Driftsonde System Reference

Charles Martin

Research Technology Facility

Atmospheric Technology Division

National Center for Atmospheric Research

Table of Contents

List of Tables

List of Figures

1 Introduction

Figure 1. Driftsonde Motherboard

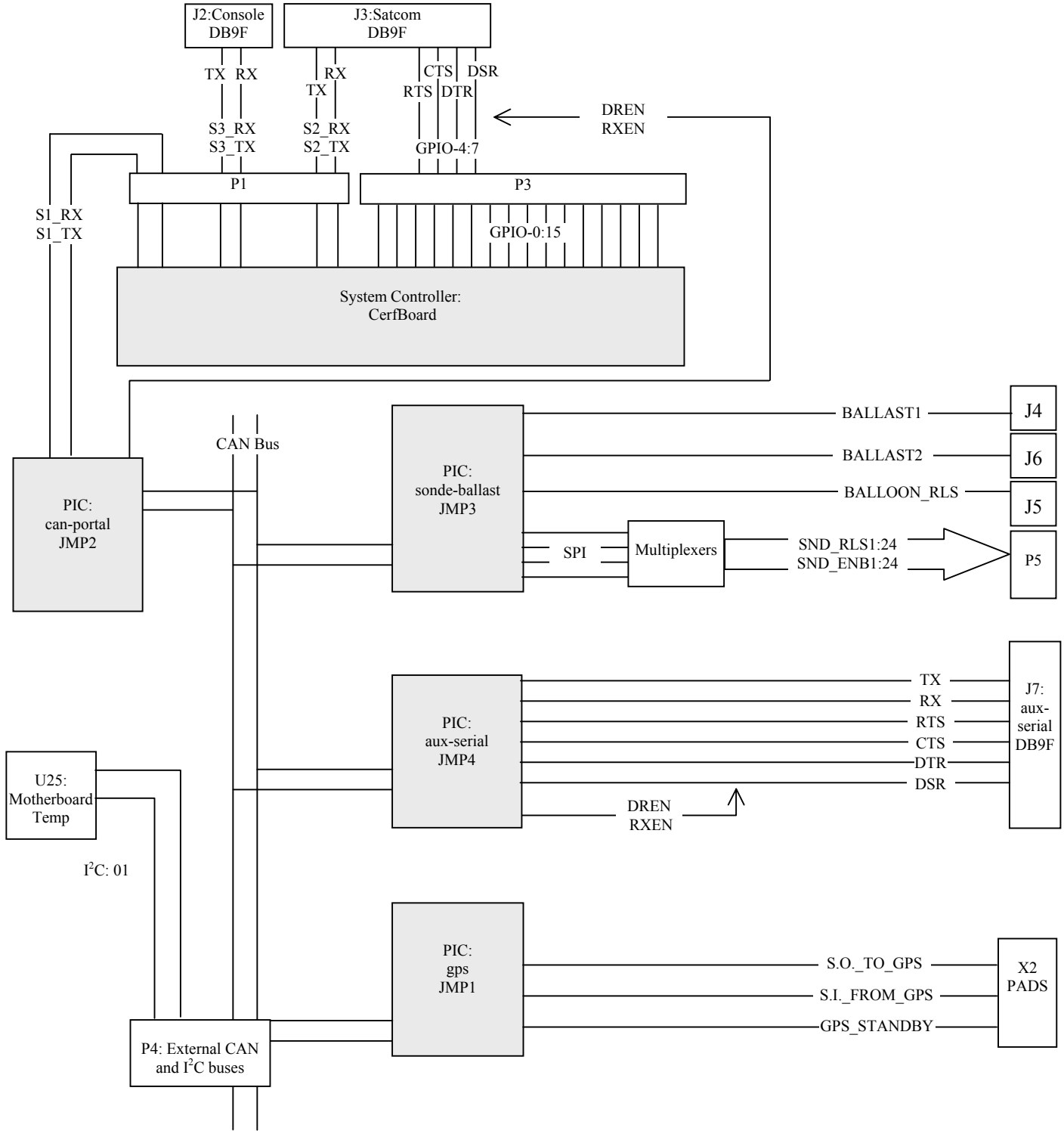
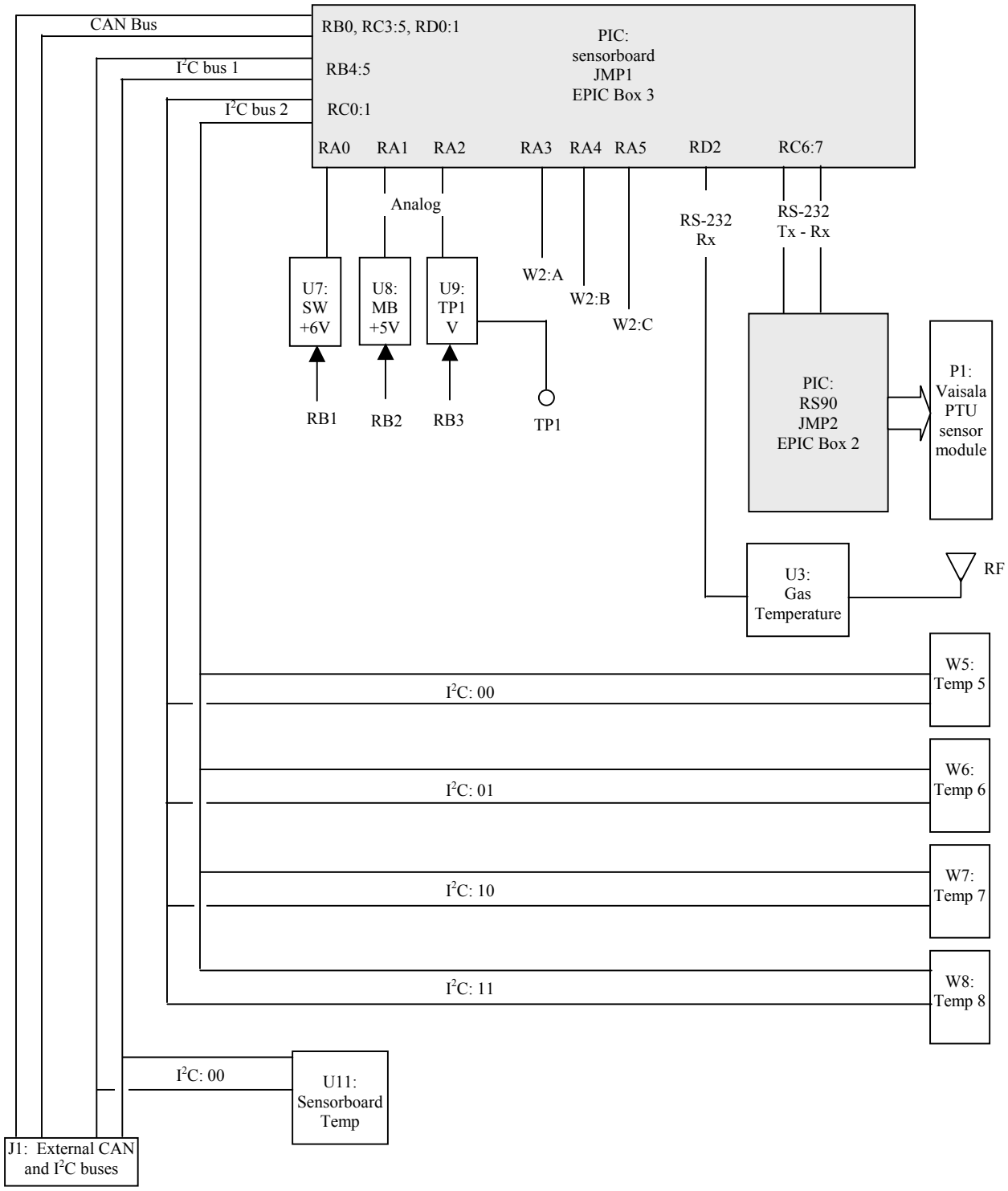


