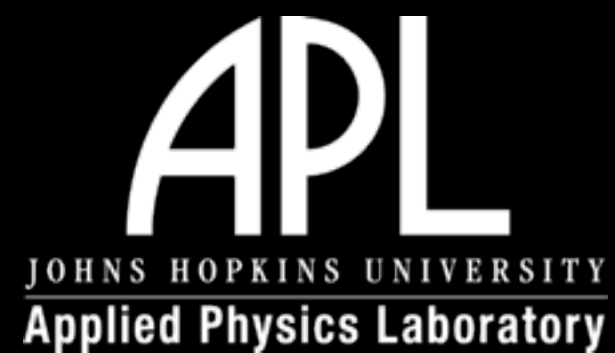


Software Workshop

R.J.Barnes



Workshop Agenda

- Overview
- History of the Radar Operating System (Radops486 to today)
- Software Design
- RST/ROS 1 (QNX4)
- RST/ROS 3 (Linux/QNX6)
- Data Analysis Software

Workshop Agenda

- RST/ROS I : QNX4
 - Architecture
 - Radar Operations
 - Hardware Interfaces
 - Writing Control Programs
 - Data File Formats

Workshop Agenda

- RST/ROS 3: Linux/QNX6
 - Architecture and differences
 - Radar Operations
 - QNX6
 - Linux
 - Writing Control Programs

Workshop Agenda

- Data Analysis Software
 - Architecture
 - Installation and requirements
 - Basic Tools
 - IDL Interfaces
 - Libraries

Overview

- SuperDARN radars have been operated for almost 30 years
- SuperDARN as an official organization has been around for 20 years
- From the PIs agreement:

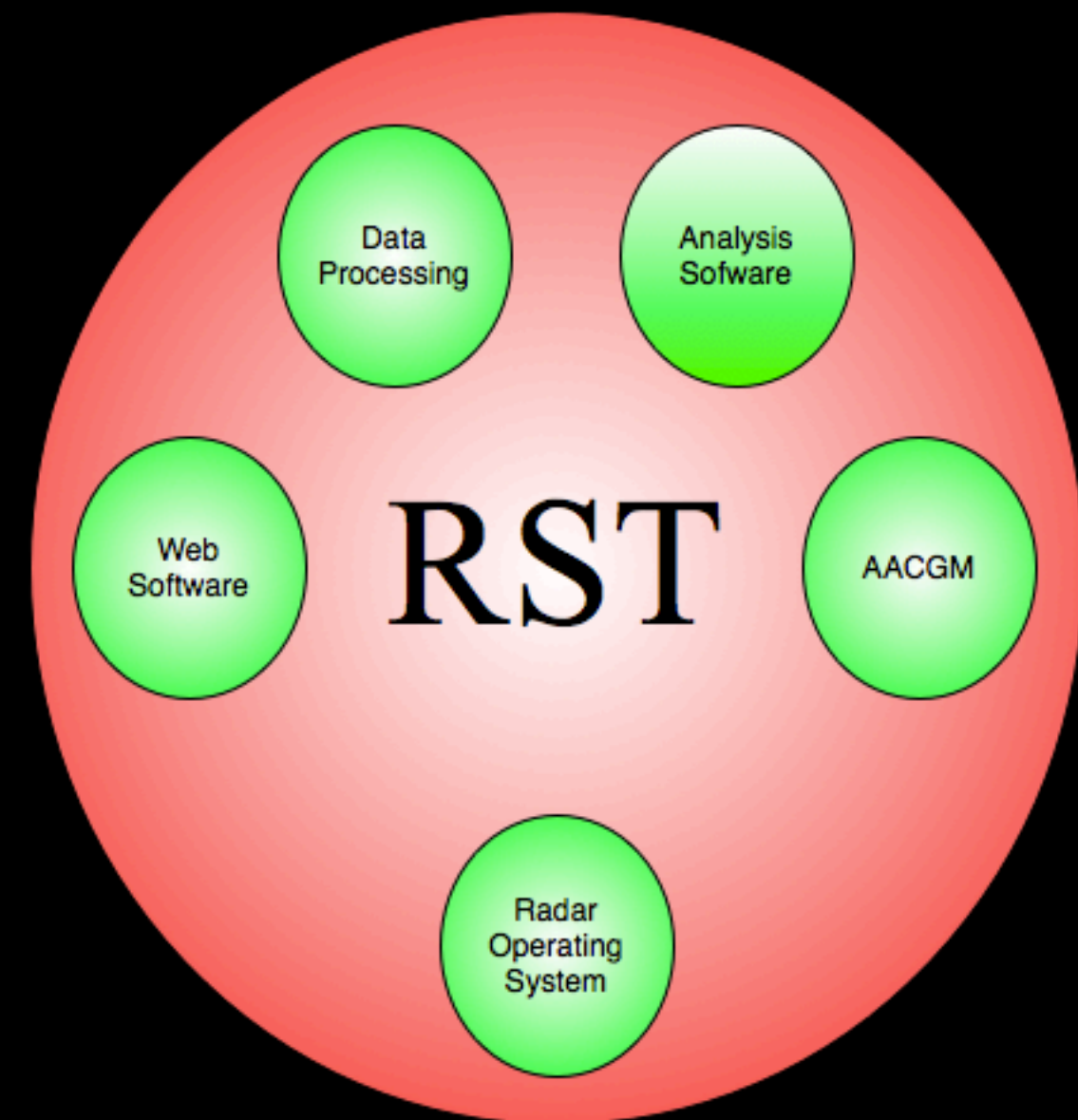
”The PIs agree to share freely amongst themselves all technical information on radar system performance, signal processing techniques, radar control software and analysis software that will lead to improved performance of the SuperDARN radars.”

