SCUBA2 Multi-Channel Electronics:

Development of Power Supply Controller Card Firmware

Prepared By:

Stuart Hadfield, 45504008

UBC Physics and Astronomy - SCUBA 2

October 24, 2006

ABSTRACT

" SCUBA2 Multi-Channel Electronics:
Development of Power Supply Controller Card Firmware"

By Stuart Hadfield

This report serves to document the development of firmware for the Power Supply Controller Card (PSUC), as part of the Submillimetre Common-User Bolometer Array 2 (SCUBA2) camera Multi-Channel Electronics (MCE). This firmware was completed in the summer of 2006, in conjunction with SCUBA2 MCE development at UBC.

The objective of the work described in this report was to develop and test PSUC firmware which conformed to initial specifications. This firmware was implemented in C and verified to work through hardware testing. The majority of the included C code was written by the author.

This report focuses on the PSUC firmware, which is only a small part of the SCUBA2 MCE project as UBC. Background of the SCUBA2 project, the MCE, and the PSU are briefly touched on. For a more in-depth look at the SCUBA2 MCE and its applications, refer to the documents and links listed in the *References* section of this report.

[1]Table of Contents

[1] Page numbering omitted due to large number of integrated files.

1.0 INTRODUCTION – SCUBA2

Many unresolved issues in modern cosmology relate directly to submillimetre astronomy[2], specifically the formation of stars and galaxies in the early universe. The extreme cold associated with the earliest evolutionary stages of the universe corresponds to radiation precisely in this wavelength band.

The main limitation to ground based submillimetre astronomy is emission, attenuation, and noise from the atmosphere. The submillimetre wavelength region contains numerous water vapor absorption bands; thus the ideal observation site is cool, dry and far from urban centers. One of the few such sites identified in the world is Mauna Kea in Hawaii, home of the James Clerk Maxwell Telescope (JCMT). The original Submillimetre Common-User Bolometer Array (SCUBA) was delivered to the JCMT in 1996 and retired from service in 2005. It has since been referred to as "the most successful submillimetre astronomy instrument" of the past decade[3]. Functioning as both a camera and a photometer, SCUBA was a substantial improvement over previous submillimetre instruments in both sensitivity and productivity. A 2001 survey by the Space Telescope Science Institute revealed that scientific results from SCUBA have been cited nearly as often as those from the Hubble Space Telescope, and much more so than those from any other ground based project.

SCUBA2 represents the hopes of an international consortium of physicists and engineers to capitalize even further on this tremendous success with the next-generation of SCUBA. The development of SCUBA2 is a collaborative effort involving the University of British Columbia, the UK Astronomy Technology Centre (ATC), the US National Institute of Standards and Technology (NIST), the Astronomy Instrumentation Group at the University of Wales at Cardiff, the

[2] Submillimetre astronomy generally concerns wavelengths between the orders of hundreds of microns and millimeters.
[3] [1]

Scottish Microelectronics Centre at the University of Edinburgh, the Joint Astronomy Centre (JAC), and several other contributing groups.

With a greatly increased field of view and sky background limited sensitivity, SCUBA2 will map large areas of sky up to 1000 times faster than SCUBA. Incorporating state-of-the-art technology will allow the realisation of the first large-format "CCD-like" camera for submillimetre astronomy. This represents a major improvement from currently available submillimetre instruments and facilities.

SCUBA 2 will be delivered to JCMT in April 2007. Other future SCUBA2 applications include examination of large-scale clustering in the universe, galaxy evolution and populations, and the Sunyaev-Zel'dovich effect[4], among others[5]. For an in-depth look at SCUBA2 the reader is referred to the References section of this report.

1.1 SCUBA2 MULTI-CHANNEL ELECTRONICS

The SCUBA2 team at UBC, based out of the Department of Physics and Astronomy and lead by Dr. Mark Halpern, is responsible for design, implementation, testing, and delivery of the SCUBA2 Multi-Channel Electronics (MCE). The following is a *brief* description of the MCE.

* * * * *

A full SCUBA2 bolometer array[6] consists of four sub-arrays butted together to give the full field-of-view, with each sub-array connected to one box of MCE. Each MCE box, or 'sub-rack', is in turn connected via a fiber optic cable to

[4] Polarimeters can be used with SCUBA2 to determine interstellar magnetic field geometries.
[5] [1]
[6] SCUBA2 detects radiation through a bolometer consisting of multiplexed a SQUID array. See references.

a single data acquisition computer system running (real-time) Linux.

Each sub-rack houses a complete set of sub-array electronics and the necessary power supplies. These electronics are implemented in the form of modular cards to facilitate independent design, testing, and revision of each functional card. Cards have been built on FPGA based designs to allow greater flexibility and functionality. Table 1 below gives a brief description of each card.

Card	Brief Description
Clock Card	Handles all comm. between subrack and outside PC and comm. between cards
Adress Card	Provides analog row select signals for 'turning on' each row of SQUID multiplexer
Bias Card	Provides 32 channels of bias voltages for the SQUIDs, and one differential channel
Readout Card	Amplifies and digitizes output from the SQUID series array
Power Supply Card	Provides supply voltages for cards in the subrack
PS Controller Card	Monitors Power Supply behavior and communicates between PS and Clock Card

Table 1: MCE Cards Description

Each sub-rack houses a backplane PCB which facilitates card connections of two types:

i) Instrument Backplane: provides connections to the relevant signals in the cryostat.

ii) Bus Backplane: provides common power and the necessary signal interconnections between each card in the sub-rack.

Figure 1 below shows a partially populated sub-rack. The sub-rack backplane can be seen at the rear of the sub-rack. A more detailed description of the MCE can be found in [2], [4], and [5].

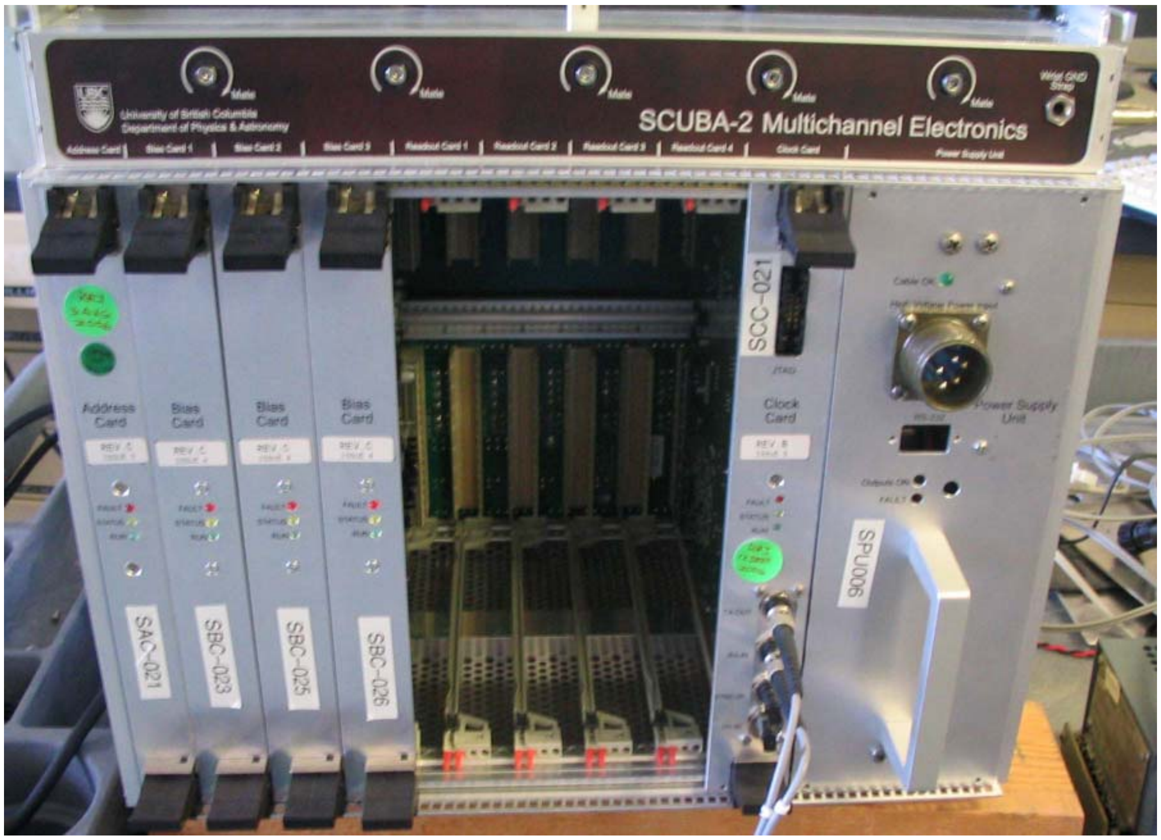

SCUBA-2 Multichannel Electronics
University of British Columbia
Department of Physics & Astronomy
Address Card
Bias Card
Clock Card
Power Supply Unit
SAC-021
SBC-023
SBC-025
SBC-026
SCC-021
SPU006

Figure 1: Sub-rack with 6 of 10 cards in place, PSU on far right.

1.2 POWER SUPPLY UNIT

The MCE Power Supply Card (PSU) was designed to allow for external monitoring and control. Its circuit board contains a 40-pin header, to which the Power Supply Controller Card (PSUC) connects. Strictly speaking, the Power Supply (PS) and Power Supply Controller Card are separate circuits which together, when connected and mechanically mounted in the card enclosure, form the PSU. However, in practice the Power Supply circuit and the combined PS/PSUC unit are in general interchangeably referred to as the PSU.

The PSU connects to the backplane via a 34-pin Winchester connector. This connector consists of power lines to each card, as illustrated in Figure 2 below, and control signals to the Clock Card (CC).

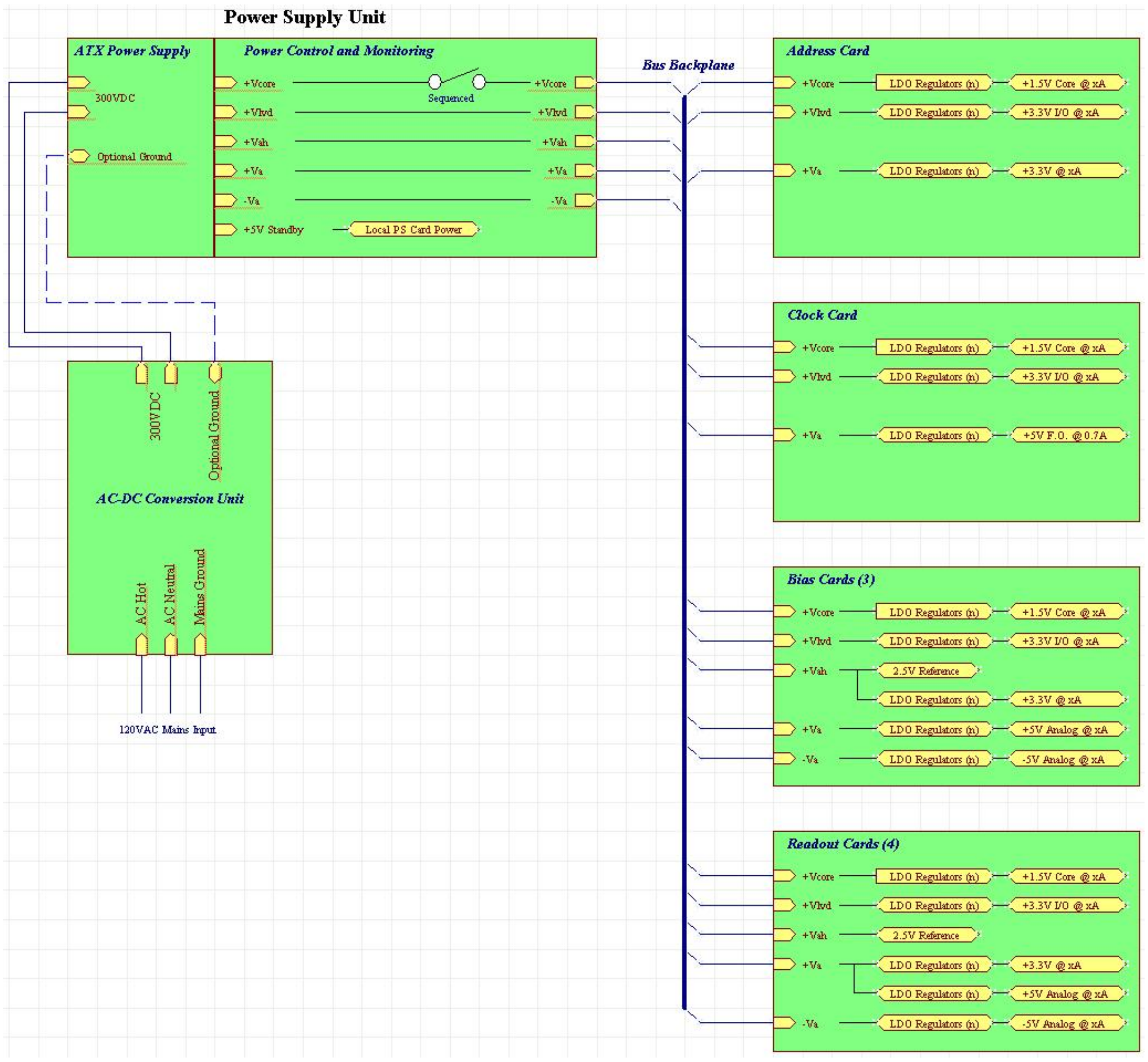


Figure 2: Bus Backplane Power Conections

Each card in the sub-rack is powered via rails running along the subrack blackplane. The PSU outputs 5 different DC voltages to the subrack, as shown in table 2:

PSU Output	Nominal Voltage (V)	Nominal Current (A)
Vlvd	4.5	4
Vcore	3	13

Vha	10.1	0.15
+Va	6.2	15
-Va	-6.6	2

Table 2: PSU Output Voltages

Control signals from the Clock Card pass through the Winchester Connector and connect directly to the PSU. The rainbow set of thin wires in Fig. 3 below are exactly these control signal. The single colored groups of thicker wires are the power lines, with the number of wires per group proportional to the nominal current output on each line[7] (more wires correspond to less thermal loss for large currents due to lower resistance).

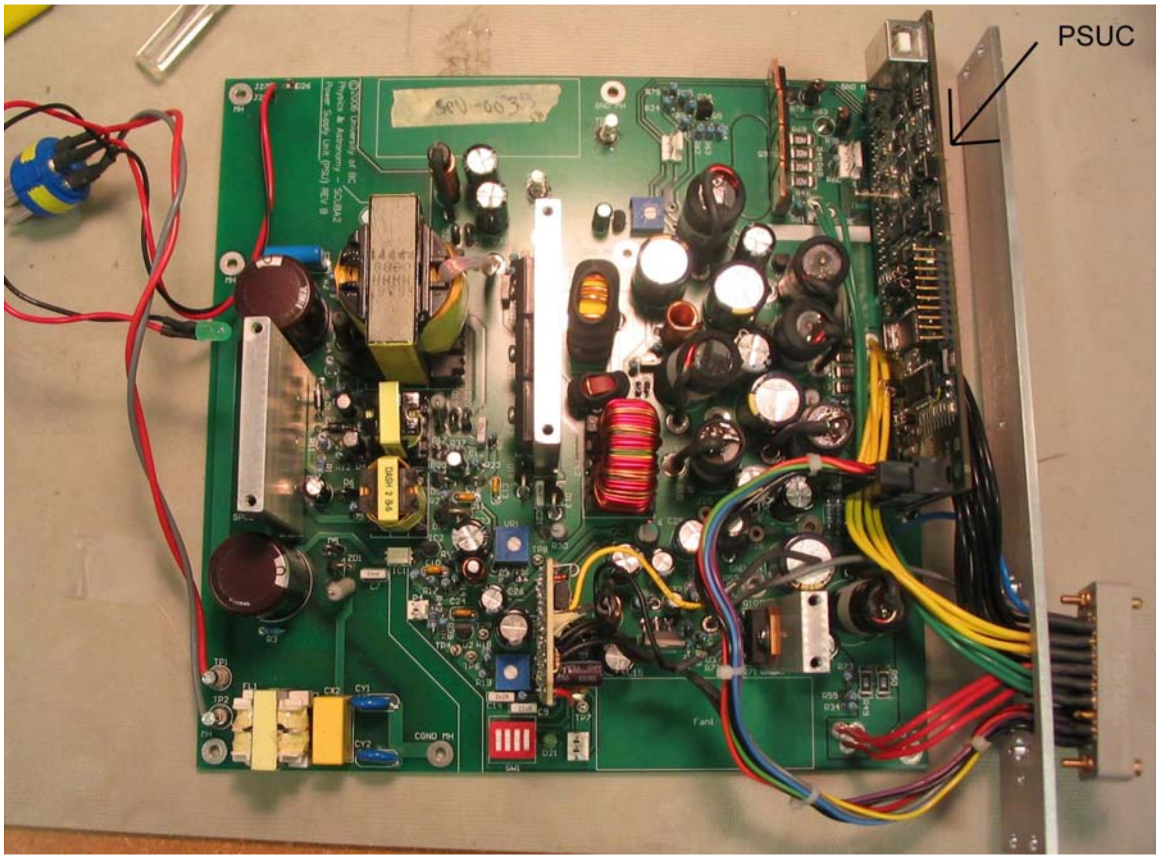


Figure 3: PSU Circuit Board with PSUC attached
The PS circuit shown here was built and tested by the author.