Figure 2. Driftsonde Sensor Board



2 CerfBoard

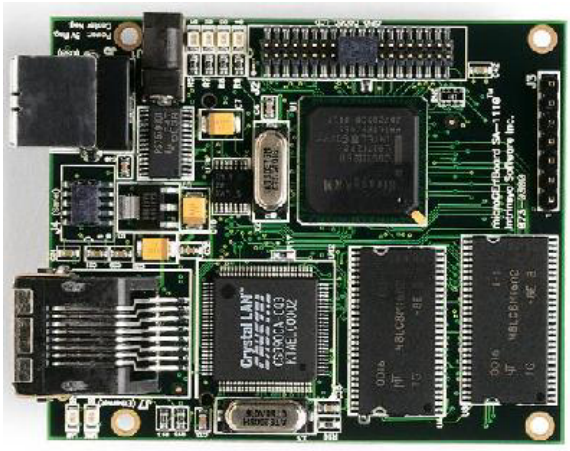


Figure 3. The CerfBoard

The CerfBoard is based on the Intel StrongARM SA-1110 processor, which is a highly integrated, low power RISC processor. This processor, with a clock speed of 206 MHz, is currently one of the most capable processors suitable for low power embedded applications. The chip has a large suite of I/O functions, which are made available off board. These functions include an Ethernet port, a slave (client) USB port, 3 serial ports, parallel I/O lines, and LCD, touch screen and audio device support. There are probably other functions that I have forgotten to mention.

The CerfBoard contains 32 MB of dynamic ram and 16 MB of non-volatile flash memory. It also has a PCMCIA slot, which can provide additional non-volatile file system space through the Compact Flash ATA/IDE mechanism.

The onboard flash memory contains a compressed kernel image, a compressed image of the root file system, and an uncompressed image of the usr file system.

2.1 Booting

At boot time, the kernel is uncompressed and read into the dynamic ram. The same is done for the root file system, except that it becomes a ram disk. The kernel is started, and the usr file system in flash is mounted (read-only) as /usr.

The normal Linux boot sequence is followed, with hardware identification and other initialization tasks occurring first. When *init* is run, it follows the standard procedure as directed by /etc/inittab. The default run state is 3 for a headless CerfBoard.

Note that the standard Intrinsyc distribution contains a script for mounting extra disks: */etc/rc.d/init.d/S80disk*. You would think that this is the appropriate place to mount the extra Compact Flash disk. However, at this point in the boot sequence, the compact flash device has not been identified by the ide/pcmcia driver (ide_cs). It turns out that the PCMCIA driver can be configured to automatically mount the compact flash disk whenever it is detected, and this mechanism works well. Configuration of the pcmcia/ide driver is described in "ATA/IDE Compact Flash", on page 5.

2.2 The boot loader

The Cerfboard bootloader is run immediately upon power up of the system. It pauses for a few seconds to allow the user to take control (by hitting a key), otherwise the Linux software is loaded from flash and executed.

If the user interrupts the boot loader, then a series of commands may be entered. The most important of these are commands that are used download new file system or kernel images to the CerfBoard ram, and then write them to the flash memory, so that the system software may be upgraded.

The boot loader also allows a command script to be executed. Thus, frequently used commands are typically packaged into a small script file on the host system. This script is downloaded and written to the flash, and then executed. Note that the script is retained in the flash, so that it may be used repeatedly as needed without repetitive down loads. The actual procedure is described in "Installing on the CerfBoard", page 33.

2.3 TFTP

The Trivial File Transfer Protocol (tftp) is used to download files to the CerfBoard from a host on the network. Thus the tftp facility must be installed and configured on a network host. In the current RedHat 7.2 system, tftp is not available by default, but must be installed from RPMS. Tftp seems to be configured using xinetd nowadays, and so xinetd must also be installed. There will be an */etc/xinetd.d/tftp* file which configures tftp. The location of the directory to and from which all files are transferred, during a tftp session, is specified in */etc/xinetd.d/tftp*. Thus, any

file system image that is to be downloaded to the CerfBoard must be placed in the tftp directory.

2.4 Bootp

Since the CerfBoard boot loader is using tftp, the CerfBoard does not yet have a network address. Therefore, the boot loader makes use of bootp to get an address for the CerfBoard. Bootp must be installed and configured on the network. It is not a normal installation feature of RedHat 7.2, and so I had to find an RPM (at rpmfind.org), install and configure it. The */etc/bootp* file contains a mapping between MAC addresses and IP addresses. The CerfBoard discards this IP address when Linux is running. It appears that the IP address has to be a legal address on the network. I tried using a 192.168 address, and the CerfBoard could not obtain the IP address.

The only small problem is that the CerfBoard MAC address must be known in order to configure bootp. If the CerfBoard is able to boot Linux, then the MAC address is displayed during the Linux startup, and can also be displayed with the ifconfig –a command. If the Linux system is trashed in the flash memory, and we have not recorded the MAC address of the CerfBoard, I don't know how we will determine the MAC address.

3 Cerfboard Linux V3.1

3.1 Software Distribution

(See Appendix C: CerfBoard Linux V3.1 for a more detailed description of this)

Intrinsyc provides a comprehensive software distribution with the CerfBoard. The set includes build environments for all of their Linux based products; i.e. the CerfCube, CerfPOD and CerfPDA. The CerfBoard uses the CerfCube infrastructure.

There are four main segments of the Intrinsyc distribution[1]:

- The Linux kernel sources
- Optional, but standard Linux contributed packages, such as ssh, strace, perl, etc. Both source and pre-built binaries are provided in some cases, in others only the sources are provided.
- The cross tool chain, which is required in order to build the kernel, and to compile applications to run on the CerfBoard.
- Pre-built kernel binaries.

A note accompanying the CerfCube Linux software CD says that the CD might be incomplete and that the distribution should be downloaded from the Instrinsyc website: ftp://humbolt.intrinsyc.com. This appeared to be true, since the build failed for the build of the cross tool chain. After the distribution was fetched from the website, the cross tool chain builds correctly.

For Driftsonde usage, the cross tool chain must be built. Rebuilding the kernel, thus far, has not been necessary, but maybe in the future if the kernel configuration needs to be changed.

There are two major directories in the distribution: intrinsyc-linux and intrinsyc-tool chain. The cross tool chain needs to be built first. If the tool chain will be used to compile the kernel for the CerfBoard, some cross-libraries must be built[2]. After this, Linux can be cross-compiled for the CerfBoard.

Note that the "Cerf™Cube for Linux Software User Guide, version 3.1" says that a precompiled copy of the cross tools are provided; look as hard as you might, but they are not to be found in the distribution. Itrinsyc should update the main documentation, which is confusing in the least.

The instructions for building the cros toolchain are found in *intrinsyc-toolchain/doc/cross-toolchain-building.txt*. A stock build is done with a simple "make" in the intrinsyc-toochain directory.

The instructions then tell how to build the cross libraries. Oddly enough, these are found in the intrinsyc-linux tree.

Finally, Linux can be configured and built, according to the directions in *intrinsyc-linux/doc*.

3.2 Flashing the firmware

Make sure that tftp is configured on the development host, and the correct files are in the tftp download directory. See section ? of the CerfBoard Software manual.

The Intrinsyc iBoot V3.1 bootloader is used for the following procedures. This relies on the presence of both bootp and tftp on the host machine.

Make sure that there is an entry in /etc/bootptab for the CerfBoard. Make sure that the entry is unique, as I had trouble when I tried to share IP names between two MAC addresses. Simply disable the MAC addresses not being used, if a unique IP address is not available, since bootp is only needed for a kernel reflash.

Assume that we are now using the iBoot bootloader. Halt the boot process by typing a character during the RTC stabilize. You will get the iBoot prompt. Use the following commands to erase the flash and execute a utility script via tftp. The flash must be erased because jffs2 will combine the remnants of a preexisting file system if it was larger (almost certainly) than the new one being installed. Be

[1] Note that the distribution directory structure does not match this breakdown of function.