Overview

- 177,000 Lines of code
- 45.85 Staff years of development
- Estimated development cost \$12.1M

Overview

- What is it called?
 - The overall name for all of the software is the Radar Software Toolkit (RST)
 - The sub-set of RST for operating the radars is the Radar Operating System (ROS)



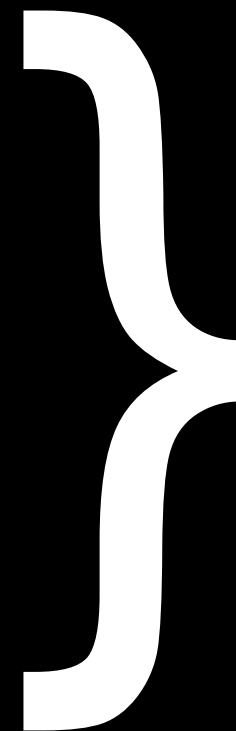
Overview

- This workshop is designed for those who do not have a background in software engineering, but do have a basic knowledge of programming
- It will attempt to explain how the Radar Operating System works and some of the engineering concepts involved
- It will cover both the existing and the new versions of ROS and hopefully a little of the data analysis software

History of The Radar Operating System

History of the Radar Operating System

- Data General PDPI I Radops
- QNX2 Radops
- QNX4 Radops486
- QNX4 Radops 2000
- QNX4 RST 1/2
- Linux/QNX6 RST 3



What I know about

History of the Radar Operating System : The Beginning

- Development of the Radar Operating System goes back to the original Goose Bay radar deployed in 1983
- The project was ahead of its time in using software running on Data General mini-computers to operate the experiment
- Traditionally this kind of experiment relied on a custom hardware solution
- The Goose Bay software solution allowed great flexibility in operating the radar and allowed the radar to run different “programs”

History of the Radar Operating System

- Brief interlude by Kile on the beginnings...