[7] Table 2 shows a two orders of magnitude difference in nominal currents

The PSU/PSUC circuit is enclosed in a large metal enclosure for both mechanical function and electromagnetic shielding, as seen in figure 4. This unit slides into the subrack and is secured in place with screws.

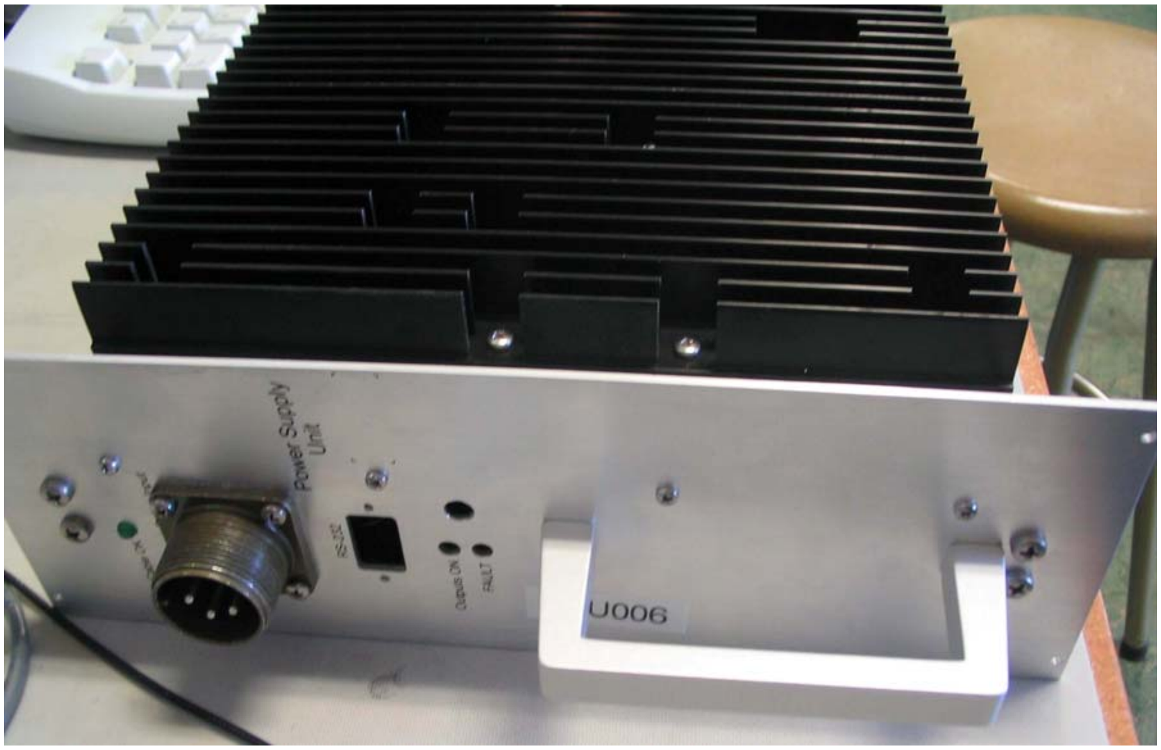


Figure 4: PSU Card (PSU and PSUC PCBs connected inside enclosure)
Black structure is a large heatsink

The PSU is itself powered by a 300V DC input through a cable from an isolated AC/DC Conversion Unit (ACDCCU). In the original PSU design these two units were integrated, but later separated due to noise issues resulting from the presence of AC signals inside the subrack.

The various connections of the PSU are summarized[8] in figure 5.

[8] Refer to PSU schematic (Appendix 5.2)

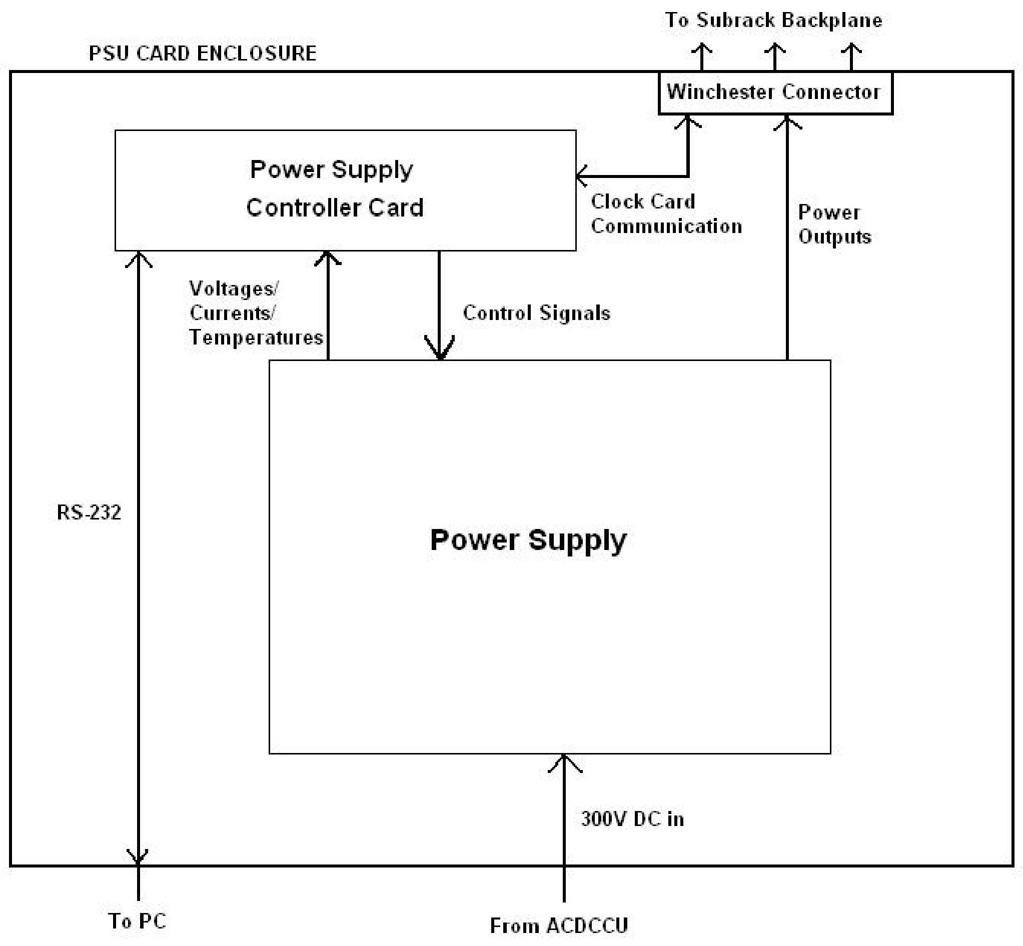


Figure 5: PSU Connection Diagram

1.2.1 PSUC DESIGN

The PSUC design is based around the ATMEL AT89C5131AM[9] 8-bit microcontroller, which is a derivative of the popular 8051 species of microcontrollers commonly found in embedded systems. A 64-pin package is used which provides a large number of inputs and outputs. Clocking is controlled by an on board 24 Mhz crystal oscillator. The necessary buffers, signal paths, and analog to digital converters (ADCs) are all integrated with the AT89 on the

[9] Abbreviated AT89

PSU PCB[10].

The PSUC monitors outputs voltages, output currents, and internal temperatures of the PSU. Each voltage signal from the PSU is fed through a low pass filter and then scaled using a non-inverting amplifier[11] before reaching the input of the ADC. Each voltage is scaled as to (ideally) input 3.0V to the ADC when at nominal value. As the maximum input readable by the MAX1271 ADC is 4.096V[12], this corresponds to scaling to 73.2% of the ADC's full range. This implies that if all voltages are at their nominal level, the ADC should report out at 73.2% of its maximal value (i.e. should report = 0.732 * 0xFFF = 0xBB6)[13] on each channel. A grounded ADC input is also read to measure any ground offset. This channel is the only channel read in bi-polar mode (i.e. read in negative voltage range also).

The PSU output currents are measured as follows: Each PSU output voltage signal runs across a very small 'shunt' resistance[14]. The PSUC then receives a voltage signal from both sides of this resistance, which is fed into a differential amplifier followed by a non-inverting scaling amplifier. Current across the shunts is thus inferred from the voltage difference and known resistances values. Current measurements are scaled to 2.5V, or equivalently 61% of full range (i.e. at nominal current levels, ADCs will output 0.61 * 0xFFF = 0x9C2 on each channel)[15].

It is important to note that precision is not important for voltage, current, and temperature readings because their values are used only for determining whether the PSU/Subrack is operating within an acceptable range. Thus these values are interpreted qualitatively more so than quantitatively. It is more

[10] Refer to PSUC schematic (Appendix 5.3)
[11] Except –Va voltage which *is* inverted so that all voltage readings are positive. Refer to page 2 of PSUC schematic (Appendix 5.3)
[12] See MAX1271 datasheet ([11])
[13] MAX1271 has 12-bit output resolution so maximum output is 0x0FFF.
[14] Refer to page 4 of PSU schematic (Appendix 5.2)
[15] Refer to page 3 of PSUC schematic (Appendix 5.3)

important to know if the subrack is ‘too hot’ than to know the exact temperature to several decimal places. As a result, the PSUC components were not designed for ultra-precision, but rather for an acceptable accuracy of ~ +/- 5%[16]. For example, simple op-amp based amplifiers and 1% resistors are used in most cases.

The PSU design also includes its own ID/Temperature Sensor and 128kb ROM storage device, as well as several currently unused signal paths. These are included to allow for future expansion without hardware revision.

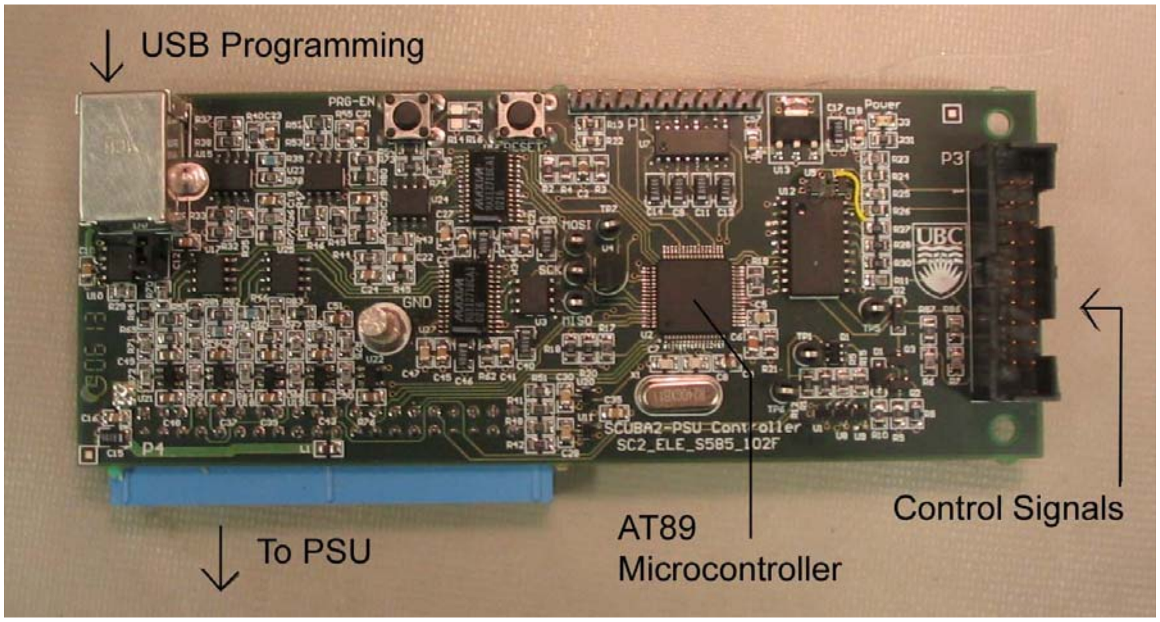
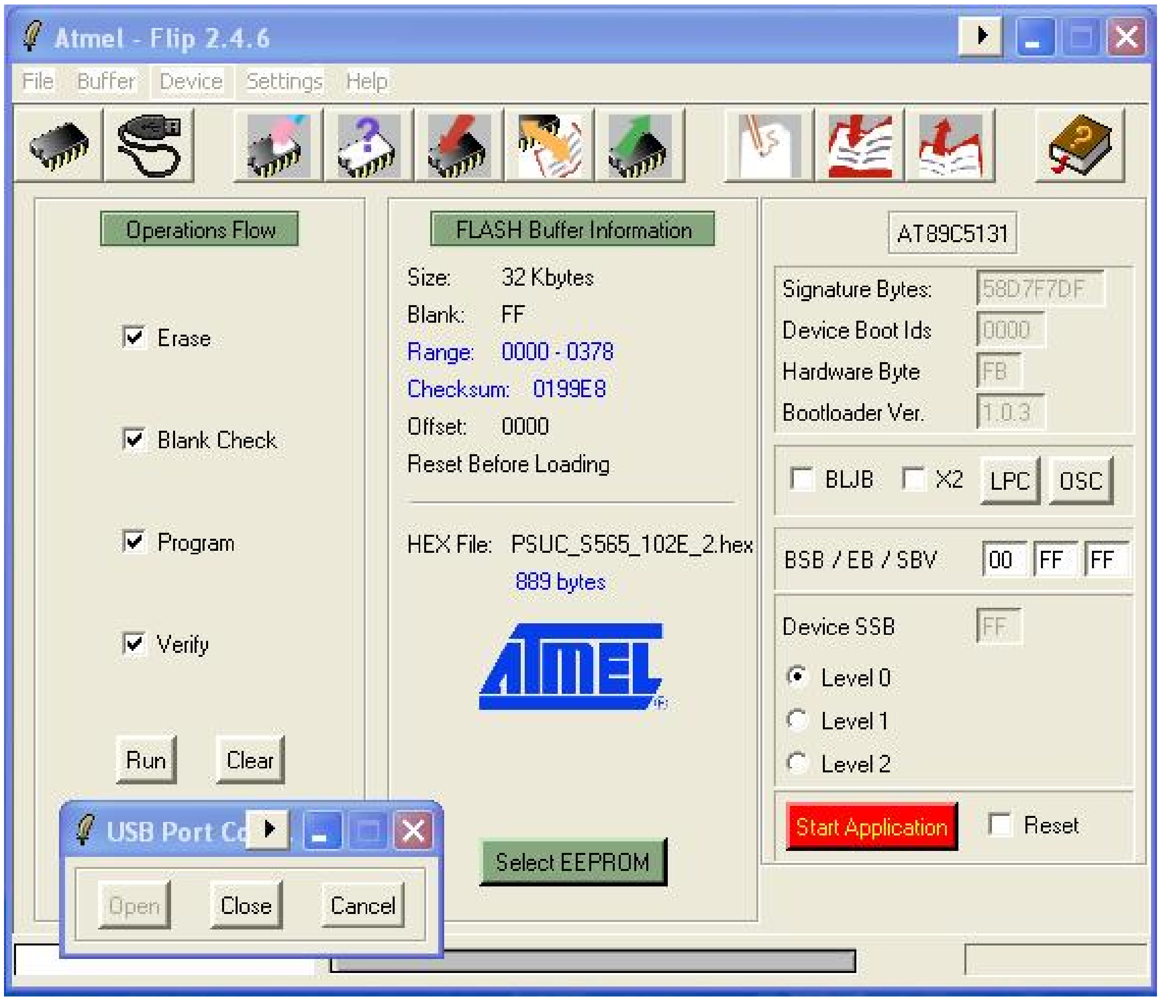


Figure 6: PSUC Circuit

All firmware for the AT89 microcontroller was written in C, and compiled using the Keil uVision software package. This program includes built-in functionality and interfacing for the AT89 through its library files. Once a source code has been compiled and built into a hex file (binary code), it is ready for loading into the AT89. The AT89 is programmed through its USB input[17] using an external computer. Atmel, the manufacturer of the AT89, provides a software

[16] See section 2.2.1 for PSUC calibration results.
[17] Note USB programming only physically possible when PSUC is not inside PSU enclosure (or alternatively lid of PSU enclosure has been removed).

package for programming the microcontroller, FLIP 2.4.6, which was used for all programming and testing. A screenshot of FLIP is shown in figure 7 below.



Operations Flow

- ✓ Erase
- ✓ Blank Check
- ✓ Program
- ✓ Verify

FLASH Buffer Information

Size:	32 Kbytes
Blank:	FF
Range:	0000 - 0378
Checksum:	0199E8
Offset:	0000
Reset Before Loading	
HEX File:	PSUC_S565_102E_2.hex
	889 bytes

AT89C5131

Signature Bytes:	58D7F7DF
Device Boot Ids	0000
Hardware Byte	FB
Bootloader Ver.	1.0.3
BSB / EB / SBV	00 FF FF
Device SSB	FF

☐ BLJB ☐ X2

[✓ Level 0] Level 1 Level 2

☐ Reset

Figure 7: ATMEL FLIP Program Screenshot

The PSUC packages the currents status of the PSU in a 36 byte data block which is sent to the CC upon request. While transmitting the datablock to the CC, the PSUC simultaneously receives back a command. Four different commands are possible in the basic specifications[18] for CC-PSU communication and are described in table 3 below.