[2] These libraries may also be required for Driftsonde applications.

careful with the *eraseflash* command to not destroy the bootloader.:

```
Iboot>eraseflash 3
Iboot>getaddr
Iboot>exec tftp:x.x.x.x loadlinux-cube
Iboot>boot
```

3.3 Setting up for Driftsonde

The attempt is to use an Intrinsyc Linux distribution with as little modification as possible. Presently we are able to use the stock distribution, without building the kernel or including packages in the firmware image. We do have to manually install a package or two (e.g. sshd) onto the CerfBoard's flash system.

The */etc/rc.d/rc.sysinit* file does a lot of system configuration that is specific to the embedded aspects of Linux used on the CerfBoard. It is a good place to start when trying to figure out how the system services are started.

3.3.1 Set the IP address

Add the CerfBoard's IP address to */etc/hosts*, and put the hostname in */etc/HOSTNAME*.

The IP address can be statically set by editing the ifconfig command found near the end of */etc/rc.d/rc.sysinit*:

```
ifconfig eth0 driftsonde2 netmask \
255.255.255.0 up
```

3.3.2 Add secure shell

The *openssh* package is found in the Intrinsyc contributed packages. The *.tgz file contains precompiled binaries; all of which will install under */usr* on the CerfBoard.

After installing on the CerfBoard, edit */usr/etc/sshd_config*, and adjust the permissions appropriately.

It appears that the "build" script can be used in some way to integrate this package directly into the cerfcube distribution; perhaps by copying the whole package into the base tree. Need to do some research to find out how to do this. Alternatively, we'll just put this package somewhere in a distribution tree with the application source code.

3.3.3 Add minicom

Copy the directory *intrinsyc-linux/contrib/minicom-2.0.0* to *intrinsyc-linux/base/src/*. Do a "make TARGET=cerfcube" in *intrinsyc-linux/base/src/minicom-2.0.0*. This will unpack and patch the package, and then build it, and then put all of the required pieces into a tarfile *intrinsyc-linux/base/targget/cerfcube/packaes/minicom-2.00.0/minicom-2.00.0.tgz.*

The tarfile can be copied to the CerfBoard and installed directly into the file system.

Need to determine how to build a new binary distribution from the base tree, so that the package is included in clean system installs.

Run minicom, and configure the follow three settings. For each of the settings, remove all of the modem dial strings, and set the lock file location to */tmp*.

Name	Device	Rate
ttySA0	/dev/ttySA0	38400,N,8,2
ttySA1	/dev/ttySA1	9600,N,8,2
ttySA2	/dev/ttySA2	38400,N,8,2

3.3.4 Buid and install gdbm

Get the gdbm package from a GNU mirror, in the gdbm directory. On 9 Jan 2002. I got version 1.8.0. Unpack into a directory.

Configure and compile the library and the test programs:

```
setenv CC arm-linux-gcc
rm config.cache
./configure –host=arm
make
make testgdbm
```

This will build the library in .libs/libgdbm.a. The *testgdbm* target will build the *testgdbm*.

Copy testgdbm to the CerfBoard, and verify that it run properly.

When compiling applications that will use gdbm, make sure that the include files can be found in the distribution source directoy,and the library in .libs. Otherwise, move .libs/libgdbm.a and the header files to a more convenient location.

3.3.5 Build and install cron

Get the vixie-cron package from the RedHat distribution, or elsewhere. I used vixie-cron-3.0.1-62.src.rpm. This rpm contained a tarfile for vixie-cron, as well as 19 patch files.

Use "*rpm –Uhv vixie-cron-3.0.1-62.src.rpm*" to unpack the source files to */usr/src/redhat/SOURCES*. A "spec" file is written to */usr/src/redhat/SPECS/ vixie-cron-3.0.1-62.spec*.

Copy the tarfile and patches to a user directory, and unpack the tarfile. Apply the patches, in the order listed in the spec file.

Next, look at the README file, and follow the instructions. I enabled sys logging in config.h, and set CC=arm-linux-gcc and OPTIM=-O2 in the Makefile.

The package built without problem. I manually stripped *cron* and *crontab*. *Cron* was copied to */usr/sbin/crond. Crontab* was copied to */usr/bin*, and set as setuid (root).

Create the directory */etc/cron.d*, or else cron will fail at startup. Files in this directory can contain system defined crontabs, which will be preserved over system restarts. These crontab files contain an extra field right after the 5 time field, which contains the user name. There are naming restrictions for the files in /etc/cron.d; it is best just to use simply names of alphanumeric characters.

A startup script (vixie-cron.init) is provided with the package, but I couldn't get it working and so I just created a simpler one: */etc/rc.d/init.d/crond*.

Just running *cron* starts the *cron* daemon. Diagnostics appear in */var/log/messages*.

To edit a crontab, export an EDITOR variable: *export EDITOR=jmacs*,

And then run *crontab –e*. Don't forget to include a line: MAILTO="" in order to stop *cron* from trying to mail messages. Note that crontabs created with crontab will disappear after reboots, since they are stored in */var*, which is recreated at boot time; use */etc/cron.d* instead.

3.3.6 Add startup scripts

Scripts are included in /etc/rc.d/init.d/ to start facilities used by the Driftsonde. Some of these facilities are stock Linux packages, and others are application software specific to Driftsonde:

- Ssh: add */etc/rc.d/init.d/sshd*, and link to it from */etc/rc.d/rc3.d/*.
- *Crond: add /etc/rc.d/init.d/crond* and link to it from */etc/rc.d/rc3.d/S9crond.*
- Multiplex: add */etc/rc.d/init.d/multiplex*, and link to it from */etc/rc.d/rc3.d/S99multiplex* and */etc/rc.d/rc3.d/S99multiplex.*
- After ftp'ing over, be sure to *chmod a+x*.

3.3.7 Add NFS mount (optional)

3.3.8 Configure users

Set a password for root!

Add an icarus account:

```
Newgroup icarus –n ?
Adduser icarus –n ?
```

Create *~icarus/.profile*, containing the following:

```
export PATH=$PATH:~icarus/
```

3.3.9 Disable telnet and ftp services

Edit */etc/inetd.conf*, and comment out the telnet line and the ftp line.

3.3.10 Odds and Sods

Change the time zone in */etc/profile* to "GMT".

Put the hostname in */etc/HOSTNAME.*

Change permissions on selected files, by adding the following to */etc/rc.d/rc.sysinit*:

```
Chmod a+rw /dev/null
Chmod a+rw /tmp
Chmod a+rw /dev/ttySA*
```

3.3.11 Install Driftsonde software

After ftp'ing over, be sure to *chmod a+x*.

- Multiplex
- AspenQC
- Gps – be sure to set the owner to root, and setuid.
- CanTalk
- driftStat

4 Driftsonde System Controller Software

4.1 microdb

Name	Description	Format
gpstime		
gpsunixtime		
gpsposition		
sysclockreset		

4.2 multiplex

This application is used to multiplex/demultiplex the communcations stream between the CAN bus and the Linux applications running on the System Controller.

The basic structure is that a family of Linux named pipes is used for communication between applications and individual CAN nodes. Two pipes are used for each CAN node. An application can send messages to a CAN node by writing to the appropriate pipe; likewise it can receive messages by reading from another pipe.

For instance, to send a message to CAN node 4, write to the pipe named *canTo004*. To read data from CAN node 4, read from pipe *canFrom004*. The pipes are named in the sense of the final application program; i.e. an application program writes **to** the *canTo* pipe to send data **to** the CAN node. It reads **from** a *canFrom* pipe to get data **from** a CAN node.

The combined CAN bus character stream is accessed through a serial port (e.g. */dev/ttySA2*). The *multiplex* program reads this serial stream, and handles the dispersing of received CAN messages to the appropriate *canFrom* pipe. Multiplex also reads data from the *canTo* pipes, and sends that to the CAN bus by writing to the same serial port.

Note: Multiple applications should not access the same CAN node concurrently. Linux pipes do not provide semantics for multiple readers and writers.

4.2.1 Can-portal protocol

The can-portal PIC, on the Driftsonde motherboard, provides the interface between the System Controller and the CAN bus. The physical interface is an RS-232 line, configured at 38400[3] baud.