History of the Radar Operating System: Radops 486

- In 1988/89 the decision was made to switch to a PC architecture:
 - PC performance was finally good enough to support the hardware demands
 - PCs were cheaply available
 - ISA interface cards for Digital Signaling and Analogue to Digital Conversion were available and reasonably affordable (~2-3K)
 - PC Operating systems were becoming more capable (QNX2)

History of the Radar Operating System: Radops 486

- System consisted of two PC class computers:
 - 486 class main computer for the main processing
 - 286 class secondary or timing computer to output the pulse sequence
- Initially built on the QNX2 Operating System

Main Computer (486 Class PC)



33.6k Baud
Modem



500MB Magneto-Optical
Disks



dt2828 A/D Card



bc620 GPS Card

Serial Cable

Timing Computer (286 Class PC)



PIO48C DIO Card

History of the Radar Operating System: Radops 486

- Shortly after QNX2 was adopted for the Radars QNX4 became available:
 - True multi-tasking real-time operating system
 - Micro-Kernel Architecture
 - Compact OS kernel that provided low-level services
 - Other OS features, device i/o, file systems, networking, etc provided by servers
 - POSIX/Unix architecture
 - Inter-Process Communication through via messages
 - Sophisticated messaging API
 - Low-overhead
 - Supported messaging over the network for distributed applications

History of the Radar Operating System: Radops 486

- Radops 486 Features:
 - Radar controlled by “radlang” programs run through the “interpreter”
 - radlang had a C like syntax but was an interpreted language with radar specific functions for controlling the radar hardware
 - The interpreter was the core of the Operating System and communicated with the rest of Radops
 - The radlang program could be suspended in the interpreter to allow parameters to be altered by the operator

History of the Radar Operating System: Radops 486

- Radops 486 Features:
 - Hardware interfaces were by custom device drivers for Radops
 - DT2828 A/D ISA card for sampling.
 - DMA transfer to main memory with interrupt on completion
 - 12 bit 4 channel driven by an external trigger pulse
 - PIO48C ISA card for digital I/O
 - Memory mapped 48 channel digital I/O
 - Output used for controlling radar hardware and for pulse sequence
 - 10MHz clock supplied as input for timing purposes

History of the Radar Operating System: Radops 486

- Radops 486 Features:
 - Main computer and timing computer communicated via serial cable
 - Timing came from a BC620 GPS receiver card
 - Custom driver would decode GPS time messages and send them to the interpreter
 - Data was written to 500MB Magneto-Optical Disks
 - Disks had to be written continuously
 - If the disk writing was interrupted, a new write session had to be started
 - Used a custom SCSI interface driver
 - Used own filesystem to deal with power failures/multiple sessions
 - Disks were very expensive ~\$50 each

History of the Radar Operating System: Radops 486

- Radops 486 Features:
 - Timing sequence code was written in assembly language for speed
 - The display was a simple monochrome ASCII output written to a terminal
 - Software was under revision control (RCS) but not versioned