Command	Codeword	PSU Action

[18] Refer to Appendix 5.1

Reset MCE	'RM'	Assert subrack reset signal
Turn Off	'TO'	Turn off PSU
Cycle Power	'CP'	Turn off PSU, wait, Turn on PSU
Status Block Request	0x00 (NULL)	Do Nothing, send PSU data block to CC on next request (default 'command')

Table 3: Clock Card commands for PSUC

The Clock Card decides when to issue a command based on the data it receives from the PSUC. The CC continually polls the PSUC for the current status of the PSU (about once every two seconds) and issues commands accordingly. For example, if the PSU temperature rises to potentially damaging levels or a given voltage/current is outside of acceptable ranges, the CC will issue a 'Turn Off' Command to the PSU. The PSUC only reports the PSU status to the CC and is not responsible for decisions as to when to act on this data.

1.2.2 SERIAL PERIPHERAL INTERFACE

The Clock Card communicates with the PSUC using a custom Serial Peripheral Interface (SPI). SPI is a loose standard for device communication using a clocked serial bitstream. It is implemented with a bus of three signals:

SCLK - Serial Clock

MISO – Master Input Slave Output

MOSI – Master Output Slave Input

The AT89 is the master for all SPI communication, and therefore it controls the SCLK line. The MOSI/MISO lines each uni-directionally send data, clocked out/in by SCLK.

The SPI bus is also used for communication with the ADCs and the EEPROM onboard the PSUC. Each device shares the same SPI signal bus and is a slave[19] to the AT89. A dedicated Slave Select (SS) line to each device is

[19] Even thought the Clock Card sends commands to the PSUC, it is a slave to the AT89 w.r.t. SPI.

used to control which slave device is communicating over the SPI bus at a given time.

The Clock Card initiates communication with the PSUC by asserting the Service Request (SREQ) signal. When the PSUC is ready to respond, it asserts the Clock Card Slave Select line (CCSS). The PSUC does not necessarily respond immediately as it may be busy with data polling or command execution. Assertion of CCSS is immediately followed by transaction of data over the MSIO/MOSI lines. The PSU sends the full 36 byte data block by default each time. During the first 6 bytes exchanged, the CC sends a two byte command to the PSUC, repeated three times to avoid error. While the remainder of the datablock is being sent, the three commands received are checked, and the ACK/NAK byte (within the <u>same</u> datablock transmission) is updated to indicate a valid command received.

The default command, which is communicated by the CC holding the MISO line low for the first 48 SCLK cycles (i.e. sending six bytes of zeros), is the Send Data Block command. This ensures that the datablock is sent each time. If an erroneous command is received, the PSU will still send a datablock on the next asserted SREQ.

A big advantage of the AT89 is its built in SPI functionality. This greatly simplifies the implementation of CC-PS communication. However, the built in SPI functionality presents certain restrictions which prevents it from being used for communication between the AT89 and the ADCs on the PSUC. Instead, the SPI must be manually controlled by the AT89. This is discussed further in section 2.2.1.

Refer to the appendix “SPI Communications between CC and PS” for a further examination of the CC-PS communication protocol.

2.0 PSUC FIRMWARE DEVELOPMENT

The PSUC firmware is very different from any other firmware used in SCUBA2. The other cards are designed with FPGAs and are ‘programmed’ in VHDL, a hardware description language not a programming language. Thus the firmware for the other cards is truly parallel, where as the PSUC firmware executes instructions sequentially.

2.1 FIRMWARE DESIGN OVERVIEW

The firmware was separated into files for interfacing with each device on the PSUC. Where necessary, header and source code was separated into different files to increase re-usability. The resulting source files are listed in table 4 below.

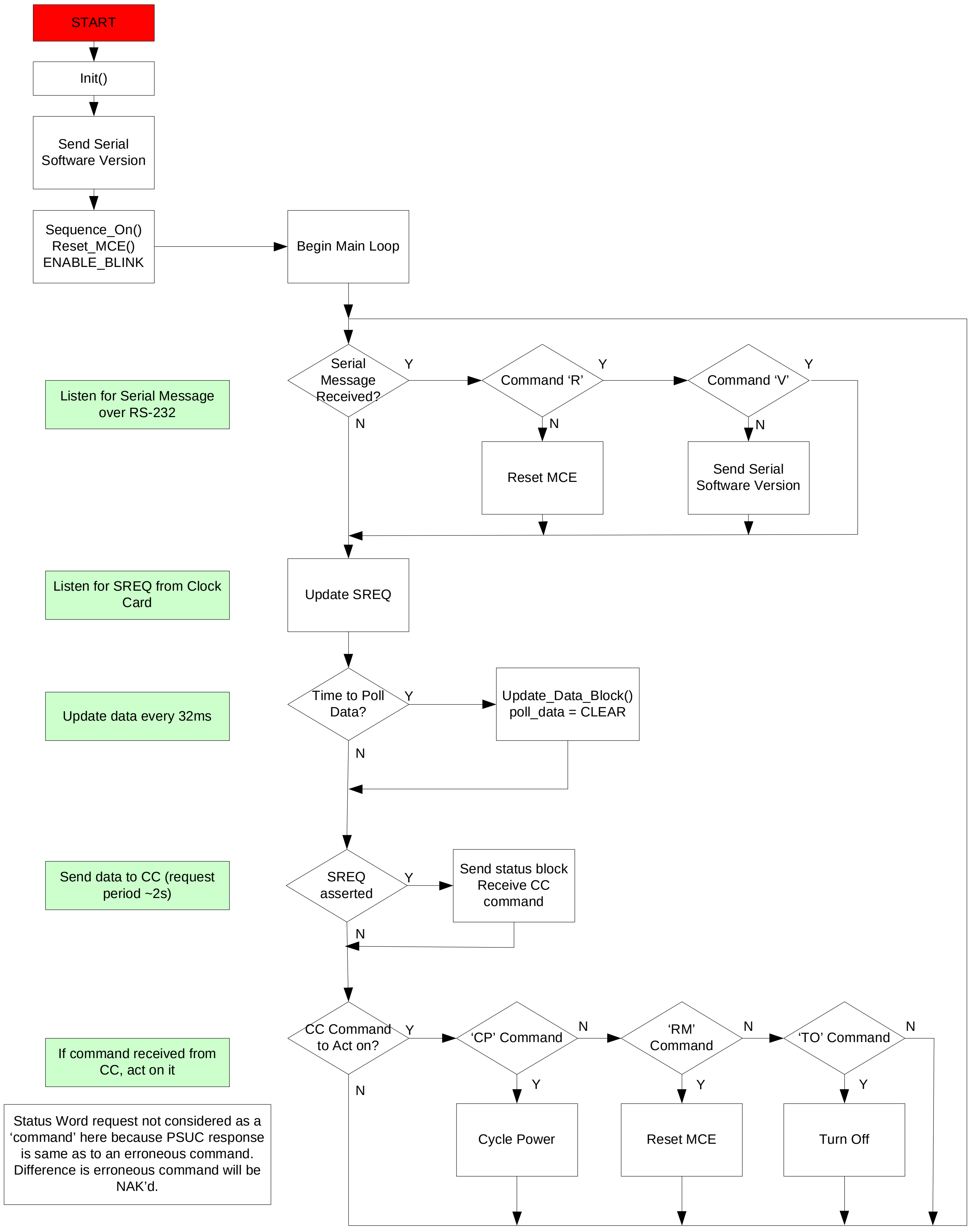
File(s)	Description
SCUBA2PS.c, SCUBA2PS.h	Main Program
IO.h	Input/Output Settings and Global Variables
DS18S20.c, DS18S20.h	DS18S20 ID/Temperature Sensor Interface
MAX1271.c	MAX1271 ADC Interface
EEPROM.c	EEPROM Interface

Table 4: Source Code File Summary

The main program is responsible for regularly updating the PSU data block. This involves polling the voltage, current and temperature values every 32mS. The datablock is then sent to the PSU on a request from the CC, which occurs roughly every two seconds. If a valid command is received from the CC, the PSUC acts on it immediately after it finishes the data bock transaction. Successful receipt of a received command is indicated to the CC by the value of the ACK/NAK byte in the datablock. Additionally, the PSUC listens for commands over the RS-232 serial input, but these occur very infrequently.

The high level operation of the PSUC program is illustrated in figure 8 below.

Figure 8: SCUBAPS.c - High Level Flow Chart


START
Init()
Send Serial Software Version
Sequence_On()
Reset_MCE()
ENABLE_BLINK
Begin Main Loop
Listen for Serial Message over RS-232
Serial Message Received?
Y
N
Command 'R'
Y
N
Reset MCE
Command 'V'
Y
N
Send Serial Software Version
Listen for SREQ from Clock Card
Update SREQ
Update data every 32ms
Time to Poll Data?
Y
N
Update_Data_Block()
poll_data = CLEAR
Send data to CC (request period ~2s)
SREQ asserted
Y
N
Send status block
Receive CC command
If command received from CC, act on it
CC Command to Act on?
Y
N
'CP' Command
N
Y
Cycle Power
'RM' Command
N
Y
Reset MCE
'TO' Command
N
Y
Turn Off
Status Word request not considered as a 'command' here because PSUC response is same as to an erroneous command. Difference is erroneous command will be NAK'd.

2.2 DEVICE INTERFACING

Interfacing functionality for the temperature sensors, ADCs, and EEPROM are each implemented in their own source files.

2.2.1 MAX1271 12-bit ADC

The Maxim MAX1271[1] is an eight channel ADC with 12-bit resolution. The PSUC has two built in MAXs, one each for voltage and current measurements respectively.

The MAX communicates with the AT89 over the SPI bus. The protocol is as follows: The AT89 sends the MAX a control byte over the first 8 SCLK cycles. This control word specifies the data acquisition mode and channel. The next five clock cycles are used by the MAX to convert the specified analog input to a 12-bit digital reading using a standard 'track and hold' technique. Transmission of this data to the AT89 begins on the following clock edge.

The total transaction of data request to receipt thus takes a total of 25 clock cycles. Unfortunately, this is 1 bit too many from being three bytes even. As a result, the internal SPI functions of the AT89 were not used as they operate at the byte level. Instead, manual control of the SPI bus was implemented in order to have total control at the bit level. This is evident in the attached code MAX1271.c wherein SCLK in manually toggled, as opposed to SPI communication with the CC where SPI is accessed indirectly (automatically) by reading/writing to the appropriate buffer.

Manual implementation of the SPI protocol also allowed much more

[1] MAX1271 abbreviated MAX

flexibility in control of SCLK. This proved to be very important as the MAX has very strict timing parameters[2] which *must* be conformed to in order to guarantee accurate and reliable results. For example, in switching between two channels reading oppositely signed maximal voltages, enough settling time must be allowed to properly track the second voltage. The ADCs were tested independently of the PSU and the timing performance was verified.

In order to verify the accuracy of the PSUC voltage and current readings, calibration data was acquired and analyzed. A dummy test jig was setup so that CC could control and communicate with the PSUC while the PSU was outside of the subrack and not enclosed in its card casing. A dummy load was connected to the PSU outputs to simulate full-load conditions. Current and voltage measurements were taken by retrieving a data block from the PSUC and by physically measuring the output voltages and shunt resistance voltage drop on the PSU itself using a digital multimeter. The two values of each measurement were then compared, as summarized in table 5.

PSU Voltages - Measured with DMM vs. Measured with PSUC

Measured on PSU-005 board*

Vi	**Nominal V (V)**	**Nominal I (A)**	**dV[3] (mV)**	**Rshunt (mOhm)**	**Current (A)**	**Voltage (V)**	**Expected V reading (%)**	**Expected I reading (%)**
Vlvd	4.5	4	99	25	3.96	4.735	77.02267	60.39
Vha	10.1	0.15	51	500	0.102	10.12	73.34495	41.48
+Va	6.2	15	63	4	15.75	6.32	74.61677	64.05
-Va	-6.6	2	108	50	2.16	-6.54	72.53455	65.88
Vcore	3	13	61	5	12.2	2.75	67.1	57.24615

PSUC data block[4] received (hex):

[2] See MAX1271 datasheet ([11])
[3] dV is the voltage drop across the shunt resistance for the corresponding output voltage
[4] Refer to Appendix 5.1 to make sense of this datablock

0093C03E 22000016 00000FFC 0B130C4A 0BF20BE0 0A700963 09950696 0A0A0A96 000006F6

Read with PSUC

	Voltage Readings			**Current Readings**		
	Hex	Decimal	%	Hex[5]	Decimal	%
Vlvd	C4A	3146	76.8254	995	2453	59.90232
Vha	BF2	3058	74.67643	696	1686	41.17216
+Va	BE0	3040	74.23687	A0A	2570	62.75946
-Va	A70	2672	65.25031	A96	2710	66.17827
Vcore	B13	2835	69.23077	963	2403	58.68132

Table 5: Calibration Measurements for PSUC.
All values based on received datablock in table.

From this data, the difference in expected versus measured reading cans be calculated in terms of ADC reading percent difference :

Vi	**dV (%)**	**dI (%)**
Vlvd	0.1973	0.488
Vha	-1.331	0.308
+Va	0.3799	1.291
-Va	7.2842	0.298
Vcore	-2.131	1.435

Table 6: Percent Error (Difference) of Measured vs. Expected Voltage/Current Values

From table 6 it can be seen that all measured voltages and current values are within[6] +/- 2.5 % of their expected value. This is within the expected deviation and more than satisfactory for gauging whether voltages and currents are within acceptable ranges. It is important to note that this test only truly pertains to the specific prototype PSUC tested. Percent error of other PSUCs should be similar but will in general be different.

[5] Maximum ADC reading is 0xFFF.
[6] Large error on –Va is indicative of erroneous resistance values on scaling network for this line. This was later fixed in hardware.

2.2.2 DS18S20 ID / TEMPERATURE SENSOR

The Dallas Semiconductor DS18S20 [7] is a multipurpose digital thermometer. In addition to measuring temperature with 0.5^{o}C accuracy, each DS contains a unique 64-bit serial code which is used for identification purposes.

The DS18S20s built into the PSU and PSUC are connected directly to the AT89. In addition, DS's are exclusively used for additional temperature sensing within the PSU. This additional sensing was originally intended to be done with thermistors, but the design was later modified to use the DS instead due to the advantages of reading temperature directly in digital format.

The DS communicates via Dallas Semiconductor's proprietary '1-Wire Bus' protocol over a single data line. This protocol, allows for many '1-Wire Bus' devices to all use the same communication line, where each device is identified for communication using its own unique ID code. However, the design of the PSU dedicates a signal line for each DS so bus conflicts do not exist. The advantage of this is that no transaction handshaking (i.e. device ID specification) has to occur, greatly speeding up the process of reading temperature. The transaction for reading temperature then greatly simplifies to sending a 'Convert Temperature' command, followed by a 'Read Temperature' command. Commands are sent serially over the single bus line, where bit symbols are communicated by sending different combinations of pulses to distinguish between '1' and '0'. Refer to the DS18S20 datasheet and attached C code for the complete '1-Wire Bus' specifications.

2.2.3 EEPROM

The PSUC contains a single Atmel AT25128A EEPROM chip which

[7] DS18S20 abbreviated DS

provides 128kb of serially programmable memory, optimized for low-power applications. The EEPROM also communicates with the AT89 via the SPI bus. Memory is read to/ written from using a set of serial commands (see datasheet).

The initial specifications for CC-PSU communications did not involve the EEPROM[8]. As such, EEPROM functionality was not included in the initial version of the PSUC firmware of which this report is concerned. However, a basic interface for communicating with the EEPROM was implemented independently of the existence of specifications. An early version of this code is included for reference.

2.3 REMAINING DEVELOPMENT

There are several features which were not included in the original specifications but are being considered for future firmware upgrades:

Watchdog Timer – This functionality is responsible for resetting the PSUC firmware in the event of the microprocessor hanging or somehow arriving in a 'bad state'. Implemented by periodically clearing a counter which causes a reset on overflow, with the idea being the counter won't clear if the program is 'stuck' and will lead to a reset. This feature will greatly improve the robustness of the PSUC firmware and is very important for field applications where there will not necessarily be an operator nearby to reset the PSUC firmware if a glitch occurs for any reason.

EEPROM Support – Allow PSUC to store the last datablock values in the event of a received shutdown command or the watchdog timer expiring. This functionality will require additional commands between the CC and PSUC to be specified to allow for precise data storage and retrieval.

[8] Basic specifications in Appendix 5.1 contains no EEPROM interface specifications.

Measurement Averaging – The possibility has been raised of using time-averaging on the voltage and current measurements to negate the possibility of a voltage spike triggering a shutdown action. However, initial testing has shown reading accuracy to be within the expected range, and noise on the power line has not yet been an issue.

These features will be implemented as deemed necessary over the remainder of the fall 2006 term, following clarification of specifications.

3.0 PSUC FIRMWARE C CODE

All PSUC firmware was implemented in C and verified to work through hardware testing. Preliminary design was done by Tom Felton, UBC Physics Electronics Lab. All code written by the author except for parts of IO.H and SCUBA2PS.C.