The can-portal PIC has a CAN node address of 2. All traffic on the can bus addressed to node 2 is written to the output serial line of this PIC. Data characters have bit 7 forced to zero. The PIC will put a node number indicator out when the data stream switches to a different CAN node. The node indicator is a character that has bit 7 set. The low 5 bits of the character then contain the node number.

Similarly, the can-portal reads its incoming serial stream, and places received characters on the CAN bus. The destination node is changed by reception of a node indicator on the incoming serial stream.

4.3 ttyReport

4.4 report

4.5 driftStat

This is a script that collects system status information, and places it in the "driftstat" entry in the system state. It is run periodically from the icarus crontab.

[3] 38400 baud was chosen as the line speed because it is four times 9600. A maximum aggregate throughput of four continuous 9600-baud data streams seems like a reasonable limit for the system.

4.6 orbcomm

This process manages communications via an Orbcomm satellite communicator (SC). The SC is configured to operate in "byte mode", and is accesses via a serial port.

Byte mode operation simply means that characters received by the SC from the serial port are transmitted as messages, and the SC writes over-the-air received messages to the serial line. There is no other control of the SC by the system controller.

The "character initiated" mode, within byte mode, is used for message control. Originated messages are bracketed by special characters before being sent to the ST2500. Terminated messages are bracketed by special characters (by the ST2500) before being sent to the DTE.

It is critical to get the byte mode of the ST2500 configured properly, and tested before flight, for both originated and terminated messages.

Orbcomm interacts with other processes in the system via the global database, which is used in a "mailbox" fashion. Messages that are destined for transmission are placed in the database, where orbcomm finds them and pulls them out to send. When it pulls the message out, it sets the value in the database to a null, to indicate that the message has been processed. Similarly, when a command comes in, orbcomm will put that command into the database. The consumer of that command is expected to clear the database value when it has processed the command. Orbcomm will queue up messages while waiting for the consumer to become available.

Perhaps we need to have a priority or out-of-band feature in the command syntax that allows incoming commands to flush the queue?

Database "mailbox" name	Producer	Consumer
SendEngReport		

4.6.1 Program Structure

Orbcomm contains four main elements:

- The initial thread
- A receive thread
- A send thread
- A shared Logger

The Logger is a class that will store text to a log file, and optionally write it to the terminal. The Logger uses a mutex to control access to internal data, so that one logger can be shared among threads. Text characters are buffer up until a newline is encountered (or the buffer is full) and then the whole line is written out with a time stamp. Each thread calls Logger with a unique id, so that unique text lines can be collected for each thread.

The initial thread creates the shared Logger, spawns the receive and send threads, and then periodically writes a mark to the Logger.

The receive thread blocks on a read from the SC serial port. All incoming traffic is written to the log, with the exception of superfluous newlines. Incoming characters are processed via a state machine, described below.

The send thread loops continuously, with one second sleeps, waiting for entries to appear in the outgoing message mailboxes of the system database. When a message appears, the send thread writes it out to the SC serial port, and sets the value in the database to an empty string. All outgoing traffic is also sent to the Logger.

4.6.2 Receive thread state machine

Upon receipt of a character, the state machine evaluates the character, may perform an action, and transitions to the next state.

State	Action	Next State
Idle	Wait for new character:	
	If new char == "<"	Sync
	Otherwise	Idle
Sync	Wait for new character:	
	If new char == 0	Collect
	If new char == ">"	Idle
	Otherwise	Sync
Collect	Wait for new character:	
	If (new char == ">") || (new char == 0)	Process

	If length > maximum (ignore character)	Process
	Append character.	Build
Process	Process message; Clear message	Idle

4.7 sonde-ballast

4.8 gps

Gps reads from the GPS CAN node. It does the following:

1. Saves the GPS retrieved time in the format of the GPRMC NEMA string, with the validation indicator appended, to the "gpstime" state variable.

2. Save the same time information, in Unix time format, to the "gpsunixtime" state variable.

3. Saves the current gps position, along with the validation indicator, to the "gpsposition" state variable.

4. Sets the system clock to the gps clock, if the absolute difference is greater than 5 seconds. If the system clock is set in this way, the gps time is stored in the same format as 1, to the "sysclockreset" state variable.

- NOTE: Because *gps* can set the system clock, it must be owned by root and have setuid enabled.

4.9 canTalk

CanTalk is used to send and display characters to and from a CAN node, using a terminal interface. It provides a link between *canTo*, *canFrom* and */dev/tty*. When canTalk is running, charcters typed by the user are sent to the CAN node, and characters received from the CAN node are printed to the terminal.

Usage: canTalk <n>

<n> is the CAN node number.

Use ctrl-c to exit.

5 Orbcomm

The Orbcomm system is a communication facility based on a constellation of approximately 30 LEO satellites. An inexpensive satellite communicator (SC) provides low bandwidth two-way messaging capability. Internet protocols (mainly email) are the mechanism for transferring information between the user and the SC.

The business model of Orbcomm is targeted at the deployment of large numbers of SCs, with each one periodically sending small data messages back to the home system. Asset tracking, fleet diagnostics and remote site monitoring are typical applications.

It is a bit difficult to determine exactly what the true system bandwidth capabilities are, but the Orbcomm system documentation, and the airtime pricing schedules lead one to conclude that a typical SC might be sending messages of 10s of bytes, several times a day. The pricing schedule seems to be structured to accommodate fleet type deployments that only tolerate small monthly fees per unit.

Driftsonde, on the other hand, wants to make use of much higher data bandwidth. The nominal messaging scenario for Driftsonde calls for 70 byte status messages sent 4 time per hour, and 1000 byte TEMP messages sent twice per day. The Orbcomm reseller[4] that we are working with offers a price structure that will meet these requirements at an affordable price. Driftsonde is at the top end of the their pricing scale, but it works for Driftsonde since the number of units is small and the lifetime of a single Driftsonde is short. One month of service will probably end up in the $50-$70 range, giving a communication cost of $2-$3 per sounding.

As important as the airtime pricing is the issue of whether Orbcomm can support the data rates that Driftsonde requires. This is really a function of the frequency with which a satellite is in view of the SC, how soon it makes contact with a ground station, and competition with other SCs at the same time.

Currently, competition with other SCs does not seem to be an issue. Steve Mazur reports that the Orbcomm system is presently operating at 3% capacity.

[4] Stanley Associates

The satellite coverage appears to be very good. The satellites are in inclined orbits, and orbit frequently enough so that one is in view for a surprisingly large percentage of time. The “Orbcomm View” software can be used to simulate the behavior of the satellites; it is very instructive to experiment with this program to get a sense of the frequency of satellite passage. It is also useful to run the “Stelcomm” software (described below) and watch the connection and signal strength characteristics as the satellites pass over.

The most problematic issue with the system performance is the distribution and utilization of ground stations. An Orbcomm satellite will collect messages from an SC as it passes over, and pass this message on to a ground station when it comes within view. The viewing footprint of a satellite is actually quite large, and so there is often a good possibility that both the SC and the ground station are in view simultaneously, in which case the message is transmitted almost immediately. In other cases, the message must be held until a ground station is reached. In Driftsonde lab testing, using the nominal data rate described above, typical latencies have been a couple minutes, with the longest ones reaching 9 or 10 minutes. This agrees with information on the Orbcomm web site, which states:

“The message delivery time (or "latency") depends on the specific hardware setup, message size and location of the remote unit. In the continental US, ORBCOMM is committed to providing 90% in 6 minutes or less, and 98% in 15 minutes or less for all message traffic.”

In the continental United States, there are a reasonable number of ground stations (WA, AZ, NY and GA), which accounts for the observed performance. The two stations in the west (WA and AZ) will probably provide reasonable latency performance for eastward propagating Driftsonde launched from Hawaii. However, it is unclear what the performance may be for other Driftsonde tracks. There are very few Orbcomm earth stations in other parts of the world.

Another issue concerns the “provisioning” of the SCs. It appears that a communicator can only communicate with selected ground stations. Thus only the U.S. ground stations are available to our SCs, as currently configured. If the satellite misses one of these stations, it must wait until the orbit brings it over one of them again. I’m not sure if this is business artifact of the system, since different countries seem to have different vendors providing

the communication service, or if it is strictly a configuration issue with the SC.