History of the Radar Operating System: Radops 486

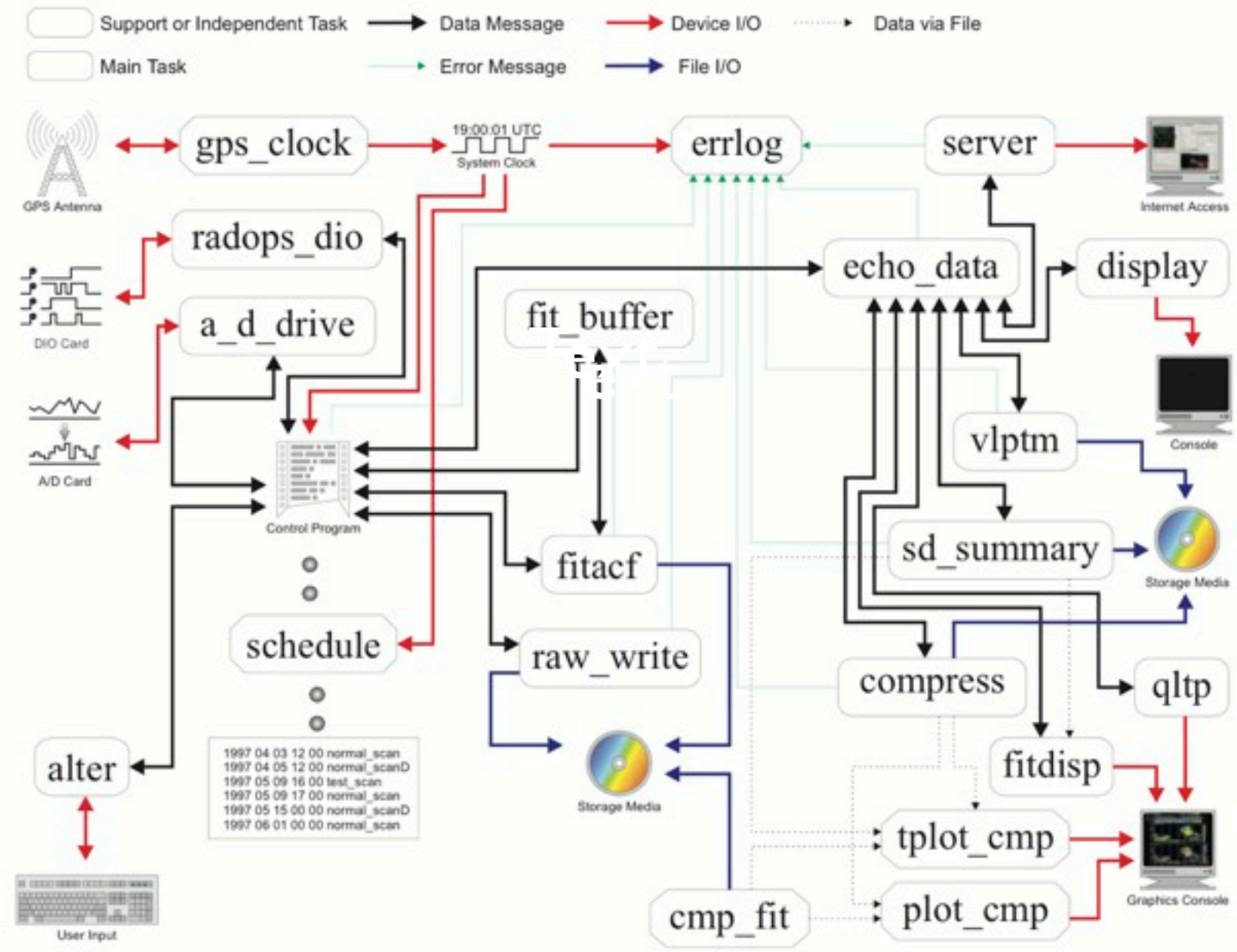
- Changes proposed by BAS/APL/Leicester in 1995:
 - Replace RadLang programs with C “control programs”
 - A C library would provide the interface functions to the radar hardware and allow full use of the rest of the QNX Operating System
 - Replace the serial link between the main computer and the timing computer
 - Use an ethernet (network) connection and QNX4 networking to communicate between the two systems
 - Replace WORM disks with cheaper CDROM burners
 - Drivers and software were available for Linux

History of the Radar Operating System: Radops 486

- Changes proposed by BAS/APL/Leicester in 1995:
 - Replace WORM disks with cheaper CDROM burners
 - Drivers and software were available for Linux
 - Replace assembly language timing code with compiled C code
 - Compiled C code was dis-assembled and compared with the original code
 - Extensive timing tests made sure that there were no problems

- Changes proposed by BAS/APL/Leicester in 1995:
 - Allow dynamic starting/stopping of ancillary radar tasks
 - echo_data was developed at BAS and allowed other software to access the radar data stream without interfering with the operation of the radar.
 - Develop a “scheduler” to control which programs should be run
 - Developed by Leicester and re-written for the new software by APL

Radops 2000



Main Computer (Pentium Class PC)

Linux Computer (Pentium Class PC)



dt2828 A/D Card

Ethernet Connection



PIO48C DIO Card

CDROM



Timing Computer (Pentium Class PC)

33.6k Baud
Modem



History of the Radar Operating System: Radops 2000

- Deployed in Goose Bay during 1996
- Eventually universally adopted across the network by ~1998