3.1 SCUBA2PS.H

3.2 SCUBA2PS.C

3.3 IO.H

3.4 MAX1271.C

3.5 DS18S20.H

3.6 DS18S20.C

3.7 EEPROM.C

```
/*************************************************************************************/
/*  Scuba 2 Power Supply Controller - SC2_ELE_S565_102D
/*************************************************************************************/


/*****  Compiler Directives / File Inclusions *****/
#pragma db
#pragma small
#pragma rom (compact)
#pragma symbols
#include <at89c5131.h>
#include <stdio.h>
#include <intrins.h>
#include "io.h"                                 // contains IO port settings and global de
calarations
#include "MAX1271.c"                            // code for interfacing MAX1271 ADCs
#include "DS18S20.c"                            // code for interfacing with DS18S20 Digit
al ID / Temperature Sensor

// Memory Parameters
#define BUF_SIZE        10                      // received serial message buffer size (co
mmands always smaller than this length in bytes)

/*****  Function Prototypes *****/
// PSUC Initialization
void init(void);                                // initializes hardware and software varia
bles

// PSU Commands
void sequence_on(void);                         // powers on MCE
void sequence_off(void);                        // powers off MCE
void reset_MCE(void);                           // resets MCE
void cycle_power(void);                         // cycle MCE power
void send_psu_data_block (void);                // send PSU datablock to CC via SPI

// Timing Functions
void wait_time (unsigned char);                 // waits input*5ms
void wait_time_x2us_plus3(unsigned char);       // waits input*2us + 3us

// Send Serial Message
void snd_msg (char *);                          // sends serial message (RS232)

// PSU Data Block Functions
void update_data_block(void);                   // updates voltage/current/temperature rea
dings
void check_digit(void);                         // calculates basis for checksum (without
ACK/NAK added)
//unsigned char get_fan_speed(void);            // currently not implemented

// Command Parsing Functions
void parse_command(void);                       // reads CC command from first 6 bytes rec
eived from SPI transaction
bit commands_match (char *, char *, char *);    // checks command rcv'd in triplicate
bit command_valid (char *);                     // checks command rc'd is valid


/*********  Variables *********/
// Memory Blocks/Pointers
unsigned char idata ps_data_blk[CC_SPI_BLEN];   // PSU data for sending to CC - declared a
s idata to conserve memory space
unsigned char idata rcv_spi_blk[CC_SPI_BLEN];   // Received SPI data block (from CC)

char *cc_command;                               // Command (from CC) pointer
```

```
unsigned char idata sio_rxbuf[BUF_SIZE];        // Serial Received Data Buffer

// index/counter variables
unsigned char data spi_idx;                     // SPI Data Block Index
char data sio_rx_idx;                           // Serial Received Message Pointer
char *msg_ptr;                                  // Serial Message to Send Pointer
unsigned char data bcnt;                        // Count of Timer0 interrupts
unsigned char data num_T1_ints;                 // Number of Timer1 interrupts to allow be
fore setting timeup_T1
unsigned char data running_checksum;            // Running total for checksum byte

// Software flags
bit cc_spi;                                     // Indicates Service Request from CC
bit spi_complete;                               // Indicates transact. with CC complete
bit sio_msg_complete;                           // Indicates Serial message received
bit poll_data;                                  // Set when time to update data block
bit timeup_T1;                                  // Set on Timer1 expiration (overlow)

bit blink_en;                                   // Set to turn on LED blink while PSUC run
ning
bit temp1_present, temp2_present, temp3_present;// Indicates if DS18S20s temperature senso
rs actually connected


/********** PSU Data Block Settings  ***************/
// PSU Data Block POINTERS - defining this way prevents pointers from being reassigned dyn
amically
#define SILICON_ID          ps_data_blk         // Read from DS18S20 LS 32 bits of 48
#define SOFTWARE_VERSION    (ps_data_blk+4)     // Software Version
#define FAN1_TACH           (ps_data_blk+5)     // Fan 1 speed /16
#define FAN2_TACH           (ps_data_blk+6)     // Fan 2 speed /16
#define PSU_TEMP_1          (ps_data_blk+7)     // temperature 1 from DS18S20
#define PSU_TEMP_2          (ps_data_blk+8)     // temperature 2 from DS18S20
#define PSU_TEMP_3          (ps_data_blk+9)     // temperature 3 from DS18S20
#define ADC_OFFSET          (ps_data_blk+10)    // Grounded ADC input channel reading
#define V_VCORE             (ps_data_blk+12)    // +Vcore supply scaled 0 to +2V
#define V_VLVD              (ps_data_blk+14)    // +Vlvd supply scaled 0 to +2V
#define V_VAH               (ps_data_blk+16)    // +Vah supply scaled 0 to +2V
#define V_VA_PLUS           (ps_data_blk+18)    // +Va supply scaled 0 to +2V
#define V_VA_MINUS          (ps_data_blk+20)    // -Va supply scaled 0 to +2V
#define I_VCORE             (ps_data_blk+22)    // Current +Vcore supply scaled
#define I_VLVD              (ps_data_blk+24)    // Current +Vlvd supply scaled
#define I_VAH               (ps_data_blk+26)    // Current +Vah supply scaled
#define I_VA_PLUS           (ps_data_blk+28)    // Current +Va supply scaled
#define I_VA_MINUS          (ps_data_blk+30)    // Current -Va supply scaled
#define STATUS_WORD         (ps_data_blk+32)    // undefined place for status word
#define ACK_NAK             (ps_data_blk+34)    // either ACK or NAK
#define CHECK_BYTE          (ps_data_blk+35)    // checksum byte


/*******    Macros  *******/
// General Macros/Parameters
#define ENABLE_BLINK            blink_en = SET;
#define DISABLE_BLINK           blink_en = CLEAR;

// Checksum Total Function
#define COMPLETE_CHECKSUM       *CHECK_BYTE = ~(running_checksum + ps_data_blk[ACK_BYTE_PO
S]) + 1;
// 2's compliment, so CHECK_BYTE + all other bytes = 0
```

```
/************************************************************************************/
/*  Scuba 2 Power Supply Controller - SC2_ELE_S565_102D
        Stuart Hadfield - SCUBA2 - July 2006
/************************************************************************************/

/*********************************************************************************
Refer to the following Data Sheets:
        Processor - Atmel AT89C5131AM
        Temperature Monitoring & Silicon ID - Dallas DS18S20-PAR
        ADC for Power Supply Voltage Monitoring - Maxim MAX1270ACAI
        EEPROM non-volatile memory - Atmel AT25128A

    NOTE: THIS PROGRAM IS NO WHERE NEAR COMPLETE OR TESTED
    The I/O pins are set correctly for this version (revF).
********************************************************************************/

/********** Version 2.2 *****************
Polling and communication functionality tested and verified */


// Header File containing function prototypes and global variable declarations
#include "scuba2ps.h"

// Constant Variables
char code asc_version[] =  "\n\rPSUC v2.10\n\r\0";          // Software Version Serial Msg
char code software_version_byte = 0x22;                    // 1 byte Software Version


/****************************************************************************************
 *  Main Program               *
 *************************** */

main()
{
    // Initialize Hardware and Software Variables
    init();

    // Output Version on Serial Port
    snd_msg(asc_version);

    // Initial Power-Up
    sequence_on();
    ENABLE_BLINK;
    //reset_MCE();                      // removed to avoid subrack reset if PSUC firmware
 reset

    /***  Main Loop - Periodically update PSU data block, respond to CC / RS232 Commands
***/
    while(TRUE) {

        // Serial I/O message ready to parse
        if ( sio_msg_complete == SET ) {
            sio_msg_complete = CLEAR;
            sio_rx_idx = 0;                                 // reset message pointer
            switch ( sio_rxbuf[0] ) {                       // parse message
                case 'R':                                   // Reset MCE Command
                    reset_MCE();
                    break;

                case 'V':                                   // Respond with Software Version
                    snd_msg(asc_version);
                    break;
                case 'D':                               // Respond with PSU data block
                    snd_msg(ps_data_block);
                    break;

                default:
                    break;
            }
        }

        // Listen for data request from clock card
        cc_spi = ~SREQ;                                 // SREQ active low

        // Time to re-poll data
        if ( poll_data == SET ) {                       // poll rate ~ 3Hz, CC Req. ~ 0.5H
z
            update_data_block();
            poll_data = CLEAR;                          // Data Poll Complete
        }

        // Send data block if it has been requested
        if ( cc_spi == TRUE) {
            send_psu_data_block();                      // Time to send SPI Data to CC
            cc_spi = FALSE;                             // Data Block Trans. Complete

        }

        // Act on command from Clock Card
        if ( cc_command != NULL )
            switch ( *cc_command ) {                    // parse received command

                case 'C':                               // Cycle Power Command
                    cycle_power();
                    cc_command = NULL;
                    break;

                case 'R':                               // Reset MCE Command
                    reset_MCE();
                    cc_command = NULL;
                    break;

                case 'T':                               // Turn Off Command
                    sequence_off();
                    cc_command = NULL;
                    break;

                default:                                // Status Req. or erroneous comman
d
                    cc_command = NULL;                  // Difference is ACK/NAK.
                    break;
        }

    }
}

/*****************************************************************************************
 *  Initialize                *
 *************************** */

void init(void)
{
    int i = 0;                                          // temporary index

/**************     Hardware Setup      **************/
```

```
    // IO Port Setup 1=Input (or Special Function) 0=Output
    P0 = 0x60;      //0110 0000
    P1 = 0xed;      //1110 1101
    P2 = 0x3c;      //0011 1100
    P3 = 0xd7;      //1101 0111

    // SPI setup                                         // CS lines active low
    CS_EEPROM = 1;
    CS_VADC = 1;
    CS_IADC = 1;
    CCSS = 1;
    SREQ = 0;           // SREQ active low but this is needed to not overload buffer U5
    MISO = 1;
    MOSI= 1;


    // Counter/Timer 0 used as a Timer in Mode 1.  Interrupt Rate: 32mS
    TH0 = 0;
    TL0 = 135;
    TR0 = ON;                                            // start timer 0

    // Counter/Timer 1 used as a Timer in Mode 1.  Interrupt Rate: 5e-3 Sec
    TL1 = LS_RELOAD_5mS;
    TH1 = MS_RELOAD_5mS;
    TMOD = 0x11;

    //Serial I/O Setup:  Using Internal Baud Rate Generator on 89C5131A.  Set to Serial Mo
de 1 at 9600 Baud using 24MHz Clock
    SCON = 0x50;                // 0101 0000
    BDRCON = 0x1e;              // 0001 1110
    CKCON0 = 0x7f;              // X2 set but 12 clocks per peripheral cycle -> 500ns/tick
    PCON = 0x80;                // 1000 0000 Double Baud Rate all others default
    BRL = 100;                  // Baud rate reload - sets Baud rate to 9600

    // PCA Counter Init             // not implemented
    //  CKCON0 |= 0x20;             // sets to 500ns per PCA tick
    //  CMOD |= 0x81;               // 1000 0001 Set PCA to stop counting during idle mode
, disable PCA interrupts, and count Fclk-periph/6 (250ns period)
    //  CCON |= 0x01;               // enable PCA interrupts

    // LED Setup
    LEDCON = 0xfC;              // LED1-3 10mA Current Source
    LED_FAULT = 0;              // Off
    LED_STATUS = 1;             // Off
    LED_OUTON = 1;              // Off

    // SPI Setup - Sets up spi in master mode with Fclk Periph/16 as baud rate and without
 slave select pin.
    //SPCON = SPI_MSTR | SPI_EN | SPI_SSDIS | SPI_CPOL1 | SPI_1M5Hz;    // CPHA = 0, trans
fer on falling SCLK
    SPCON |= SPI_MSTR;          // Master mode
    //SPCON |= SPI_6MHz;        // Fclk Periph/4 (6MHz)
    SPCON |= SPI_1M5Hz;         // Fclk Periph/16 (1.5Mhz)
    SPCON &= SPI_CPOL0;         // CPOL = 0, Clk idle state 0
    SPCON &= SPI_CPHA0;         // CPHA = 0, sample data on Clk rising edge
    SPCON |= SPI_SSDIS;         // Disable SS
    SPCON |= SPI_EN;            // Run SPI

    // Interrupt Setup
    ES = 1;                     // Enable SIO Interrupts
    IEN1 |= 0x04;               // Enable SPI Interrupts
    ET0 = 1;                    // Enable Timer0 Interrupts
    ET1 = 1;                    // Enable Timer1 Interrupts
    EA = 1;                     // Enable Global Interrupts
//  EC = 1;                     // Enable all PCA Interrupts


/****************   Initialize Variables    ********************/

// Initialize flags
    poll_data = SET;            // Initial data poll
    cc_spi = CLEAR;             // Clear remaining flags
    spi_complete = CLEAR;       // SPI transmission/reception complete status bit
    sio_msg_complete = CLEAR;
    timeup_T1 = CLEAR;
    DISABLE_BLINK;              // Initially disable LED blink

// Initialize other variables
    spi_idx = 0;                // Reset pointer for SPI data output
    sio_rx_idx = 0;             // reset serial message pointer
    bcnt = 0;
    num_T1_ints = 0;
    running_checksum = 0;

// Initialize pointers
    cc_command = NULL;
    msg_ptr = NULL;

// Initialize data blocks to all zeros
    for(i=0; i < CC_SPI_BLEN; i++) {
            ps_data_blk[i] = 0;
            rcv_spi_blk[i] = 0;
    }
    for(i=0; i < BUF_SIZE; i++) {
            sio_rxbuf[i] = 0;
    }

// Initialize PSU data block - these aspects of data block set only once
    ds_get_4byte_id(PSUC_DS18S20, SILICON_ID);   // assign ID to PSU block
    *SOFTWARE_VERSION = software_version_byte;   // Software Version    byte


/*****************      Initialize Devices      ***************/
    //check for presence of DS18S20 temperature sensors
    temp1_present = ds_initialize(PSU_DS18S20);
    temp2_present = ds_initialize(DTEMP1_ID);
    temp3_present = ds_initialize(DTEMP2_ID);
}

/*****************************************************************************************
 *  Turn-On (Startup) Sequence         *
 ***************************/

void sequence_on (void)
{
//  wait_time( T100mS );
    nPSU_ON = 0;
    wait_time( T100mS );
    nCORE_ON = 0;
    LED_OUTON = 0;                                  // 0 = LED on
}

/*****************************************************************************************
 *  Turn-Off Sequence           *
 ******************************/
```

```
void sequence_off (void)
{
    nCORE_ON = 1;
    wait_time( T100mS );
    nPSU_ON = 1;
    wait_time( T100mS );
    LED_OUTON = 1;                                  // LED off
}

/*****************************************************************************************
 *  Reset MCE      *
 ******************/

void reset_MCE (void)
{
    BRST = 1;                                       // Pulse Reset Line for 100mS
    wait_time( T100mS );
    BRST = 0;
}

/*****************************************************************************************
 *  Cycle Power        *
 **********************/

void cycle_power (void)
{
    sequence_off();
    wait_time( T100mS );
    sequence_on();
}

/*****************************************************************************************
/*  Send PSU Data Block to CC via SPI      *
/*******************************************/
// Sends the 36 byte PSU Status Block to the CC via SPI interface while simultaneously
// receiving a command from the CC.  ACK/NAK byte near end of datablock indicates whether
// a valid command was received during the SAME datablock transmission.

void send_psu_data_block (void)
{
    // Begin Transaction
    spi_idx = 0;                                    // Start at beginning of data block
    CCSS = 0;                                       // Select Clock Card to listen on SPI bus


    // Send first 34 of 36 bytes
    //need to calculate checksum based on ACK/NAK byte after CC command recv'd)
    while(spi_idx < ACK_BYTE_POS) {
        SPDAT = ps_data_blk[spi_idx];               // send byte #spi_idx
        while(!spi_complete);                       // wait for end of byte transmission
        spi_complete = 0;                           // clear software flag
        spi_idx++;                                  // increment data block index
    }

    // Update ACK/NAK byte and send
    parse_command();                                // Check if command rcv'd, set ACK/NAK byt
e

    // Send ACK/NAK byte
    SPDAT = ps_data_blk[ACK_BYTE_POS];
    while(!spi_complete);                           // wait for end of byte transmission
    spi_complete = 0;                               // clear software flag
    // Update Checkbyte and send
    COMPLETE_CHECKSUM;                          // 2's compliment, so CHECKSUM_BYTE + all
other bytes = 0
    SPDAT = ps_data_blk[ACK_BYTE_POS + 1];      // Send Check byte
    while(!spi_complete);                       // wait for end of byte transmission
    spi_complete = 0;                           // clear software flag

    // Finish Transaction
    CCSS = 1;                                   // De-select Clock Card
}

/*****************************************************************************************
/*  Wait Timer - 5ms Multiples          */
/*****************************************/
//Sets up T1 interrupt to loops x 5mS, waits specified time then returns

void wait_time (unsigned char loops)
{
    timeup_T1 = CLEAR;