Steve Mazur has provided a document that describes application development for "roaming" with Orbcomm; this appears to be quite complicated. It seems to involve some knowledge

The "provisioning" issue needs to be examined in detail. It could have disastrous impact on Driftsonde Orbcomm usage for trans-Atlantic and trans-Pacific flights.

Orbcomm sells software (LEOsphere) for simulating and analyzing satellite access behavior. Unfortunately, they charge $600 for the package. It would allow us to answer many of the questions about Orbcomm's suitability for Driftsonde.

5.1 Orbcomm Gory Details

Orbcomm publishes a number of extensive documents that describe the system. They are usually in the form of specifications. After some initial befuddlement, it became clear that these specifications describe both how the satellite communicators interact with the other Orbcomm infrastructure, as well as mandating a specific standard for interoperation with the end user equipment.

Defining a standard for the end user interoperation in theory allows the drop in replacement of SCs from differing vendors.

In Orbcomm parlance, a message sent by the SC to the earth segment is called an "originated" message. A message received by the SC from the earth segment is called a "terminated" message.

5.1.1 Protocols

The Orbcomm specifications require that the vendor provide two serial interface protocols for using the SC: "Orbcomm Protocol Mode", "Byte Mode", and an optional "Command Mode".

The Orbcomm mode allows the end user full access to configuration of the SC behavior. It uses a strict message format, complete with checksums. The end user software must be fairly sophisticated to make use of this protocol.

The byte mode allows the unit to be used with minimal interaction. Originated messages can simply be streamed out to the SC for transmission; likewise SC terminated messages can be simply sent out to the end user equipment.

The command mode is intended to be a vendor proprietary protocol. Perhaps this is to allow a vendor to "add value" or differentiate their product?

5.1.2 Message Types

There are (at least) three different message types defined by Orbcomm: the report, the message and the globalgram.

The report is a very small message (6 data bytes), with a variety of fixed data contents. It appears that this message is meant to be very efficiently processed by the Orbcomm system; perhaps it is inexpensive to send as well.

The message is an arbitrary text message, of variable length. It is stated in a couple of places that messages up to 2 KB are practical. Steve Mazur also said that there is a "sweet spot" in system for messages less than 107 bytes in length; that the transmission time for any message sizes less than this would be the same. Perhaps this relates to the time slots available during the SC-satellite communication? In any event, longer messages are broken up by the SC and reassembled by the ground segment, to arrive intact at the destination. Recent testing showed that 1600 byte messages, sent twice a day, went through quickly and reliably.

The Orbcomm satellites have a capability to forward a message to a nearby satellite, when a ground station is not visible. This is known as a globalgram. It is not clear what the restrictions are on globalgrams, and what the price penalty would be. These might be very useful for Driftsonde TEMP messages.

5.2 Stellar ST2500

The Stellar ST2500 communicator is purchased directly from Stellar (www.stellar-sat.com, 703-234-7832). Recently (Jan 2002), our contact at Stellar has been Steve Mazur (steve@stellar-sat.com, 703-234-4186), General Manager. Steve worked as an engineer at Orbcomm before coming to stellar, and he is the best technical resource for Orbcomm that we have encountered thus far. Heidi Flaugh (heidi@stellarsat.com, 703-627-4680) is our service representative for Stellar, and has been very responsive as well. Although her phone number is on

the east coast, she is currently based in Colorado Springs, and she can be reached at that number.

5.2.1 Stelcomm

The Stelcomm Windows program is used to initialize and maintain the ST2500. Connect a comm port (1-4) on the PC to the DB-9 serial connector on the ST2500.

Stelcomm interacts with the ST2500 using the "Orbcomm Protocol" mode of communication. Since we use the ST2500 in byte mode, it is likely that the unit will not respond initially to Stelcomm. However, ST2500 operates in protocol for a couple of seconds after power up. To synchronize Stelcomm with the ST2500 in this situation, run Stelcomm and wait for it to time out trying to connect with the unit. Then cycle the power on the ST2500, and immediately press the "Reconnect" button in Stelcomm. The program should then be able to access the ST2500.

Note that the ST2500 only seems to stay in protocol mode, if accessed this way, for a couple of minutes. To keep the unit available to Stelcomm, set it into protocol mode. Don't forget to reset it to byte mode before deploying with the Driftsonde.

5.2.2 Configuration for Driftsonde

The following parameters are configured in the ST2500 for operation in the Driftsonde. The "tab" column refers to the tab selected in the Stelcomm "Parameters" window:

Tab	Value	Parameter	Effect
Unit Unit	60	Messages time out: Originated	Outgoing messages are aged off of the ST2500 if not sent within this many minutes.
	Checked	Byte Mode	ST2500 will operate in byte mode.
	0	Message time out: Originated	
	1	Message time out: Terminated	
	65536	Originate buffer	Buffer size for outgoing messages
	65536	Terminate buffer	Buffer size for incoming messages
	Checked	Last message sent first	Message queue treated as last in, first out.
Misc.	9600	Main Serial baud rate	The serial connection will operate at 9600 baud in byte mode.
Byte Mode	Bmode trigger	Initiated by character	Start and end of message (originated and terminated) is signified by timeout on received characters
	Bmode length	2000	Don't use length as the end of message indicator
	Bmode timeout	10	Message is created after no characters have been received for this many seconds
	Bmode message typs	Originated message	As opposed to REPORT or Globalgram
	Return byte	5	Don't know where this comes in; it may be required in order for outgoing messages to be recognized bt the ST2500
	Tx SOM[5]	60	Start of message indicator for terminated messages. This character ("<") is prefixed to terminated messages before being passed to the DTE
	Tx ROM	62	">" character appended to terminated messages, to signal the end of the message.
	Rx SOM	2	Sent by the DTE to the ST2500 to indicate the start of an originated message.
	Rx EOM	3	Sent by the DTE to the ST2500 to indicate the end of an originated message.

[5] These designations are from the perspective of the ST2500; i.e. Tx indicates that the unit is transmitting to the DTE, and Rx is for messages received from the DTE.

6 Driftsonde PIC Software

6.1 Can-portal

6.2 Sonde-ballast

6.3 Gps

6.3.1 Garmin GPS 35-LVC

The unit contains battery backed non-volatile storage, which retains the active configuration. The battery is recharged during normal operations; it's not stated how long the battery backup will last.

The factory defaults set the unit at 4800 baud, with a bunch of output messages enabled. The unit can be configured by sending NEMA messages to it via a terminal emulator. It appeared that the GPS 35 would not recognize commands unless the serial line was set to 8 data bits and two stop bits. The command set is described in the "*GPS 35 LP TracPak GPS SMART ANTENNA TECHNICAL SPECIFICATIONS*" manual.

For Driftsonde usage, configure the baud rate to 9600:

`$PGRMC,,,,,,,,,,4` (that's 10 commas)

Power cycle the unit after sending this command in order for it to take effect.

Disable all output messages, and then enable the PGRMV and GPRMC output messages:

```
$PGRMO,,2
$PGRMO,PGRMV,1
$PGRMO,GPRMC,1
$PGRMO,GPGGA,1
```

When configured to print $GPGGA, $GPRMC and $PGRMV, the unit will occasionally skip an output reading. I don't know if the number of output messages affects this.

6.4 Aux-serial

6.5 Sensorboard

This PIC controls the activities of the Driftsonde sensor board (see Figure 2, page 12). This processor integrates the following sensors and busses, placing the data on the CAN bus:

- The rs-232 output from the RS90 PIC.
- I^2C bus number 1, which is available on both the sensorboard and the motherboard. One LM92 temperature sensor on each borad is on this bus.
- I^2C bus number 2, which is available on the sensorboard. This bus connects to 4 sets of pads which can be used for off board I^2C needs.
- 6 analog channels. Three of these are dedicated to sensing the 6V battery supply, the 5V power supply, and a test point. The other three are available via pads for off board measurements.
- An rs-232 (receive only) data stream coming from the "LINX" wireless module. The data stream is currently used for a remote gas temperature sensor.