History of the Radar Operating System: Real-Time Data

- In 1996 Communication with the Radars was still by analog modem
- Very limited bandwidth and only the terminal display and radar shell provided real-time interaction with the radar
- Linux boxes had been deployed to use with the CDROM burners

History of the Radar Operating System: Real-Time Data

- APL was involved with “Upper Atmospheric Research Collaboratory”

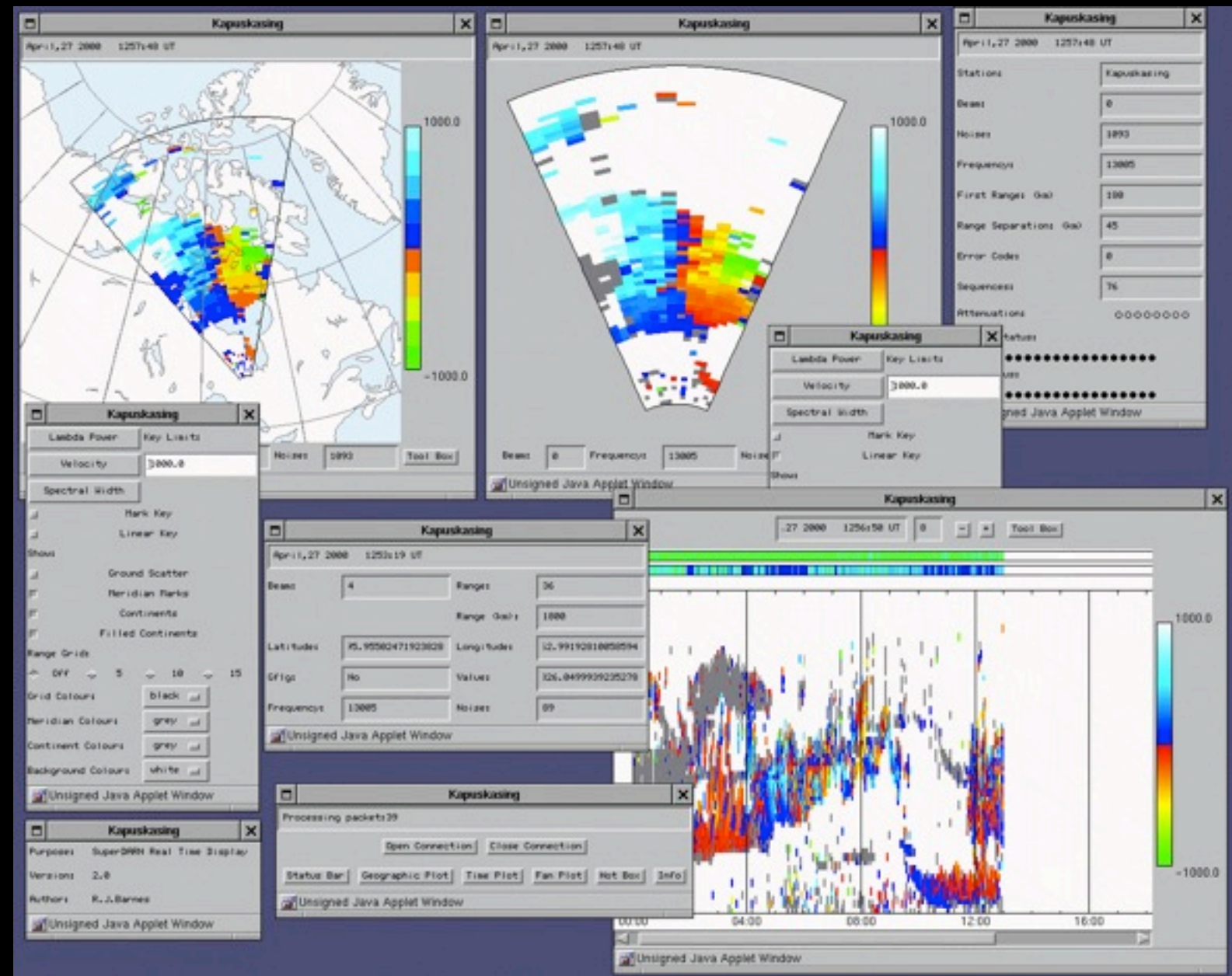
“The Upper Atmospheric Research Collaboratory (UARC) was a joint venture of researchers in upper atmospheric and space physics, computer science, and behavioral science. The UARC project was developed as an international networked collaboration laboratory to allow geographically distributed scientists to collaborate over the data acquisition and analysis. Specifically it provided (1) human-to-human communication and shared computer tools and workplaces, (2) group access and use of a network of data, information, and knowledge sources, and (3) remote access and control of instruments for data acquisition, fabrication of artifacts, and interaction with the physical world.”