    TL1 = LS_RELOAD_5mS;                        // Interrupt interval set to 5mS
    TH1 = MS_RELOAD_5mS;
    num_T1_ints = loops;                        // time expires after 1 interrupt
    TR1 = ON;
    while ( timeup_T1 != SET );                 // wait here for specified time to expire
}

/*****************************************************************************************
/*  Microsecond Wait Timer          */
/*************************************/
// returns 2*time_us_div2 + 3 (in uS)....tested and verified
// therefore works for a minimum of 3us (time_us_div2 = 0) or maximum of 513us (time_us_di
v2 = 0xFF)
// from numbers below, delay = time_us_div2 * (1.25+ 0.25 + 0.5) + 1.25+ 0.25 + 1 + 0.5 =
2*time_us_div2 + 3 (in uS)

void wait_time_x2us_plus3 (unsigned char time_us_div2)      // 1.25 us to call function
{
    while(time_us_div2>0) {                     // each comparison takes 1.25 uS
        time_us_div2--;                         // 250ns operation
    }                                           // 500ns delay to begining of loop
    _nop_();                                    // 250 ns delay so total is an integer
}                                               // 500 ns to return from function

/****************************************************************************************/
/* Timer0 Service Routine               */
/****************************************/
// Interrupt occurs every 32ms when enabled   -  used for LED blink and polling data

void timer0_isr (void) interrupt 1 using 3
{
    ++bcnt;
    if ( bcnt == BRATE320mS) {
        bcnt = 0;
        poll_data = SET;                        // poll data every 320ms
        if (blink_en == SET)
            LED_FAULT = ~LED_FAULT;             // toggle LED every 320ms if enabled
   }
}

/****************************************************************************************/
/* Timer1 Service Routine               */
/****************************************/
// Interrupt occurs every 5ms when enabled   - used for wait_time()
```

```
void timer1_isr (void) interrupt 3 using 3
{
   --num_T1_ints;                                     // count the number of interupts
   if (num_T1_ints == 0) {                            // check if interrupt time is up
      TR1=OFF;                                        // Stop the timer
      timeup_T1 = SET;                                // Indicate time is up
   }
   else {                                             // reload timer
      TL1 = LS_RELOAD_5mS;                            // interrupts always occur every 5mS
      TH1 = MS_RELOAD_5mS;
   }
}

/*************************************************************************************/
/* Send Serial Message      */
/***************************/

void snd_msg (char *message)
{
   msg_ptr = message;
   TI = SET;                                          // Generates SIO interrupt
}

/*************************************************************************************/
/* Serial Interrupt Service Routine     */
/***************************************/
// Interrupt driven serial I/O

void serial_isr (void) interrupt 4 using 2
{
    char msg;

    // Transmitted Data Interrupt
    if ( TI == SET ) {
        TI = CLEAR;                                   // Clears TI Interrupt
        msg = *msg_ptr;
        if (msg != NULL) {                            // If msg not NULL, load into trans. buffe
r
            ++msg_ptr;
            SBUF = msg;
        }
        else msg_ptr = 0;
    }

    // Received Data Interrupt
    if ( RI == SET ) {
        RI = CLEAR;                                   // Clears RI Interrupt
        msg = SBUF;
        sio_rxbuf[sio_rx_idx++] = msg;
        if (sio_rx_idx >= (BUF_SIZE-1))
            --sio_rx_idx;
        if (msg == LF) {                              // LineFeed indicates end of message
            sio_rx_idx = 0;
            sio_msg_complete = SET;                   // Indicate entire message received
        }
    }
}

/*************************************************************************************/
/* SPI Interrupt Service Routine     */
/************************************/
// read and clear spi status register
```

```
void spi_isr (void) interrupt 9
{
    switch( SPSTA )
    {
        // SPIF flag set --> transmission complete
        case 0x80:
            rcv_spi_blk[spi_idx] = SPDAT;       // read receive data
            spi_complete = 1;                   // indicate transaction finished
            break;

        /* error cases -> refer to pg. 96 in AT89 datasheet */
        // mode fault
        case 0x10:
            // this does not apply as single master on SPI bus and SSDisable bit set in SP
STA register

            break;

        // write collision
        case 0x40:
            // write collision does NOT cause an interrupt therefore this should be elsewh
ere if needed
            // currently ONLY the function send_psu_data_block() ever writes to SPDAT so w
rite collision not possible
            break;

        default:
            break;
    }
}

/************************************************************************************* */
/* Retrieve Data Block       */
/*****************************/
//Updates PSU Data Block with Current Values

void update_data_block (void)
{
    // Fan Speeds
    // get_fan_speeds();                                           // not implemented

    // DS18S20 - Temperatures - read only if present
    if (temp1_present){
        ds_get_temperature(PSU_DS18S20, PSU_TEMP_1);               // temperature 1
    }
    else
        ds_get_temperature(PSUC_DS18S20, PSU_TEMP_1);

    if (temp2_present)
        ds_get_temperature(DTEMP1_ID, PSU_TEMP_2);                 // temperature 2
    if (temp3_present)
        ds_get_temperature(DTEMP2_ID, PSU_TEMP_3);                 // temperature 3

    /*** ADC - Voltage and Current Readings - refer to documentation ***/
    // Ground reading scaled to 2mV per division (+/- 2.047V range)
    read_adc(ADC_CH5, ADC_BI_5V, VOLTAGE, ADC_OFFSET);             // Grounded input reading
(bipolar)

    // Voltages scaled to ~61% of nominal values, unipolar
    read_adc(ADC_CH0, ADC_UNI_10V, VOLTAGE, V_VCORE);              // +Vcore supply scaled
    read_adc(ADC_CH1, ADC_UNI_10V, VOLTAGE, V_VLVD);               // +Vlvd supply scaled
    read_adc(ADC_CH2, ADC_UNI_10V, VOLTAGE, V_VAH);                // +Vah supply scaled
    read_adc(ADC_CH3, ADC_UNI_10V, VOLTAGE, V_VA_PLUS);         // +Va supply scaled
    read_adc(ADC_CH4, ADC_UNI_10V, VOLTAGE, V_VA_MINUS);        // -Va supply scaled

    // Currents scaled to ~73% of nominal values, unipolar
    read_adc(ADC_CH0, ADC_UNI_10V, CURRENT, I_VCORE);           // Current +Vcore supply
    read_adc(ADC_CH1, ADC_UNI_10V, CURRENT, I_VLVD);            // Current +Vlvd supply
    read_adc(ADC_CH2, ADC_UNI_10V, CURRENT, I_VAH);             // Current +Vah supply
    read_adc(ADC_CH3, ADC_UNI_10V, CURRENT, I_VA_PLUS);         // Current +Va supply
    read_adc(ADC_CH4, ADC_UNI_10V, CURRENT, I_VA_MINUS);        // Current -Va supply

    // release SCLK
    SCLK = 1;                                           //**needed for SPI in send_psu_data_block
to work**


    // Bookkeeping
    // Status Word currently not used (initialized to 0)
    // *STATUS_WORD = 0;                                        // undefined status word -
 higher byte
    // *(STATUS_WORD+1) = 0;                                    // undefined status word -
 lower byte
    *ACK_NAK = 0;                                               // Clear any ACK/NAK


    // Check Digit pre-Calculation
    check_digit();                                              // updates running checksu
m total - done here for quick response in send_data_block()
}

/*************************************************************************************/
/* Generate Check Digit    */
/***************************/
// Implemented as checksum for now to optimize calculation speed (tradeoff for sub-optimal
 error detection)
// Checksum byte totals 0 when summed with the other 35 bytes in the PSU data block  (igno
ring addition overflow)
// *** This function calculated total of first 34 bytes in checksum
// *** Finish checksum calculation and set in data block using COMPLETE_CHECKSUM macro (**
AFTER** ACK/NAK byte has been set)

void check_digit (void)
{
    int j;
    running_checksum = 0;                                       // reset checksum
    for(j = 0; j < ACK_BYTE_POS; j++) {                         // sum data block up to AC
K byte
        running_checksum += ps_data_blk[j];
    }
}

/*************************************************************************************/
/* Parse Command Received from CC    */
/*************************************/
// could to make this more robust - varying degrees of complexity in how to implement this
// current protocol receives 3 2-byte command in first 6 bytes of PSU Data Block transacti
on

void parse_command(void)
{
    //assume commands are in first 6 bytes of rcv'd SPI block, ordered and repeated thrice
    if ( commands_match(rcv_spi_blk, rcv_spi_blk+2,rcv_spi_blk+4) && command_valid(rcv_spi
_blk) ) {
        cc_command = rcv_spi_blk;
        *ACK_NAK = ACK;
    }

    // ACK command iff valid command received in triplicate
    else {
        cc_command = NULL;                                  // else NAK command
        *ACK_NAK = NAK;
    }
}

/*****************************************************************************************/
/* Matching Commands Check    */
/*****************************/
// returns true if three matching commands sent else false

bit commands_match (char *com_ptr_1, char *com_ptr_2, char *com_ptr_3)
{
    // first two commands match
    if( (*com_ptr_1 == *com_ptr_2) && (*(com_ptr_1 + 1) == *(com_ptr_2 + 1)) ) {

        // third command matches
        if( (*com_ptr_1 == *com_ptr_3) && (*(com_ptr_1 + 1) == *(com_ptr_3 + 1)) )
            return TRUE;
        else
            return FALSE;
    }

    else
        return FALSE;
}

/*****************************************************************************************/
/* Valid Command Check    */
/***************************************/
// returns true if command received is valid

bit command_valid (char *com_ptr)
{
    // If command is valid return TRUE
    if( (*com_ptr == 0) && (*(com_ptr+1) == 0) )                // Status Request (default
)
        return TRUE;
    else if( (*com_ptr == 'C') && (*(com_ptr+1) == 'P') )       // Cycle Power Command
        return TRUE;
    else if( (*com_ptr == 'R') && (*(com_ptr+1) == 'M') )       // Reset MCE Command
        return TRUE;
    else if( (*com_ptr == 'T') && (*(com_ptr+1) == 'O') )       // Turn Off Command
        return TRUE;
    else
        return FALSE;
}
```

```
/*************************************************************************************/
/*      I/O Assignments             */
/**********************************/
// Revision history:
// $Log: io.h,v $
// Revision 1.2  2006/08/30 19:54:19  stuartah
// Implemented checksum
//
// Revision 1.1  2006/08/29 21:06:06  stuartah
// Initial CVS Build - Most Basic Functionality Implemented
//
/*************************************************************************************/

// AT89 I/O Pin Assignments
sbit BUS_SP2 =      P0^5;       // Bus Spare 1
sbit BUS_SP1 =      P0^6;       // Bus Spare 2

sbit FAN2_SPD =     P1^0;       // Fan 2 Tacho - Input
sbit CS_IADC =      P1^1;       // Chip Select Current ADC - Output
sbit SER_DSR =      P1^2;       // RS-232 Data Set Ready - Input
sbit SREQ =         P1^3;       // Clock Card Service Request - Input
sbit CCSS =         P1^4;       // Active for PS Data on SPI - Output
sbit MISO =         P1^5;       // SPI MISO - Input
sbit SCLK =         P1^6;       // SPI CLK - Output
sbit MOSI =         P1^7;       // SPI MOSI - Output

sbit BRST =         P2^0;       // Subrack Reset - Output
sbit nPSU_ON =      P2^1;       // Turn On PSU - Output
sbit PSUC_ID =      P2^2;       // Dallas DS18S20 PSUC ID - Input
sbit DTEMP2 =       P2^3;       // Dallas DS18S20 PSU Digital Temp2 - Input
sbit DTEMP1 =       P2^4;       // Dallas DS18S20 PSU Digital Temp1 - Input
sbit PSU_ID =       P2^5;       // Dallas DS18S20 PSU ID - Input
sbit nCORE_ON =     P2^7;       // Core Voltage On - Output

sbit CS_VADC =      P3^2;       // Chip Select Voltage ADC - Output
sbit CS_EEPROM =    P3^3;       // Chip Select EEPROM Atmel AT25128A - Output
sbit FAN1_SPD =     P3^4;       // Fan 1 Tacho - Input
sbit LED_FAULT =    P3^5;       // LED1 - Output    0 = off 1 = on
sbit LED_OUTON =    P3^6;       // LED3 - Output    0 = on  1 = off
sbit LED_STATUS =   P3^7;       // LED2 - Output    0 = on  1 = off

sbit SPARE2 =       P4^0;       // Bus Spare 1
sbit SPARE1 =       P4^1;       // Bus Spare 2


// PSU Data Block Settings
#define CC_SPI_BLEN     36      // Bytes in SPI Block to Clock Card
#define ACK_BYTE_POS    34      // ACK/NAK byte position - used instead of (CC_SPI_BLEN -
2) for optimization


// I/O Pin Bit Masks - For DS18S20 Addressing
#define PSUC_DS18S20    0x04
#define DTEMP2_ID       0x08
#define DTEMP1_ID       0x10
#define PSU_DS18S20     0x20


// SPI Interface
#define ADC_SPI_BLEN    1               // Bytes in SPI Block to ADC
#define SPI_MSTR        0x10            // SPCON Bit Set for Master
#define SPI_CPOL0       ~0x08           // SPCON Bit Set for Clock Polarity - Active low
#define SPI_CPHA0       ~0x04           // SPCON Bit Set for Clock Phase - Active low
```

```
#define SPI_1M5Hz       0x03            // SPCON Bits D7,D1,D0 for 1.5MHz
#define SPI_6MHz        0x01            // SPCON Bits D7,D1,D0 for 6MHz
#define SPI_EN          0x40            // SPCON Bit D6 Enables SPI
#define SPI_SSDIS       0x20            // SPCON Bit 5 Set disables SS Interrupts


// General Keywords
#define ON          1
#define OFF         0
#define TRUE        1
#define FALSE       0
#define SET         1
#define CLEAR       0
#define ENABLE      0
#define DISABLE     1
#define VOID        0x0
#define CR          0x0d
#define LF          0x0a
#define ACK         0x06
#define NAK         0x15
#ifndef NULL
    #define NULL    0x00                // NULL usually defined
#endif


// Timing Parameters
#define MS_RELOAD_5mS   216             // timing confirmed with 24MHz Clock
#define LS_RELOAD_5mS   239             // Timing register loaded with 0xFFFF - (216)(239)
 = 0xD8FF = 10000, implies 500ns delay per click
#define T5mS            1
#define T15mS           3
#define T25mS           5
#define T100mS          20
#define BRATE320mS      10


// ADC Control Channel/Mode Select
#define ADC_CH0     0x80
#define ADC_CH1     0x90
#define ADC_CH2     0xA0
#define ADC_CH3     0xB0
#define ADC_CH4     0xC0
#define ADC_CH5     0xD0
#define ADC_CH6     0xE0
#define ADC_CH7     0xF0

#define ADC_UNI_5V  0x1
#define ADC_BI_5V   0x5
#define ADC_UNI_10V 0x9                 // default mode used
#define ADC_BI_10V  0xd

#define VOLTAGE     0
#define CURRENT     1
```

```
/*****************************************************************************************
*******/
/*      ADC Interface - Maxim MAX1271           */
/*********************************************************
MAX1271 ADC Interfacing Function
    Manually implements SPI transaction (lack of SSTRB signal means data must be clocked m
anually)
    Currently SCLK almost uniform...timing verified as acceptable
    Timing verified for extreme case of switching consecutively between +/- maximum input
levels
*****************************************************************************************
*******/

unsigned char bdata adc_data;                          // bit adressable variable
sbit ADC_MS_DBIT = adc_data^7;


/****************************************************************************************
 *  Read ADC    *
 ************** */
// non-pipeling implementation

void read_adc(char chan, char mode, bit adc_sel, char *target)
{
    unsigned char bit_cnt, *temp_char_ptr;
    unsigned int adc_reading=0;

    MISO = 1;                                          // port bit set for input

    //SPI must be disabled to manually control SCLK
    SPCON &= ~SPI_EN;                                  // SPEN = 0:

    adc_data = chan + mode;                            // higher 4 bits determine channel, lo
wer 4 bits determine mode
    SCLK = 0;                                          // make sure CLK is low
    _nop_();                                           // delay for hardware
    _nop_();

    // select ONE ADC only - done with adc_sel bit as sbit/sfr types cannot be passed into
 functions
    if (adc_sel == VOLTAGE)
        CS_VADC = 0;
    else
        CS_IADC = 0;

    // Send Control Byte - shift out 8 bits (8 clock cycles)
    for (bit_cnt=1 ; bit_cnt <=8; bit_cnt++) {
        MOSI = ADC_MS_DBIT;                            // starts conversion, data clocked in
to ADC on rising clock edge
        SCLK = 1;                                      // loads data bit
        adc_data = adc_data<<1;
        SCLK = 0;
    }

    MOSI = 0;                                          // don't start new conversion

    /***    Wait For Ready  ----  need to change, no SSTRB pin connection **/
    // while ( ADC_STRB == LOW );
    // Need 5 clock cycle delay in place of waiting for SSTRB signal to assert.
    // ADC starts shifting out data on 14th clock signal.
    for (bit_cnt=1 ; bit_cnt <=5; bit_cnt++) {
        SCLK = 1;
        _nop_();
        //_nop_();                                     // include for uniform timing
        SCLK = 0;
        _nop_();
        //_nop_();                                     // include for uniform timing
    }

    /***     now clock in 12 data bits  ***/
    // get first bit
    SCLK = 1;
    if ( MISO == 1) {                                  // MSB is ready at DOUT
        ++adc_reading;
     }
    SCLK = 0;                                          // this edge latches bit

    // get last 11 bits
    for ( bit_cnt=1 ; bit_cnt<=11 ; bit_cnt++ ) {
        adc_reading = adc_reading<<1;                   // rotate reading
        SCLK = 1;
        if ( MISO == 1) {
            ++adc_reading;
        }