6.5.1 Sensorboard PIC flash programming

Note that the RS90 Vaisala sensor module uses pin B6 on the RS90 PIC. This sometimes fouls up the programming of the PICS on the sensorboard. To program the sensorboard PIC, you might need to remove the Vaisala sensor module. The RS90 PIC must also be put into reset (switch 2 on the EPIC box). Even then, it can take many tries before the sensorboard PIC will accept programming.

6.5.2 Firmware design

Due to the number and variety of peripherals connected to this PIC, the firmware is more complex than most of the others in the system.

In particular, the need to interface with two RS-232 streams, while retaining real-time response to the CAN bus, made the development challenging. Implementing a receive only software UART, based on interrupts generated by the PIC timer1, solved this

problem. The timer interrupts at three times the nominal baud rate. A state transition machine is executed to control start bit detection and bit sampling. Overrun and frame errors are detected. The design of the state machine comes from the excellent work of Bob Ammerman[6]. Ammerman's original design cleverly uses vertical logic in order to receive 8 serial data streams, of the same baud rate, simultaneously. The scheme was modified here to sample only one stream. An extra benefit is that arbitrary sample line inversion is trivially implemented if needed.

Through fine tuning of the algorithm, it was possible to get the CPU usage of the software UART down to 4% for 1200 baud operation. Higher baud rates would proportionally increase the system load.

The software UART continuously samples the incoming RS-232 data stream, storing the characters in a receive buffer until a newline is found. A buffer full flag is set, and following characters are ignored until the main loop processing clears the buffer full flag. There are several benefits to this scheme, the first being that the PIC is never blocked while waiting for characters on the incoming stream. Additionally, by stopping character collection after the newline is received, the main loop has all of the (assumed) dead time following the newline to get around to processing the available buffer, which can eliminate the need for a double buffer implementation. This is assuming that there will be an inactive period at the end of each line of the data stream arriving on the software UART port.

The second rs-232 data stream is also processed under interrupt control, this time using the PIC's hardware UART.

The main loop of the firmware follows the standard Driftsonde scheme, where a counter is incremented continuously. During each pass through the loop, the CAN system is polled for service, receiving messages or sending buffered characters as needed. When an initial limit on the loop counter is reached, the LED heartbeat is turned on. A few counts later, the LED is turned off, and task processing begins.

A sub-counter is used during task processing, so that one of a number of tasks is performed in succession.

[6]http://www.piclist.com/techref/microchip/16F84-rs232-8ch-ba.htm

The tasks all format a measurement and send it to the system controller via the CAN bus. The tasks are:

- Sample the sensors on the two I2c busses and format into degC. If a sensor is out of range (±100 ºC), an empty field is transmitted.
- Sample the analog inputs, and format as voltages. Note that the first three analog inputs are from the output of analog switches, which are controlled by RB1:3. The switches are used to turn off voltage divider circuits which otherwise would represent a needless waste of current.
- Send the available RS90 data string. Superfluous characters are remove form the string, and blanks are replaced by commas.
- Send the available gas temperature string. This string is sent without modification, since it will be subject to corruption from the RF link, and the system controller can use the checksum to detect this.

Each message sent to the system controller begins with a character identifier, and is terminated by a newline:

Table 1. Sensorboard PIC Output Messages

Sensors	Fields	Sample Message
RS90	Pressure (mb), T (ºC), RH1(%), RH2(%), Internal Temperature (ºC)	R845.91,23.05,11.38,11.60,24.73
Gas Temp.		G$7.41 -52.04 24.87*EB 29078
Analog Voltages	SW+6V(V), MB+5V(V), TP1 (V)	A1.84,1.64,0.00
I^2C sensors	Motherboard T(ºC), Sensorboard T(ºC)	T26.8,27.0,24.6

6.6 RS90

7 Appendix A: Installation Checklist

Table 2. Linux Software Installation Checklist

Linux	
	/etc/hosts
	/etc/fstab
	Mkdir /net/dyn80-21
	/etc/rc.d/rc.sysint (edit IP name)
	/etc/rc.d/init.d/driftsonde
	/etc/rc.d/init.d/crond
	/etc/rc.d/init.d/sshd
	/etc/cron.d/icarus
	/usr/sbin/crond (from vixie-cron/cron) (chmod u+s)
	/usr/bin/crontab (from vixie-cron/cron) (chmod u+s)
	/etc/rc.d/init.d/sshd
	~icarus/.profile (from home/bash_profile)
	~icarus/microdb
	~icarus/orbcomm
	~icarus/gps (chown root; chmod u+s)
	~icarus/multiplex
	~icarus/canTalk
	~icarus/timeup
	~icarus/AspenQC
	~icarus/class.data
	~icarus/er2.data

Linux	
	/etc/rc.d/rcd.3/S90sshd -> ../init.d/sshd
	/etc/rc.d/rcd.3/S91crond -> ../init.d/crond
	/etc/rc.d/rcd.3/S92driftsonde -> ../init.d/driftsonde

Table 3. PIC Firmware Installation Checklist

PIC	
	Motherboard JMP #1 (EPIC Box #1): gps.hex
	Motherboard JMP #2 (EPIC Box #1): can-portal
	Motherboard JMP #3 (EPIC Box #1): aonde-ballast
	Motherboard JMP #4 (EPIC Box #1): aux-serial
	Sensorboard JMP #1 (EPIC Box #3): sensorboard.hex
	Sensorboard JMP #2 (EPIC Box #2): new freq_meas20mhz.hex
	Gas temperature Probe: proprietary (requires special cable)

Table 4. Stellar ST2500 Configuration Checklist

Stellar	
	Set main serial baud rate to 9600
	Message timeout: 120 minutes
	Mode: Byte Mode
	Bmode trigger: initiated by character
	Rx SOM:
	Rx EOM:
	Tx SOM:
	Tx EOM:

Stellar	

Table 5. Garmin GPS 35 Configuration Checklist

Garmin GPS 35	
	Set baud rate to 9600
	Disable all messages
	Enable $GPRMC and $PGRMV

8 Appendix B: Temperature Testing

During the course of recent development of the Driftsonde, ad-hoc temperature performance tests have been carried out. This has been done by simply running the Driftsonde electronics in the environmental chamber, lowering the temperature, and verifying that the system is still running correctly. This has been tried on the three prototype systems (#'s 1, 2 and 3), although without recording the individual results for each system.

To date (17 Feb 2002), the electronics subjected to the tests included the motherboard, sensorboard and Garmin 25 GPS module. The orbcomm and gas temperature units have not been tested. A gas temperature sensorboard was transmitting outside of the chamber and being received correctly by the sensorboard. Likewise, a GPS signal reradiated from outside of the chamber, was received correctly by the Garmin unit, with navigation lock being obtained.

The general result is that if the systems are started at room temperature, they will operate successfully down to –40 ºC. at least one of the systems has been chilled to these temperatures and run without problem for up to 24 hours. Others have been operated this for somewhat less than 24 hours.

A test made on 16-17 February 2002, with unit #2, involved operating the system for several hours at –40 ºC, and then lowering the temperature to –60 ºC. The system failed shortly after reaching the minimum temperature. The test was far from rigorous, since the system was operating unattended, and not directly observed during the procedure.

An analysis of the failure follows, with time 0 being the point at which the temperature was lowered. The chronology was deduced from the *report.log* and *orbcomm.log* files, which were preserved throughout the test.

Time	Event
0	Chamber temperature set-point changed to –50 ºC
5m	Temperature –50 ºC. All systems functional
25m	All systems functional. Chamber temperature set-point changed to –60 ºC
30m	All systems functional. System appears to have been rebooted, according to *orbcomm.log*. Not clear why it would reboot, unless the power was intentionally cycled.
34m	Temperature –60 ºC. All systems functional.
36m	GPS looses navigation lock
78m	Data from sensorboard and GPS lost. (suspect PIC failure, see below)
+14h	CerfBoard is still operational, but with problems. GPS, sonde-ballast PIC and sensorboard PICs are recycling continuously, probably due to watchdog timeouts. End of test.