History of the Radar Operating System: Real-Time Data

- We experimented with connecting the Radars to the Internet
 - The modems were attached to the Linux box and PPP was used to connect them using TCP/IP
 - Initially grafted onto the APL network, later to a local ISP
 - A TCP/IP Server process was written for QNX that allowed a limited sub-set of FITACF data to be transmitted via TCP/IP packets

History of the Radar Operating System: Real-Time Data

- Java was just becoming widely used and we developed a display applet that could show the radar data in real-time.



History of the Radar Operating System: Real-Time Data

- The Real-Time display used actual digital data, which meant it could also be used for the newly developed global convection patterns
- As costs came down, modems were replaced by high-speed Internet connections
- CDROMS were replaced by direct nightly downloads
- High speed Internet connections are now standard at all Radar Sites

History of the Radar Operating System: Stereo

- The University of Leicester was investigating the possibility of using the unused capacity of the radar system
- By adding a second receiver and phasing matrix it was possible to create a “stereo” radar implementation
- Stereo allows two sets of data to be gathered in parallel
- The STEREO software was developed by APL/University of Leicester

History of the Radar Operating System: RST I

- By 2002 Radops 2000 was installed at most radar sites
- It had a number of limitations from a software engineering point of view

History of the Radar Operating System: RST I

- Prior to RST the SuperDARN data analysis software and Radar Operating System were independent pieces of software
 - The analysis software lacked any packaging or version control
 - Changes in one had to be propagated to the other
 - Changes in file formats, were difficult to implement
 - Changes in algorithms like FITACF were hard to coordinate between the analysis and Operating System software
 - Capabilities of the analysis software were not available to the Operating System
 - The analysis software had access to AACGM, IGRF, other analytical tools

History of the Radar Operating System: RST I

- Radops 2000 still used the original Radops486 data transport scheme
 - FITACF was run during the integration period, so the data was always delayed by one integration
 - Timing tests showed that there was plenty of spare capacity on modern PCs and FITACF could be calculated inline at the end of the integration period
 - Grid files and other summary products could also be generated on site

History of the Radar Operating System: RST I

- Distribution of the code still required archives to be manually created and assigned a version number
- Compiling the code required multiple steps and could easily trip up the novice user

History of the Radar Operating System: RST I

- RST I was the answer to most of these problems:
 - The core analysis software was placed under strict version control
 - Legacy fortran code was replaced with C (including AACGM)
 - IDL interfaces to the new C libraries were written

History of the Radar Operating System: RST 2

- By 2004 RST was well established but was suffering from growing pains
- Libraries were difficult to find and it was hard to identify what they did
- Lots of trivial bugs remained
- Functions had inconsistent or vague names
- The IDL interfaces were still based on the original analysis software
- New radar hardware such as Digital Receivers were starting to appear

History of the Radar Operating System: RST 2

- RST 2 incorporated:
 - A new directory structure according to the software purpose
 - Formal naming convention for library function: FitRead(), FitWrite()
 - New IDL interfaces that mapped to the C libraries
 - Native IDL interfaces
 - Improved packaging and compilation scripts
 - New site libraries and interfaces for handling new hardware
 - New dataMap based format for raw data files
 - Replacing the GPS timekeeping with NTP

