wires between boards and boxes:

Wire Colors

Wires (off board) will be AWG22 stranded wires.
All of the prototype colors are available from DigiKey

Signal	ProtoColor	Flight Color
Power		
+5V	red	TBD
+8/12V	yellow	TBD
unreg	orange	TBD
Grounds		
clean (digital ground)	green	TBD
dirty (motors, noisy stuff)	black	TBD
shield (only connected at one end, to shield noise)	gray	TBD
Signals		
I2C (both data and clock)	blue	TBD
Dallas 1 wire	violet	TBD

All other data lines can use any other color. Currently there are only two other colors: White and Brown. This isn't many, but:

- many of the signals are covered by the serial bus
- various combinations of twisting these wires, say with ground and shield as well as other data lines can give many possibilities
- it will also be possible to use a marker to make white with a black stripe (or a red stripe, etc.)

- we hope to find more colors… if you see some let us know

Action Items

Obtain Flash PICs (16F873/874)

Bug List and Bug Fixes

Lesson Learned

Bryan Palmintier (6/1/99)

- shouldn't do systems lead and subsystem lead at the same time
- too much work
- stepping down as subsystem lead
- first experience with CPU
- experience with embedded systems and PICs, though
- read through 68332 docs
- familiar with OS and HW
- happy with PIC progress
- definitely not as much progress as hoped with CPU
- expected:
- CPU
- I^2C interface
- basic OS functionality (file saving, etc.)
- problem was because of teamwork and also because of complexity of issues the subsystem had to deal with
- ideally want to settle on HW first, then develop; but putting off development too long to decide on HW issue is bad
- we have made some progress in terms of limiting down the CPU options
- there's a trade-off between having to get the project done and exploring options
- if could do the quarter over again:
- not sure what he'd do differently ("ask me again next quarter")
- can't use printf for debugging
- must take care with interrupts and loops
- no vectored interrupts in PICs
- no dual level of interrupts (e.g. if there is a printf in an I^2C interrupt, will almost certainly miss the next timer interrupt)
- not everything can be decided by consensus if you want to stay on schedule
- in a multi-programmer project, need to have standards and expectations defined:
- skeleton file
- capitalization
- tested, modular code
- these things should have been done/ready last quarter
- try to keep the formal part of meetings short
- subsystem lead should have outline for design review
- worthwhile to define (implement) interfaces up front (e.g. I^2C)
- should already be using it by now—get with it!
- because of ROM limitation, will need to start looking at more compact code
- modular programming with a group is tricky and requires planning

Jim Cooley (6/2/99)

- how to burn EEPROMs that are doubled (high and low)
- must follow certain steps
- on Epic Win, always have Options→Read File Before Programming checked
- working on two subsystems (CPU and Communications) helped because there are many interrelated issues (also on Orion comm/databus system)
- learned a lot about how long pre-planning takes
- thought we'd have more HW built, but it took too long to make decisions
- not sure why that was
- takes even longer to order and obtain the HW

Lee Boyce (6/3/99)

- can only be effective on one subsystem at a time
- too many meetings for the amount of progress we had to show for them
- need a plan up front rather than a plan by consensus
- a bad decision is better than no decision
- subsystem leads need to be able to make decisions (need the authority/empowerment to do so)

Vlad Beffa (6/4/99)

- frustrated by lack of expertise in the CPU/HW/OS issues and by inability to get things done/lack of progress made
- frustrated by limited amount of coding don

Appendix A – Contacts

Lee Boyce

Email: boyce@stanford.edu

Hm: (650) 322-3965

Wk: (650) 725-3310

Jim Cooley

Email: jcooley@stanford.edu

Hm: (650) 497-1528

Wk: (650) 723-5450

Bryan Palmintier

Email: bryanp@stanford.edu

Caleb Tilo Kemere

Email:
ckemere@leland.stanford.edu

whereabouts unknown

Vlad Beffa

Email: vbeffa@leland.stanford.edu

Appendix B – CCS Compiler Subtleties