In summary, the scenario appears to be as follows.

As the temperature was lowered, the GPS module was the first device to be impacted, as it lost navigation lock. It was still broadcasting its data, and the time was still incrementing from the internal clock.

During this period, the CerfBoard rebooted, for unknown reasons. It is possible that after this reboot, the Ethernet MAC was not functional.

At the lowest temperature, the CerfBoard stops receiving data from the GPS and sensorboard CAN nodes. It is likely that these two PICS, and the sonde-ballast PIC were now in the cycling reset mode discovered at the end of the test. It is also likely that the watchdog timers caused the resets. The PICs would boot up, immediately hang, and be rebooted by the timer. The main issue is what was causing these PICs to hang? Perhaps there are parts of the code that were caught in a loop, waiting for a response from an external device, such as the CAN bus. But then why did these three PICs fail, while the can-portal and aux-serial PICs operated normally? At the end of the test, these latter two PICs could be accessed normally, and would respond to commands.

At +14h the CerfBoard is still operating, but in a crippled mode. Access was only available through the console device. "*Ifconfig eth0*" showed the MAC address to be ff:ff:ff:ff:ff:ff:ff:ff.

After the test, the electronics were removed from the chamber and allowed to warm up to room temperature. It took approximately 30 minutes for the three ailing PICS to recover, which seemed well after the point at which they would have reached operating temperature. During this time, they exhibited the recycling boot behavior. Eventually all parts came back to normal operation.

9 Appendix C: CerfBoard Linux V3.1

Here are a few notes about the Intrinsyc Linux distribution, version 3.1

The distribution provides a complete source tree and precompiled binaries, as well as the arm cross compiler tools.

The stock binaries, for the kernel and the root file system, can be used directly from the distribution. Alternatively, these can be regenerated using the arm tools. The cross tools must be built from the distribution; they are not distributed in binary form.

It appears best to install the Intrinsyc distribution in the default location; i.e. below $(HOME), since the build scripts appear to depend on this. Two directories are installed from the distribution CD:

~/intrinsyc-linux – the linux distribution, for all Intrinsyc products: cerfcube, cerfpod and cerfpda.

~/intrinsyc-toolchain – the source distribution for the arm tool chain.

9.1 The ARM cross tool chain

The cross tool chain can be compiled according to instructions in the documentation directory. The tools will be installed in *~/Intrinsyc*. Paths for the account should be amended to located the cross tools. Note that the cross tools have prefixes that designate the arm architecture, such as arm-linux-gcc, arm-linux-strip, etc.

Makefiles for our applications should provide a mechanism to choose the correct tools, if they are going allow builds for both arm and x86 targets.

9.2 The Cerf Linux O/S

The *~/intrinsyc-linux/base* directory contains the operating system sources, source packages, and binary distribution. There are three primary directories below this one:

~/intrinsyc-linux/base/src – contains the source code for Linux and applications, and scripts and makefiles for building them for the Cerf products. Builds will typically be initiated by doing an Intrinsyc specific make out of a sub-directory below this directory.

~/intrinsyc-linux/base/build – this directory is used for the artifacts of the builds. Source trees are copied here, patches applied, and compiles made in these directories. The linux-2.4.9/linux/ subdirectory is equivalent to a standard linux source directory (/usr/src/linux) when building the kernel.

~/intrinsyc-linux/base/target – packaged results of the builds are placed in this tree. The Instrinsyc makes, taking place out of *~/intrinsyc-linux/base/src*, will use the ~/intrinsyc-linux/build directory for compilation and linking, and then the output products are collected and placed in the *~/intrinsyc-linux/base/target* directory tree. For instance, building the kernel results in the creation of *~/intrinsyc-linux/base/target/cerfcube/packages/linux-2.4.9/linux-2.4.9_noHUD_32.tgz.* This tarfile contains the zImage kernel file, and the modules which will be installed in */lib* on the cerfcube.

9.3 Modifying and building the kernel

Most of the commands for building the kernel are run from the src sub-tree of the distribution:

```
cd ~/intrinsyc-linux/base/src/linux-
2.4.9/linux
```

First the existing build directory is cleaned, the sources copied there,patches applied, and configured. Note that locally made changes to the source code, in the build tree, will be over written during this step:

```
make clean prepare config TARGET=cerfcube
```

At this point, the kernel is ready to build, with the stock configuration provided by Intrinsyc. If the kernel needs to be configured to add extra capabilities, this can be done by doing a *make xconfig* from the *~/intrinsyc-linux/base/build/linu-2.4.9/linux* directory, before proceeding to the next step.

Now the kernel is compiled, and packaged:

```
make compile install TARGET=cerfcube
```

As mentioned above, this creates a tar file containing the zImage and modules: *~/intrinsyc-linux/base/target/cerfcube/packages/linux-2.4.9/linux-2.4.9_noHUD_32.tgz*.

9.4 Configuring and building the root file system

How is this built? It results in the root jffs2 image, which will contain the kernel modules. Thus, to make kernel modifications permanent, we need to ultimately create the root file system, so that the flashing step (below) will bring these along automatically. During kernel experimentation however, the updated modules can be gotten out of the *linux-2.4.9_noHUD_32.tgz* file, ftp'ed to the CerfCube, and installed for testing.

9.5 Preparing for download

The CerfBoard bootloader uses tftp for downloading the kernel and root file system to the flash. Three objects are required in the /tftpboot directory:

loadlinux-cube – this is a script that the bootloader will run, to download and flash the kernel and root file system. Typically we have modified this script to make it a little more amenable to kernel development, replacing the verbose file names with shorter ones, which are links to actual files of choice. This makes it easier to experiment with different kernels, etc. Look in this file to determine which files from tftpboot are being loaded into the CerfBoard.

zImage – this is the kernel image. It will be written to the section of the flash containing the kernel (above the boot loader). When the CerfBoard boots up, it is uncompressed from flash into RAM, and run from there.

root file system – this is an image of the root file system, in jffs2 format. It will be flashed into the area of the flash reserved for the root file system. This does not need to be compressed, since the root file system will be operating out of the flash.

9.6 Hints

The CerfBoard boots up with a fixed IP address of 192.168.1.100. to configure it to use DHCP to get an IP address, add a link in */usr/sbin*:

```
ld /usr/sbin
ln –s /sbin/ifconfig ifconfig
```

This seems a curious method for configuring this. The situation is that the */etc/rc.d/init/network* script will run dhcp if it finds */usr/sbin/ifconfig*, which is not present in the default installation. Perhaps this is an error in the distribution?

10 Appendix D: CerfBoard Linux V2.4

The information in this section applies to the Intrinsyc V2.4 distribution. Most of this has been invalidated by the upgrade to Intrinsyc V3.1.

The CerfBoard comes with a rudimentary software manual, titled "CerfCube™ for Linux, Version 2.4.1". Much of the information given in this section is found in the manual, as well as more detailed descriptions of some aspects.

The software development kit distributed by Intrinsyc contains at least the following (and probably more):

- The linux kernel sources, which can be used to reconfigure and build a new kernel for the CerfBoard.
- An arm cross tool chain based on GNU tools. These are installed in /usr/bin, as arm-linux-gcc, arm-linux-g++, etc.
- A collection of utility packages, such as ssh, apache, etc., which can be built for arm-linux. I believe that some, if not all, of these packages also are distributed in a binary form.
- A set of scripts and directories that are used to assemble all of the pieces necessary to recreate the static images required at boot time on the CerfBoard. Ultimately, to reconfigure the default operation of the system, these images are recreated and downloaded into the CerfBoard flash.

In changing the system software on the CerfBoard, we may in general be doing three things: adding a new package (e.g. ssh) to the system software, modifying the root or usr file systems (e.g. changing */etc/passwd*), or rebuilding the kernel. These activities are described in the next three sections, which explain the activities to be carried out on the host system. After everything has been prepared on the host system, the changes must be installed on the CerfBoard, as explained in a latter section.

The head of the Intrinsyc source tree is */usr/intrinsyc-linux*. A large collection of support scripts is found in the *QUICKIES* sub-directory. These scripts expect to be executed from */usr/intrinsyc-linux*. Be sure to change to this directory before running the scripts.