        //else _nop_();                                // include for uniform timing
        SCLK = 0;                                      // loads next bit
   }

    // de-select ADC
    if (adc_sel == VOLTAGE)
        CS_VADC = 1;
    else
        CS_IADC = 1;

    // clear ports
    MISO = 1;
    MOSI=1;

    // re-enable SPI
    SPCON |= SPI_EN;

    // return adc_reading;
    temp_char_ptr = &adc_reading;                      // need CHAR ptr to access individual
bytes of int adc_reading
    *target = *temp_char_ptr;                          // higher order byte
    *(target+1) = *(temp_char_ptr+1);                  // lower order byte
}
```

```
/*************************************************************************************/
/*       Silicon Serial Number / Temperature Sensor Functions - DS18S20          */
/********************************************************************************/


/*****  Refer to DS18S20 Datasheet for Command and Timing Specs *****/
/*  The transaction sequence for accessing the DS18S20 is as follows:
Step 1. Initialization (reset pulse)
Step 2. ROM Command (followed by any required data exchange)
Step 3. DS18S20 Function Command (followed by any required data exchange)
Step 4. Read returned bytes                                                   */


/*********  Function Prototypes  **************/
// External Functions/Variables (defined in scuba2ps.c)
extern void wait_time_x2us_plus3(unsigned char);                        // Waits (2*Value + 3)
 microseconds

// 'Public' Functions - ONLY these functions should be called externally
bit ds_initialize( char );                                              // Initializes DS18S20
void ds_get_4byte_id( char, char* target);                              // Reads Silicon ID, s
ets target value
void ds_get_temperature( char, char* target);                           // Reads temperature f
rom DS memory, sets target value

// The following functions are declared as 'static' to make them 'private'
// Command Functions
static void ds_convert_T( void );                                       // Starts temperature
conversion
static bit ds_reset(void);                                              // Command Reset

// 1-Wire Bus Protocol I/O Functions
static void ds_write_byte(unsigned char);                               // Writes Byte
static unsigned char ds_read_byte(void);                                // Reads Byte
static void ds_write_bit(bit);                                          // Writes Bit
static bit ds_read_bit(void);                                           // Reads Bit
static bit read_bus(void);                                              // Physical bus line b
it read


/**************  DS18S20 Comands  *****************/
#define READ_ROM            0x33                                        // Note: READROM comma
nd only works with a single device on the bus   (as in current PSUC design)
#define SKIP_ROM            0xCC
#define CONVERT_T           0x44
#define READ_SCRATCHPAD     0xBE


/*************  DS18S20 Timing Parameters  *************/
// timing as per DS18S20 datasheet (DS18S20.pdf) and "1-Wire Communication Through Softwar
e"
// (http://www.maxim-ic.com/appnotes.cfm/appnote_number/126 or 1WireCom.pdf)
// timing are RECOMMENDED times and can be adjusted for timing optimization...see above do
cs
#define WAIT_TIME_1uS       _nop_(); _nop_(); _nop_(); _nop_();                 // 1 uS
#define WAIT_TIME_A         wait_time_x2us_plus3(1); WAIT_TIME_1uS;             // 6 uS
#define WAIT_TIME_B         wait_time_x2us_plus3(30); WAIT_TIME_1uS;            // 64 uS
#define WAIT_TIME_C         wait_time_x2us_plus3(28); WAIT_TIME_1uS             // 60 uS
#define WAIT_TIME_D         wait_time_x2us_plus3(3); WAIT_TIME_1uS;             // 10 uS
#define WAIT_TIME_E         wait_time_x2us_plus3(3);                            // 9 uS
#define WAIT_TIME_F         wait_time_x2us_plus3(26);                           // 55 uS
//#define WAIT_TIME_G                   0                                       // 0 uS -
not needed
```

```
#define WAIT_TIME_H         wait_time_x2us_plus3(238); WAIT_TIME_1uS;           // 480 uS
#define WAIT_TIME_I         wait_time_x2us_plus3(33); WAIT_TIME_1uS;            // 70 uS
#define WAIT_TIME_J         wait_time_x2us_plus3(203); WAIT_TIME_1uS;           // 410 uS


/***************** Source Code **********************/
//#include DS18S20.c                                                            // source
file
```

```
/*************************************************************************************/
/*       Silicon Serial Number / Temperature Sensor Functions - DS18S20         */
/******************************************************************************/

/*****  Refer to DS18S20 Datasheet for Command and Timing Specs *****/
/*  The transaction sequence for accessing the DS18S20 is as follows:   (Refer to datashee
ts)
Step 1. Initialization (reset pulse)
Step 2. ROM Command (followed by any required data exchange)
Step 3. DS18S20 Function Command (followed by any required data exchange)
Step 4. Read returned bytes                                                   */

// header file - contains function protypes and operational parameters
#include "DS18S20.h"

// variables
unsigned char bdata command_bit_adr;                // temporary bit-adressable variable f
or reading/writing at bit level
sbit command_lsb = command_bit_adr^0;

unsigned char adr_mask;                             // sbits CANNOT be passed between func
tions...therefore bit mask used to adress P2 line (all Ds18S20s connceted tp P2)
                                                    // value MUST be one of 0x04 - PSUC_ID
, 0x08 - DTEMP2, 0x10 - DTEMP1, 0x20-PSU_ID


// Physical Bit Writing - Used to support multiple DS18S20s on different busses - Bus defa
ult state is HIGH
#define DRIVE_BUS_LOW       P2 = P2 & ~adr_mask;    // Write a 0
#define RELEASE_BUS         P2 = P2 | adr_mask;     // Write a 1 (bus default high)

/*************************************************************************************/
/*  Initialize DS18S20      */
/***************************/
bit ds_initialize( char mask )
{
    bit present = 0;

    // Initialize
    adr_mask = mask;                                // select deviec
    present = ds_reset();

    //  Send CONVERT T command if device prsent
    if(present)
        ds_convert_T();                             // initial convert takes about a secon
d to return accurate readings

    return present;
}

/*************************************************************************************/
/*  GET Silicon ID         */
/***************************/
// ROM Code Format  [ 8bit CRC | 48bit Serial Number | 8 bit Family Code =0x10 ]
// Sent LSB first with bits sent LSb first

void ds_get_4byte_id( char mask, char *target )     // returns pointer to lowest 32 bits o
f 48 bit Serial Number
{
    bit presence_detect = 0;                        // for detecting presence pulse on res
et

    unsigned char family_code;                      // for storing returned bytes
    unsigned char serial_number[6];
    unsigned char crc_code;

    // Initialize
    adr_mask = mask;                                // select device
    presence_detect = ds_reset();                   // for now ignore presence pulse (assu
me always detected)

    // Send ROM command
    ds_write_byte(READ_ROM);                        // this command skips the Function Com
mand step

    // Receive back 8 bytes
    family_code = ds_read_byte();
    serial_number[5] = ds_read_byte();              // read back lower order bytes first
    serial_number[4] = ds_read_byte();
    serial_number[3] = ds_read_byte();
    serial_number[2] = ds_read_byte();              // currently store all 6 bytes of seri
al code.  Could ignore unneeded bytes and set psu block directly here
    serial_number[1] = ds_read_byte();
    serial_number[0] = ds_read_byte();
    crc_code = ds_read_byte();                      // ignore CRC code as only one device
on bus.  May implement error check later.

    // Set pointer to lowest 4 bytes
    *target = serial_number[2];                     // ignore highest 2 bytes of Silicon I
D
    *(target+1) = serial_number[3];
    *(target+2) = serial_number[4];
    *(target+3) = serial_number[5];
}

/*****************************************************************************************/
/* GET Temperature       */
/************************/
// DS18S20 returns 2 byte signed temperature 0.5 deg. Celsius per bit (refer to datasheet)
// Function returns 1 byte temperature (signed byte, 1 deg. Celsius per bit)

void ds_get_temperature( char mask, char *target )
{
    unsigned char value, sign;                      // stores value and sign info
    adr_mask = mask;                                // select device

    /*  Send CONVERT T command */
    ds_convert_T();

    /*  Send READ SCRATCHPAD command */
    // Initialize
    ds_reset();                                     // for now ignore presence pulse (assu
me always detected)

    // Send ROM Command                             // send SKIP ROM command
    ds_write_byte(SKIP_ROM);                        //one device on bus only so don't need
 to adress

    // Send Function Command
    ds_write_byte(READ_SCRATCHPAD);                 // send READ SCRATCHPAD command

    /*  Read Temperature Data */
    // Read back scratchpad
    value = ds_read_byte();                         // this byte contains only magnitude i
nformation ( 4 byte 2's compliment form)
    sign = ds_read_byte();                          // this byte contains only sign inform
```

```
ation ( 4 byte 2's compliment form) ( = 00000000 or 11111111)

    // Issue reset
    ds_reset();                                     // this terminates scratchpad reading
(no need to read further bytes...ignore CRC check byte for now)

    // Scale to single byte and return              //***Note: this truncates the 0.5 degr
ee least significant digit ... ie 25.5 and 25 both become 25 (floor function)
    value >>= 1;                                    // divide value by 2 (scale from 0.5 d
eg C to 1 deg. C per bit)

    if (sign > 0)                                   // *** need to make this more robust t
o allow for errors in sign byte
        value |= 0x80;                              // set MSB to indicate 2's compliment
(0 shifted in to MSB in above line)

    /*  'Return' Scaled Temperature  */
    *target = value;
}

/*****************************************************************************************/
/*  Initiate Temperature Conversion    */
/****************************************/

static void ds_convert_T (void)
{
   /*   Send CONVERT T command */
    // Initialize
    ds_reset();                                     // ignore presence pulse (assume alway
s detected)

    // Send ROM Command
    ds_write_byte(SKIP_ROM);

    // send Function Command
    ds_write_byte(CONVERT_T);
    while( ds_read_bit() );                         // wait for DS to return a 1 which ind
icates temperature conversion complete
}

/*****************************************************************************************/
/*  1-Wire Bus Reset Pulse    */
/*******************************/
// Generates a 1-wire reset pulse and returns 1 iff presence pulse detected

static bit ds_reset(void)
{
    bit presence = 0;

    //Initial Delay
    //WAIT_TIME_G;                                  // 0 uS

    //reset pulse
    DRIVE_BUS_LOW;                                  // drive bus low
    WAIT_TIME_H;                                    // hold low to indicate reset
    RELEASE_BUS;                                    // release bus

    //detect presence pulse
    WAIT_TIME_I;                                    // wait for presence pulse
    presence = ~read_bus();                         // sample for presence pulse, indicate
d by bus being pulled LOW
    WAIT_TIME_J;                                    // reset sequence recovery time
    return presence;                                // return presence indicator
```

```
}

/*****************************************************************************************/
/*  1-Wire Bus Protocol - Write Byte    */
/****************************************/

static void ds_write_byte(unsigned char command)    // ***sent LSB first
{
    int a;
    command_bit_adr = command;                      // load command byte into bit-adressab
le variable

    for ( a = 0; a < 8; a++ ) {                     // Write single bit at a time, LSB to
MSB
        ds_write_bit(command_lsb);
        command_bit_adr >>=1;                       // right-shift (bit 1 -> LSB)
    }
}

/*****************************************************************************************/
/*  1-Wire Bus Protocol - Read Byte    */
/***************************************/

static unsigned char ds_read_byte(void)             // ***read LSB first
{
    int b;
    unsigned char read_temp = 0;

    for ( b = 0; b < 8; b++ ) {                     // Read single bit at a time, LSB to M
SB
        read_temp >>= 1;                            // right shift data byte
        if( ds_read_bit() )                         // read bit
            read_temp |= 0x80;                      // if '1' then set MSb of read_temp; e
lse do nothing
    }

    return read_temp;                               // return byte (normal MSB first forma
t)
}

/*****************************************************************************************/
/*  1-Wire Bus Protocol - Write Bit    */
/***************************************/

static void ds_write_bit(bit com_bit)
{
    // write a 1
    if (com_bit) {
        DRIVE_BUS_LOW;                          // drive bus low to initiate write time sl
ot
        WAIT_TIME_A;                            // hold line low
        RELEASE_BUS;                            // release bus
        WAIT_TIME_B;                            // hold bus high for write time slot and a
llow recovery time
    }

    // write a 0
    else {
        DRIVE_BUS_LOW;                          // drive bus low to initiate write time sl
ot
        WAIT_TIME_C;                            // hold bus low over slot
        RELEASE_BUS;                            // release bus
        WAIT_TIME_D;                            // recovery time
    }
}

/*****************************************************************************************/
/*  1-Wire Bus Protocol - Read Bit    */
/**************************************/

static bit ds_read_bit(void)                        // read ONLY works after master has writte
n a READ-type command
{
    bit temp_bit;

    DRIVE_BUS_LOW;                                  // drive bus low to initiate read time slo
t
    WAIT_TIME_A;                                    // hold line low
    RELEASE_BUS;                                    // release bus
    WAIT_TIME_E;                                    // allow settling time
    temp_bit = read_bus();                          // read bit
    WAIT_TIME_F;                                    // recovery time
    return temp_bit;                                // return read bit value
}

/*****************************************************************************************/
/*  Physical Bit Read    */
/*************************/
//reads bit from input specified by adr_mask

static bit read_bus(void)
{
    if( (P2 & adr_mask) == 0 )
        return 0;
    else                                            // P2 & adr_mask = adr_mask
        return 1;
}
```

```c
/*************************************************************************************/
/*        AT25128A SPI EEPROM Interface           */
/**********************************************************************************/

/*** Note: This code is incomplete *****/

// Instructions
#define WREN    0x06            // WRITE ENABLE - device powers up in write disable state,
 so all commands must be predeeded by this one
#define WRDI    0x04            // WRITE DISABLE
#define RDSR    0x05            // READ STATUS REGISTER
#define WRSR    0x01            // WRITE STATUS REGISTER
#define READ    0x03            // READ COMMAND
#define WRITE   0x02            // WRITE COMMAND


// variables
unsigned char bdata status;                 // bit-adressable variable for reading STATUS
register
sbit nRDY = status^0;



/*************************************************************************************/
/*  Initialize AT25128A   */
/***************************/
void rom_initialize(unsigned char mask )
{
    rom_write(WREN);                // enable programming instructions

}

/*************************************************************************************/
/*  Send Command Byte AT25128A   */
/****************************/
// ***** This function presumes CS_EEPROM already selected *****
void rom_send_comand(unsigned char instr )
{
    SPDAT = instr;                      // send byte
    while(!spi_complete);               // wait for end of transmission
}


/*************************************************************************************/
/*  Write AT25128A   */
/****************************/
void rom_write_data( unsigned char instr, unsigned char address, unsigned char wdata )
{
    // Select EEPROM
    CS_EEPROM = 0;

    // Enable WRITE
    rom_send_command(WREN);

    // Send address and data to write
    rom_send_command(WRITE);
    rom_send_command(address);
    rom_send_command(wdata);

    // Programming occurs after EEPROM de-selected
    CS_EEPROM = 1;

    // Wait for WRITE to finish
    rom_read(RDSR);                              // check STATUS register
    while(nRDY != 0);                            // wait for EEPROM to become READY

}

/************************************************************************************/
/*  Read AT25128A   */
/****************************/
void rom_read(unsigned char address, unsigned char numbytes )
{
    // Select EEPROM
    CS_EEPROM = 0;

    rom_send_command(READ);
    rom_send_command(address);

    // EEPROM will automatically increment address and keep returning bytes until CS retur
ns high
    while(numbytes > 0)
    {
        read byte
    }

    // De-select EEPROM
    CS_EEPROM = 1;

}
```