Note that the directory structure below */usr/intrinsyc-linux* is very confusing. There seem to be a lot of artifacts that are meant to support other hardware and software configurations. You just have to grit your teeth and live with the tangle of directories and files.

Table 6. Sub-directories of /usr/intrinsyc-linux

Important sub-directories of */usr/intrinsyc-linux*	
QUICKIES	
PACAKGES	
PACAKGES/static	
CONFIG	

10.1 Adding a new package

The Intrinsyc manual gives instructions for adding additional packages to the system. The procedure involves cross compiling the package, and then copying a compressed tar file to the *RELEASE/arm/cerfcube* directory. An entry is added to the list of desired packages, in *CONFIG/cerfcube-utilities-cross*. The ram disk creation scripts will then include the package in the build of the ram disk distribution.

The source directories for each package exist immediately below */usr/intrinsyc-linux.* The package builds appear to take place in these directories.

Be sure to read the instructions in section 4 of the Intrinsyc manual for creating new packages.

10.2 Modifying / and /usr

PACKAGES/static contains a number of files; the most important ones seem to be *dev-0.0.1.tgz* and *cerfcube-etc-0.0.1.tgz*. The former contains the */dev* entries for the CerfBoard, while the latter contains files destined for */etc*, as well as couple of files for */usr/bin* and */bin*[7].

[7] Note the misnomer in the tar file name "..*etc*..".

To modify these files, first create a scratch directory, and then unpack the tar file in it:

```
mkdir scratch
cd scratch
tar –xzvf ../cerfcube-etc-0.0.1.tgz
```

Now you can modify the existing files, or add new ones. You can also add directories here as well, even at the / level. This can be useful for creating mount directories that are at root level.

Once the modifications are complete, overwrite the old tar file with a replacement:

```
tar –czvf ../cerfcube-etc-0.0.1.tgz *
```

10.3 Packaging the file systems

Once the new packages have been installed, and */* and */usr* have been modified, these sources need to be combined into file system images that can be transferred to the CerfBoard flash memory. An image will be created for each of */* and */usr*. Recall that the / image will be compressed once it is created, since it ultimately will be uncompressed from the flash and mounted in a ram disk on the CerfBoard.

The following procedure cleans up artifacts left over from previous packaging, creates temporary file systems mounted as loop back devices, populates these file systems, and then copies them to files as raw images.

All of the following takes place from within */usr/intrinsyc-linux*.

First, cleanup from your past sins:

```
rm –rf tmp
QUICKIES/gen-cleanup_ramdisks.sh
```

Now create the filesystems, populate them and save the images in files.

```
QUICKIES/ix86-create_cerfcube.sh
QUICKIES/gen-create_ramdisks.sh
```

The result of these machinations is the creation of usrdisk and rootdisk.gz. These are copied to /tftpdir, where they will be available for downloading to the CerfBoard:

```
cp usrdisk rootdisk.gz /tftpdir
```

Go to section 10.5, to install the new file system images on the CerfBoard.

10.4 Rebuilding the kernel

Haven't tried this yet; Intrinsyc gives instructions in their manual.

10.5 Installing on the CerfBoard

Once the newly updated file systems and/or kernel are available in the tftp directory, they can be downloaded to the CerfBoard. As mentioned earlier, the CerfBoard boot loader is used for this purpose.

Power on the board, and enter any keystroke to stop the auto boot-up. At this point the appropriate files are downloaded to ram using tftp, written to flash, and then the system is rebooted. Intrinsyc provides a script to do all of these steps (*boot_2.4.0*), and thus directs you to down load and execute that script, as follows:

```
Cerf>memset c0000000 0 1ff
Cerf>tftp boot_2.4.0 c0000000
Cerf>flash 200000 c0000000 1ff
Cerf>exec 20000
```

I had to search around through the Intrinsyc distribution, to find this script. It was in *(/usr/intrinsyc-linux/PACKAGES/static/boot_2.4.0_cube*). It contains these commands:

```
tftp rootdisk.gz c0000000
flash   00200000 c0000000 600000
tftp usrdisk c0000000
flash   00800000 c0000000 800000
tftp zImage_2.4.0_noHUD_32 c0000000
flash   00100000 c0000000 100000
&memcpy c0008000 00100000 100000
&memcpy c0500000 00200000 600000
&boot   c0008000 0 1f
```

You see that the script is down loading and flashing the kernel, and the root and usr file systems. After the flash has been written, the kernel and root ram disk are copied back into ram, and Linux is booted. I have written a few other scripts that can be used to carry out the same process for individual components.

10.6 Specific Configuration Tasks

10.6.1 IP Address Configuration

The IP address can be statically set by editing the ifconfig command found near the end of */etc/rc.d/rc.sysinit*:

```
ifconfig eth0 128.117.80.81 netmask \
255.255.255.0 up
```

It appears that dhcp can also be used to dynamically obtain the IP address. The */etc/rc.d/init.d/network* script will run the dhcp client daemon if it (*/usr/sbin/dhcpcd*) is found. It's not clear what will happen if both methods are tried at the same time; I suppose the ifconfig line should be commented out if dhcp is going to be used.

10.6.2 ATA/IDE Compact Flash

As mentioned earlier, the */etc/rc.d/init.d/disk* script would appear to be useful for mounting accessory disk drives, but it does not work for the compact flash drive in the PCMCIA slot. This is because at the time that the script is executed during the boot procedure, the ide drive has not recognized the CF card. However, the PCMCIA system can be configured to mount the flash as a disk drive whenever the CF card is detected. This is done by configuring */etc/pcmcia/ide.opts*:

```
case "$ADDRESS" in
*,1,*,1)
   INFO="ATA/IDE in socket 1, partition 1"
   DO_FSTAB="n" ; DO_FSCK="n" ; DO_MOUNT="y"
   FSTYPE="ext2"
   OPTS="noatime"
   MOUNTPT="/cf1"
   ;;
*,*,*)
    PARTS="1"
    # Card eject policy options
    NO_CHECK=n
    NO_FUSER=n
    ;;
esac
```

This configures the PCMCIA system to mount partition 1, of the card in slot 1, as an ext2 file system, on mount point */cf1*. With a little bit of studying, the Linux PCMCIA HOWTO is a very good guide for these configuration issues.

The CF card will automatically be mounted on */cf1*. The mount will fail however, if the mount directory does not exist. The best way to permanently create the mount point is to create the directory in the root file system, and have it tarred into the root file system package. Thus, I created the directory in ….PACKAGES*/static/scratch/cf1*, incorporating it into the ram disks.

10.6.3 E2fsprogs

10.6.4 Other tweaks

/dev/null – the default permissions are not world writable. The CerfCube developer said on the mailing list that he had missed this. NEED TO FIND OUT HOW TO CHANGE THIS IN THE DISTRIB.

/sbin/route – change permission to suid root, so that the user can add routing when the ppp link is brought up. There needs to be a better way to do this; maybe something in the *wvdial* package could be accommodated for this. Note that the *wvdial* people say that the main class library could be used to build your own specialized application.

10.6.5 Additions to rc.local

Maybe *wvdial*.conf should just be placed in /etc on the distribution?

```
cp /usr/sbin/wvdial.conf /etc
chmod go-rw /etc/wvdial.conf
chmod a+rw  /dev/tty
chmod a+rw  /dev/null
chmod u+s   /sbin/route
chmod a+r   /var/log/messages
```

10.7 Applications

10.7.1 Pppd

```
Make CC=arm-linux-g++
```

10.7.2 Wvdial

```
Make CC=arm-linux-g++
```

10.7.3 Minicom

Make CC=arm-linux-g++

10.7.4 Libwww

```
./configure --disable-shared arm-linux
```

```
make CC=arm-linux-gcc \
  LN=arm-linux-ln  RANLIB=arm-linux-ranlib \
  NM=arm-linux-nm
```

```
./w3c -get http://128.117.80.208/cgi-
bin/driftsonde.pl -form "data=`date`:`uname
-a`"
```