4.0 CONCLUSION

The objectives of the work described in this report were successfully achieved. Firmware was developed for the SCUBA2 MCE Power Supply Controller Card which has been demonstrated to monitor power supply behavior, communicate with the clock card, and act on received commands. This firmware conforms to the initial specifications it set out to meet, and has been verified through hardware testing. The voltage and current measurements of the PSU were determined to be within a 2.5% deviation from the expected values. This is less than the expected deviation and very much acceptable for voltage/current monitoring. The next step is to implement the last few extra features and prepare the firmware for ‘release.’

The development of PSUC firmware was a long and arduous effort which ultimately proved quite rewarding. This report summarized the many successes of the PSUC firmware development, but did not touch on the countless bugs, roadblocks, and pitfalls the author encountered along the way. Nor does this report address the many other contributions the author has made to SCUBA2 over the last six months. Firmware development forces the designer to understand every aspect of the device at hand. In this sense it lies at the crossroads between computer engineering and traditional electronics.

SCUBA2 is set be delivered in 2007 and is set to immediately contribute to cutting edge cosmological research. Clearly, the Multi-Channel Electronics are a key component of this. I look forward to future results produced by SCUBA2, and in this regard I am fortunate to have contributed to this project.

5.0 APPENDIX

The following three documents are internal SCUBA2 files provided for reference.

5.1 SPI COMMUNICATIONS INTERFACE BETWEEN CC AND PS

January 31, 2006 (TF)
September 7, 2006 (SH)

The CC and the PS Card communicate via an SPI interface. This interface consists of the following signals:

MOSI – Master Output - Serial output data from the PS Card
MISO – Master Input - Serial input to the PS Card
SCLK – Serial Clock - generated by PS Card
SREQ – Service Request - generated by the CC
CCSS – Clock Card Slave Select - asserted by PS before SCLK for CC data transfers

All transfers are initiated by the CC via the Service Request Signal, SREQ. The PS card provides the SPI clock so it is the SPI "Master". This configuration was chosen as the controller on the PS card already communicates to other devices on the PS card as an SPI Master. The PS Card always has a block of data, as defined in Table 1 below, ready to transmit to the CC. To retrieve the data block the CC asserts SREQ. The PS responds with a 1.5MHz clock burst of 288 cycles (192uS transfer). Data clocked out of the PS Card is read on the rising edge of SCLK. As data is read by the CC on the MOSI line a command is being sent on the MISO line.

Power Command Structure

Commands are either the Request Status command which occurs at a periodic rate, a Subrack Reset or Power Command. Commands from the CC are 2 bytes long. Commands are clocked into the PS card as the data is clocked out. As each data block is 36 bytes and each command is 2 bytes a command may be repeated up to 18 times. To ensure command integrity each command must be sent at least 3 consecutive times. The power supply card checks that at least 3 consecutive commands match then acts upon the command. If all commands match the ACK character (0x60 is inserted in the data buffer otherwise the NAK character is inserted. The common periodic Request Status command is signaled by holding the MISO pin low for the whole 256 cycle clock burst. This is essentially the default command. Other commands use 2 unique ASCII characters for each command. The following commands are defined:

Cycle Power Command

Byte 1 (ASCII ‘C”)

D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	0	0	0	1	1

Byte 2 (ASCII ‘P’)

D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	1	0	0	0	0

Reset MCE Command

Byte 1 (ASCII ‘R”)

D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	1	0	0	1	0

Byte 2 (ASCII ‘M’)

D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	0	1	1	0	1

Turn Off Command

Byte 1 (ASCII ‘T”)

D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	1	0	1	0	0

Byte 2 (ASCII ‘O’)

D7	D6	D5	D4	D3	D2	D1	D0
0	1	0	0	1	1	1	1

The SPI clock rate is set to 1.5MHz, but may change. At this rate the Power Supply should be able to determine if a command is valid in time to update the ACK/NAK byte. The check digit for data with ACK and data with NAK will have to be calculated in advance with the correct checksum inserted once the command’s validity is determined.

Power Supply Data Block

Byte #s	Item	Bytes	Description
0	Silicon ID	4	32 least sig bits of 48 bit ID
4	Software Version	1	Encoded as hex byte. 0xYZ = version Y.Z
5	Fan1 Tachometer	1	RPM / 32
6	Fan2 Tachometer	1	RPM / 32

7	PSU Temperature 1	1	8 bit two’s comp. (1 deg. increments)
8	PSU Temperature 2	1	8 bit two’s comp. (1 deg. increments)
9	PSU Temperature 3	1	8 bit two’s comp. (1 deg. increments)
10	ADC Offset	2	Digitized Analog Ground
12	Supply Voltage 1	2	+Vcore Supply
14	Supply Voltage 2	2	+Vlvd Supply
16	Supply Voltage 3	2	+Vah Supply
18	Supply Voltage 4	2	+Va Supply
20	Supply Voltage 5	2	-Va Supply
22	Supply Current 1	2	Current +Vcore Supply
24	Supply Current 2	2	Current +Vlvd Supply
26	Supply Current 3	2	Current +Vah Supply
28	Supply Current 4	2	Current +Va Supply
30	Supply Current 5	2	Current -Va Supply
32	Status Word	2	For future expansion
34	ACK/NAK	1	ACK if command correct/NAK otherwise
35	Check Digit	1	2’s compliment of sum of all other bytes
	Total	36	36 x 8 = 288 clocks on the SPI Interface

Status Word Definition
No bits defined in current implementation:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0

D15	D14	D13	D12	D11	D10	D9	D8
0	0	0	0	0	0	0	0

Check Digit
Many methods are possible. Trade-off between computational complexity (and speed) vs. error detection ability. Current implementation is a checksum to for minimal processor overhead:

```
Sum  = 0                                (1 byte character)
For each byte in Data Block
        Sum = Sum + byte                (Ignoring overflow)
Checkbyte = ~Sum +1                     (2’s compliment)
```

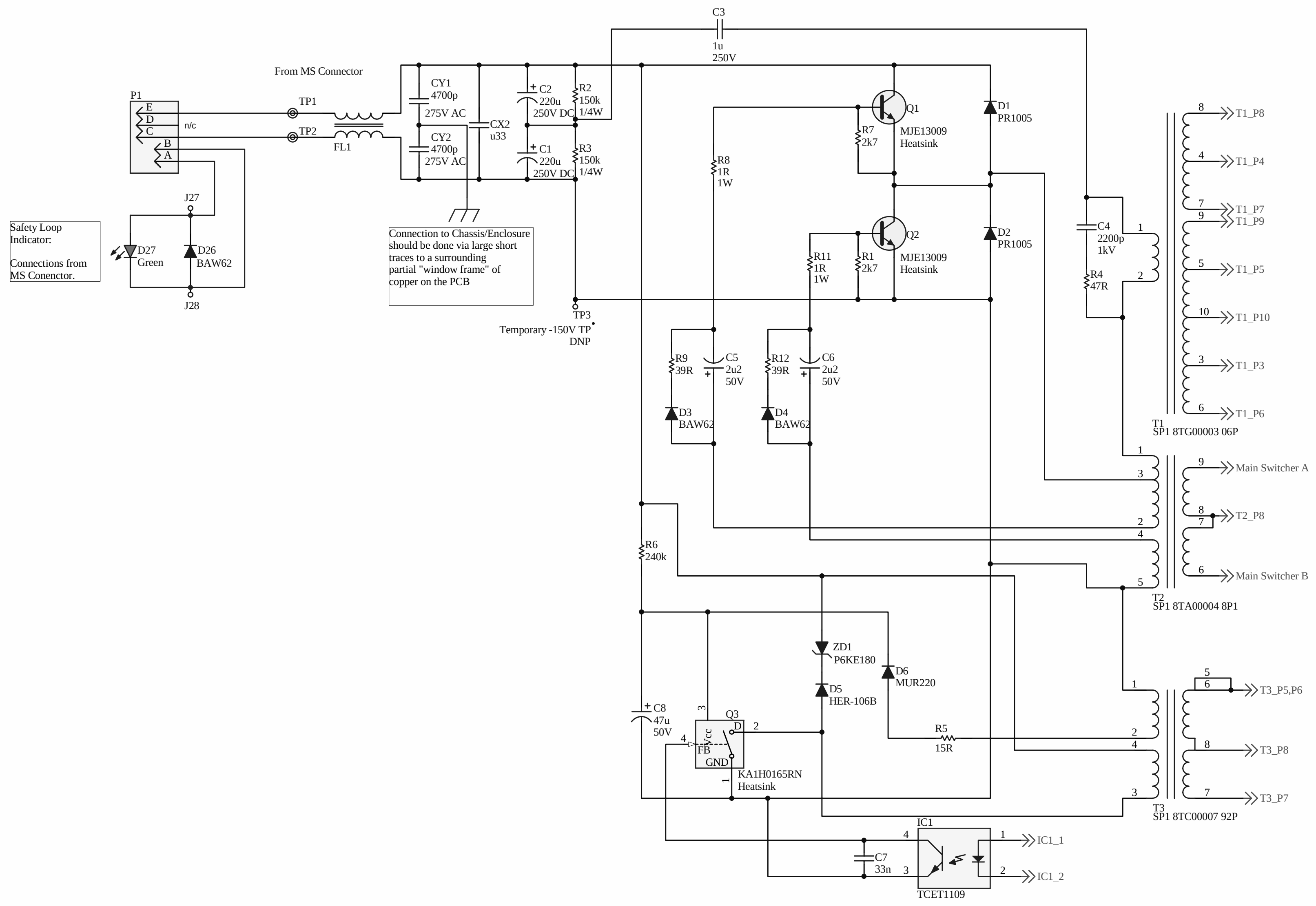
Hence on CC receipt of data block. Checkbyte + (all other bytes in datablock) should = 0.

5.2 Power Supply Card Schematic

PSU- All Sheets
PSU_S585_103B_1.SchDoc;PSU_S585_103B_2.SchDoc;PSU_S585_103B_3.SchDoc;PSU_S585_103B_4.SchDoc

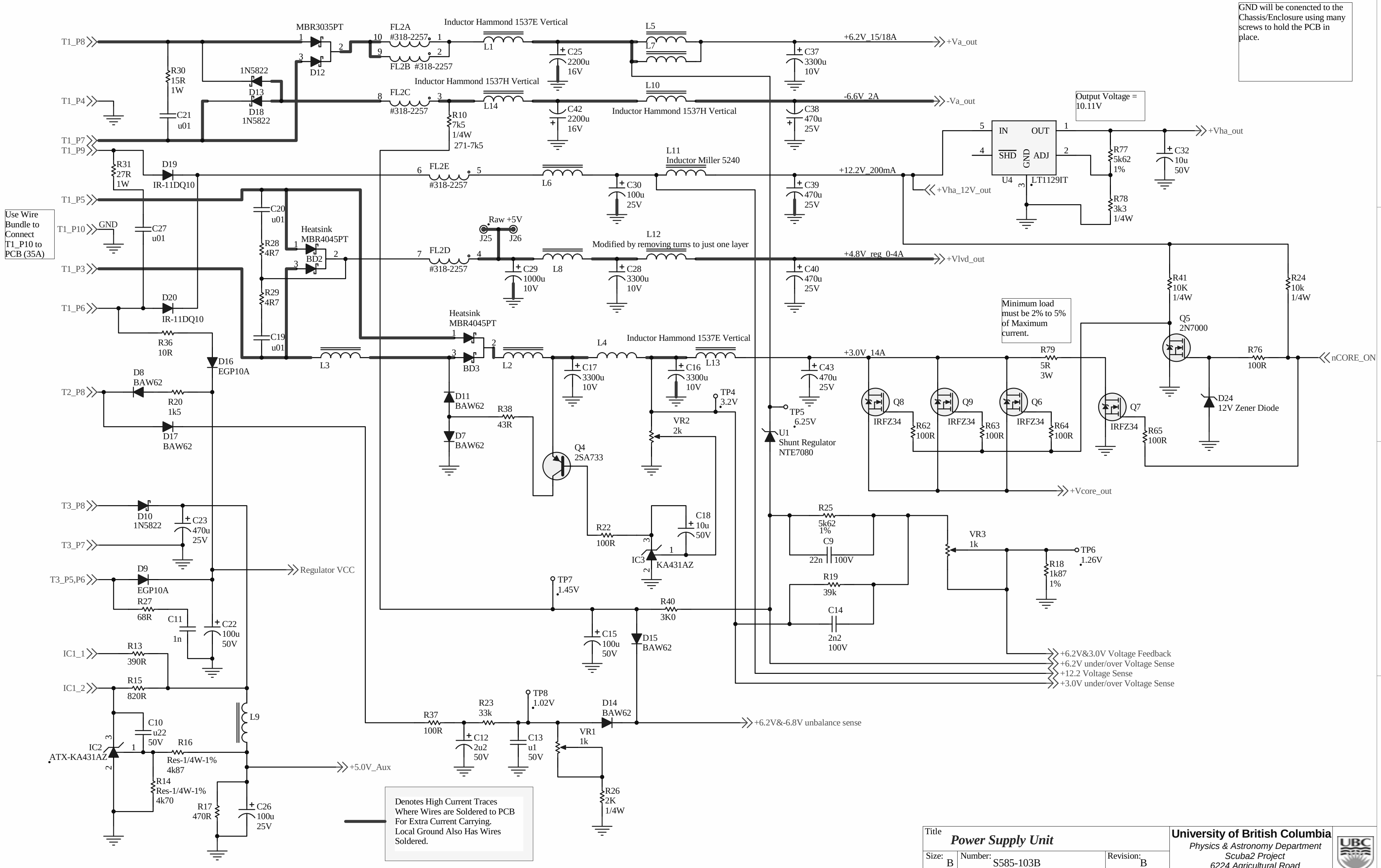
Title *Power Supply Unit*			**University of British Columbia** *Physics & Astronomy Department* *Scuba2 Project* *6224 Agricultural Road* *Vancouver BC V6T 1Z1 Canada*
Size: C	Number: S585-103B	Revision: B	
Date: 10/25/2006 Time: 3:15:00 AM		Sheet 1 of 4	
File: MASTERSHEET.SCHDOC			

?



Title: *Power Supply Unit*			University of British Columbia *Physics & Astronomy Department* *Scuba2 Project* *6224 Agricultural Road* *Vancouver BC V6T 1Z1 Canada*
Size: B	Number: S585-103B	Revision: B	
Date: 10/25/2006 Time: 3:15:01 AM		Sheet 2 of 4	
File: PSU_S585_103B_1.SchDoc			

?



Title **Power Supply Unit**		**University of British Columbia**
Size: B	Number: S585-103B	Revision: B
Date: 10/25/2006	Time: 3:15:01 AM	Sheet 3 of 4
File: PSU_S585_103B_2.SchDoc		

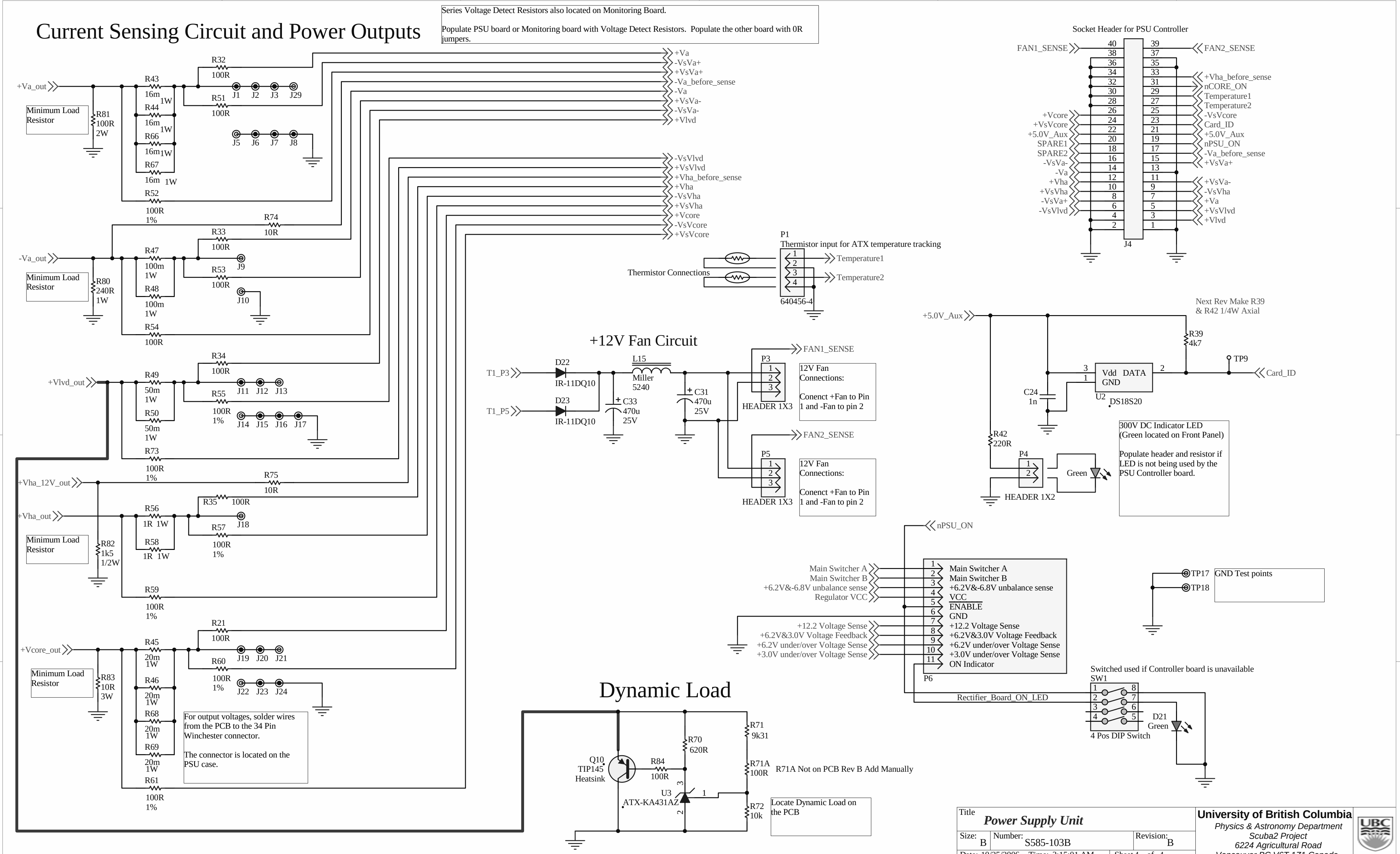
Physics & Astronomy Department
Scuba2 Project
6224 Agricultural Road
Vancouver BC V6T 1Z1 Canada

?

Current Sensing Circuit and Power Outputs

Series Voltage Detect Resistors also located on Monitoring Board.

Populate PSU board or Monitoring board with Voltage Detect Resistors. Populate the other board with 0R jumpers.

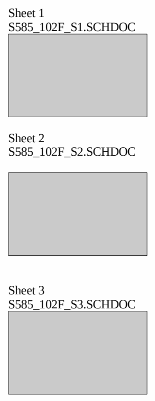


Title	*Power Supply Unit*
Size: B	Number: S585-103B
Revision:	B
Date: 10/25/2006	Time: 3:15:01 AM
Sheet 4 of 4	
File:	PSU_S585_103B_3.SchDoc

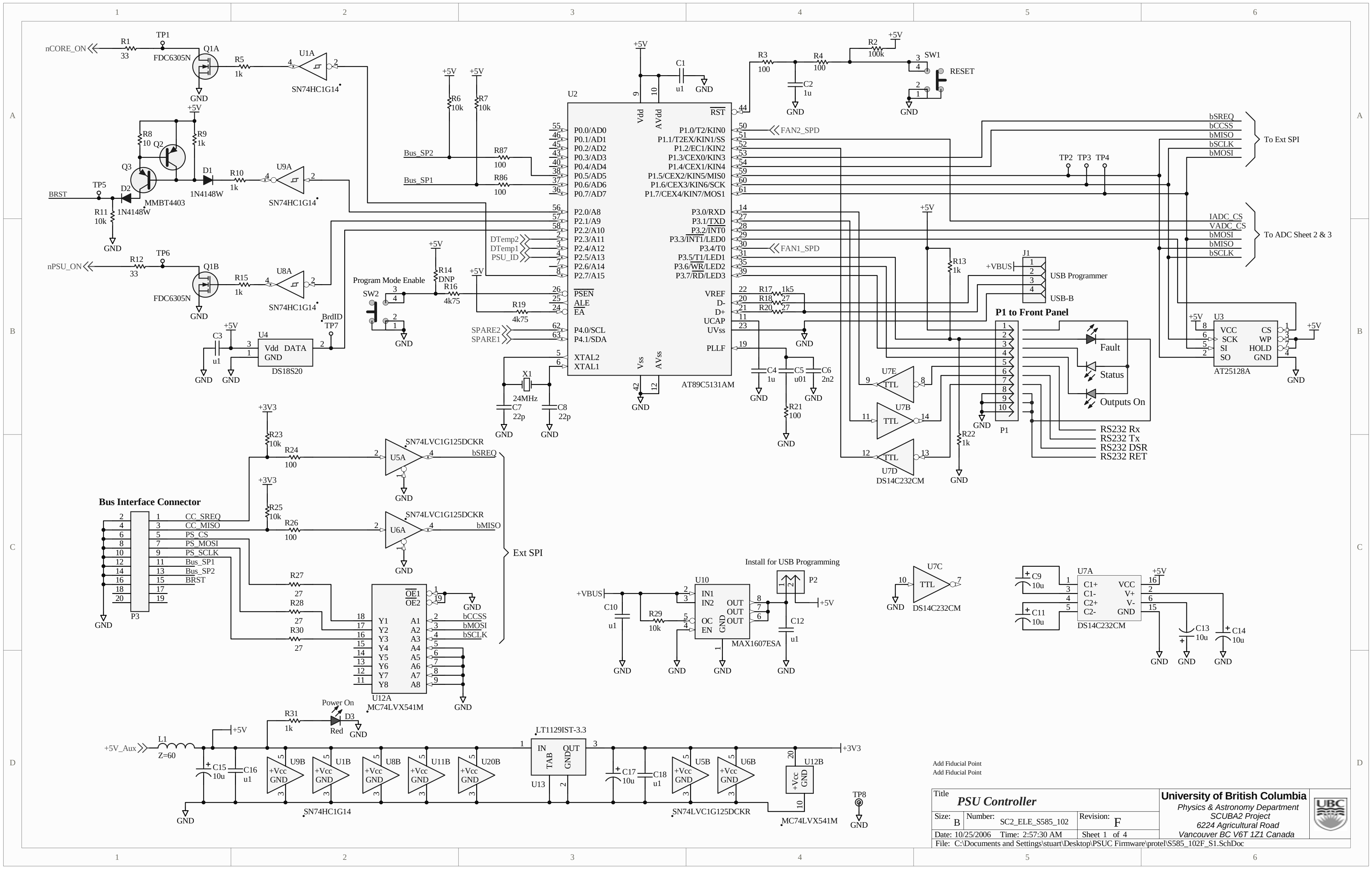
University of British Columbia
Physics & Astronomy Department
Scuba2 Project
6224 Agricultural Road
Vancouver BC V6T 1Z1 Canada

?

5.3 Power Supply Controller Card Schematic



Title **SCUBA2 Power Supply Card**			**University of British Columbia** *Physics & Astronomy Department Electronics Lab 6224 Agricultural Road Vancouver BC V6T 1Z1 Canada*
Size: B	Number: SC2_ELE_S585-002	Revision: F	
Date: 10/25/2006 Time: 3:03:40 AM		Sheet 0 of 3	
File: S585_102F_S0.SchDoc			



Title: **PSU Controller**

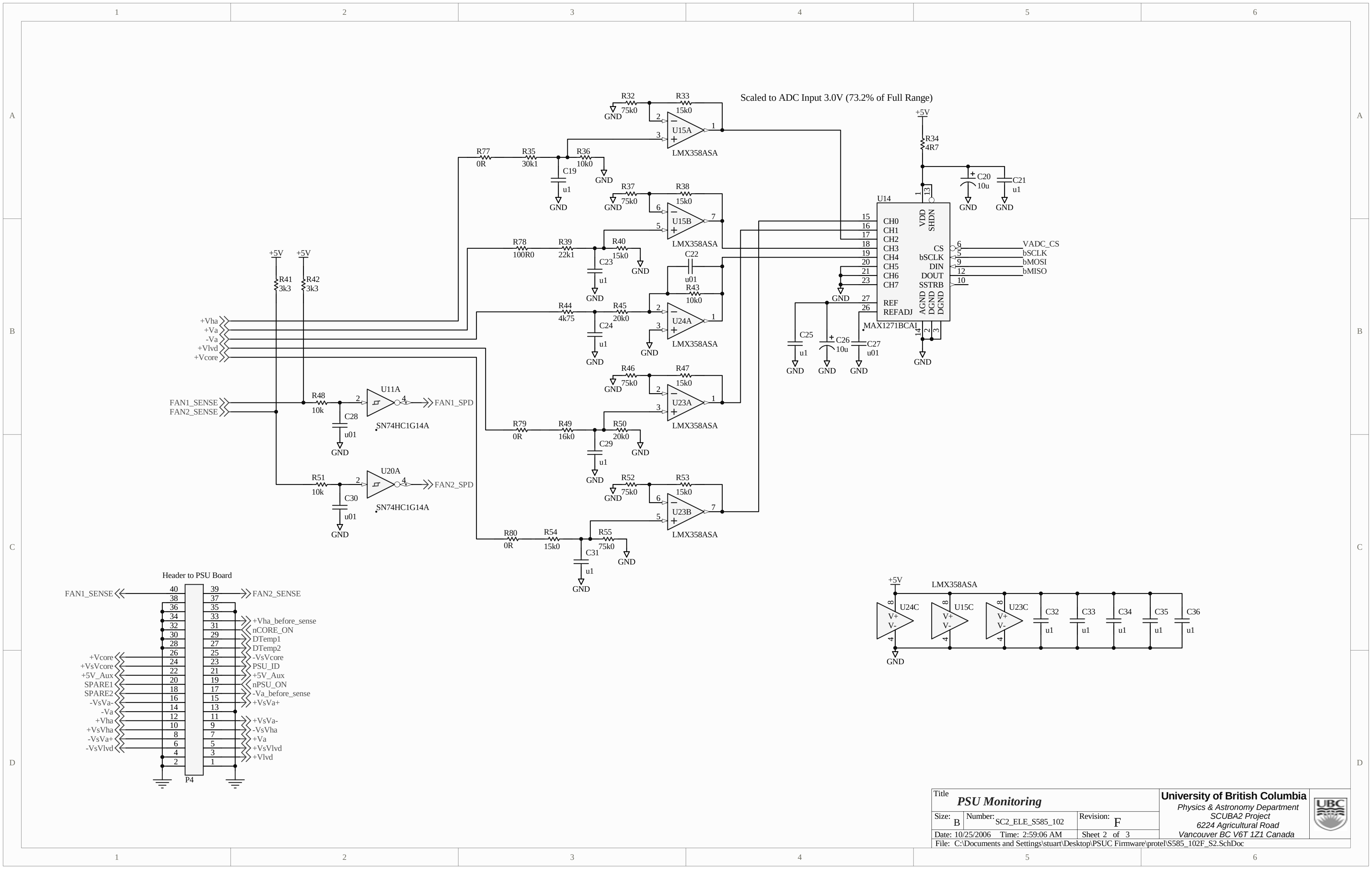
Size: B Number: SC2_ELE_S585_102 Revision: F

Date: 10/25/2006 Time: 2:57:30 AM Sheet 1 of 4

File: C:\Documents and Settings\stuart\Desktop\PSUC Firmware\protel\S585_102F_S1.SchDoc

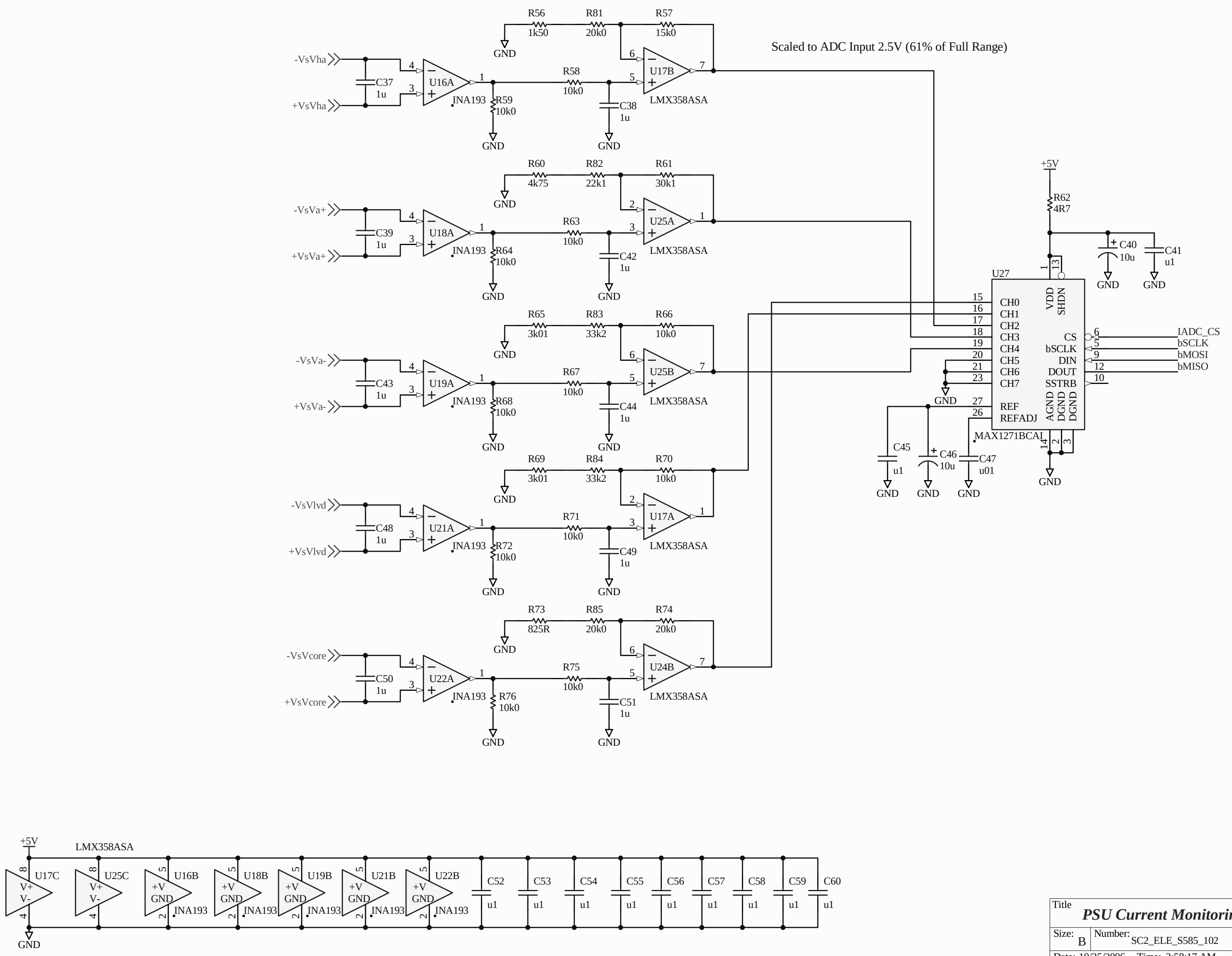
University of British Columbia

Physics & Astronomy Department
SCUBA2 Project
6224 Agricultural Road
Vancouver BC V6T 1Z1 Canada



Title	*PSU Monitoring*
Size: B	Number: SC2_ELE_S585_102
Revision:	F
Date: 10/25/2006	Time: 2:59:06 AM
Sheet 2 of 3	
File:	C:\Documents and Settings\stuart\Desktop\PSUC Firmware\protel\S585_102F_S2.SchDoc

University of British Columbia
Physics & Astronomy Department
SCUBA2 Project
6224 Agricultural Road
Vancouver BC V6T 1Z1 Canada



Title *PSU Current Monitoring*		**University of British Columbia**
Size: B	Number: SC2_ELE_S585_102	Revision: F
Date: 10/25/2006	Time: 2:58:17 AM	Sheet 3 of 3
File: C:\Documents and Settings\stuart\Desktop\PSUC Firmware\protel\S585_102F_S3.SchDoc		

Physics & Astronomy Department
SCUBA2 Project
6224 Agricultural Road
Vancouver BC V6T 1Z1 Canada

LIST OF ABBREVIATIONS / GLOSSARY

0x###: Hexadecimal number

AC/DC: Alternating Current / Direct Current

ACDCCU: (SCUBA2 MCE) AC/DC Conversion Unit

ADC: Analog to Digital Converter

ACK/NAK: Acknowledge / Non-acknowledge

AT89: Atmel AT89C5131AM Microcontroller

CC: (SCUBA2 MCE) Clock Card

CCSS: Clock Card Slave Select signal

DS: DS18S20 ID/Temperature Sensor

DMM: Digital Multimeter

EEPROM: Electrically-Erasable Programmable Read-Only Memory

FPGA: Field Programmable Gate Array

JCMT: James Clerk Maxwell Telescope (Hawaii)

PCB: Printed Circuit Board

PSU: Power Supply Card

PSUC: Power Supply Controller Card

MAX: MAX1271 12-bit ADC

MCE: (SCUBA2) Multi-Channel Electronics

MISO: Master Input Slave Output signal

MOSI: Master Output Slave Input signal

PCB: Printed Circuit Board

ROM: Read-Only Memory

RS-232: Serial data communication standard

SCUBA2: Submillimetre Common User Bolometer Array

SPI: Serial Peripheral Interface

SQUID: Superconducting Quantum Interference Device

SREQ: Clock Card Service Request signal

VHDL: VHSIC Description Language

VHSIC: Very High Speed Integrated Circuit

REFERENCES

Cited:

[1] Fich, Michel. “SCUBA-2: A Submillimetre Wavelength Camera for Astronomy.” http://astro.uwaterloo.ca/SCUBA2/Posters&Presentations/SCUBA2_descriptionV1.pdf

[2] Hadfield, Stuart. "Testing of SCUBA2 Multi Channel Electronics." 2004 (Co-op Report).

[3] UK ATC. “Unveiling the Universe at Submillimetre Wavelengths.” http://www.roe.ac.uk/ukatc/projects/scubatwo/whatis/summary.html

Internal SCBUA2 docs:

[4] sc2mce\system\sys_design\docs\MCE User's Manual\MCE User's Manual v13.pdf

[5] sc2mce\system\sys_design\docs\ functional_desc.pdf

Datasheets (available online or from manufacturer):

[6] Atmel AT25128A SPI Serial EEPROM

[7] Atmel AT89C5130A 8-bit Flash Microcontroller

[8] Atmel 8051 Microcontrollers Hardware Manual

[9] Dallas Semiconductor DS18S20 High-Precision Digital Thermometer

[10] Dallas Semiconductor “1-Wire Communication Through Software.”

[11] Maxim MAX1271 Serial 12-bit ADC

Further info:

[12] http://astro.uwaterloo.ca/SCUBA2/

[13] http://www.jach.hawaii.edu/

[14] http://www.physics.ubc.ca/~scuba2/

[15] http://www.roe.ac.uk/ukatc/projects/scubatwo/index.html