Linux Computer



Main Computer



Digital Receiver/AD Card

Broadband Internet
Connection

Ethernet Connection

Internet



PIO48C DIO Card

Timing Computer

History of the Radar Operating System: RST 3

- The rapid expansion of the radar network and the adoption of several new hardware designs started to tax RST2
- QNX4 was becoming increasingly long in the tooth. Device driver support was becoming harder to support
- Discussion on moving from QNX4 to QNX6 or Linux had been underway for several years

History of the Radar Operating System: RST 3

- Lots of experience had been developed at Alaska in working with QNX6
- However there was concern about being locked into another proprietary Operating System
- Both Linux and QNX6 had radically different APIs from QNX4
- It was decided to adopt a hybrid approach

History of the Radar Operating System: RST 3

- The Operating System would be split into two components:
 - A Hardware interface component that would communicate with the radar hardware
 - An operations component that would run the control program and perform the data analysis
- The two components would be independently developed and communicate with each other using OS neutral communication

- QNX6 would be used for the hardware interface
- Linux would be used for the control component
- The interface between the two would be by TCP/IP messages
- The interface would implement a “virtual” radar - any hardware could be configuration could be supported by identifying the key operations of the hardware and implementing them in the interface
- The Operating System would be able to determine the hardware setup by interrogating the hardware component

Software Design

Software Design: Functions of The Radar Operating System

- Schedule operational modes of the radar through control programs
- Control radar hardware to take raw experimental data (I&Q samples)
- Process data through analysis algorithms
 - calculate Auto-Correlation Function, Cross Correlation Function
 - apply FITACF algorithm
- Store data in output files
 - RawACF, IQdat, rawdump data files
- Provide the operator with a real-time display: tdisplay, Java Real-Time Display
- Allow the operator to change radar parameter (radar_shell)

Software Design: Functions of The Control Program

- Generate Timing Sequence
 - Convert pulse pattern to sequence of hardware control bits that control the radar hardware
- Find best frequency to transmit and receive on (Clear Frequency Search)
- Transmit pulse pattern
 - Command timing computer to output the timing sequence

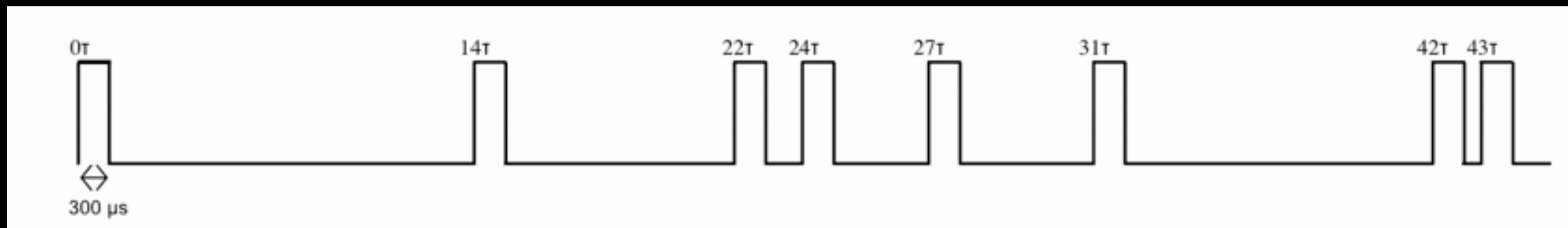
Software Design: Functions of The Control Program

- Take raw I&Q samples
 - Program A/D card or digitizer to sample data and transfer to main memory
- Calculate Auto-Correlation Function (ACF), Cross-Correlation Function (XCF)
- Perform ACF fitting (FITACF)
- Transmit data to other tasks for display, storage
- Check for changes in operating parameters by the user

Software Design: The Timing Sequence

- What is the timing sequence?
 - The timing sequence is a sequence of bytes that are used to control the radar hardware
 - These bytes are used to control the digital IO board which in turn controls the actual radar hardware through the BAS box
 - The timing sequence is generated from the radar pulse pattern and the operating parameters

Software Design: The Timing Sequence



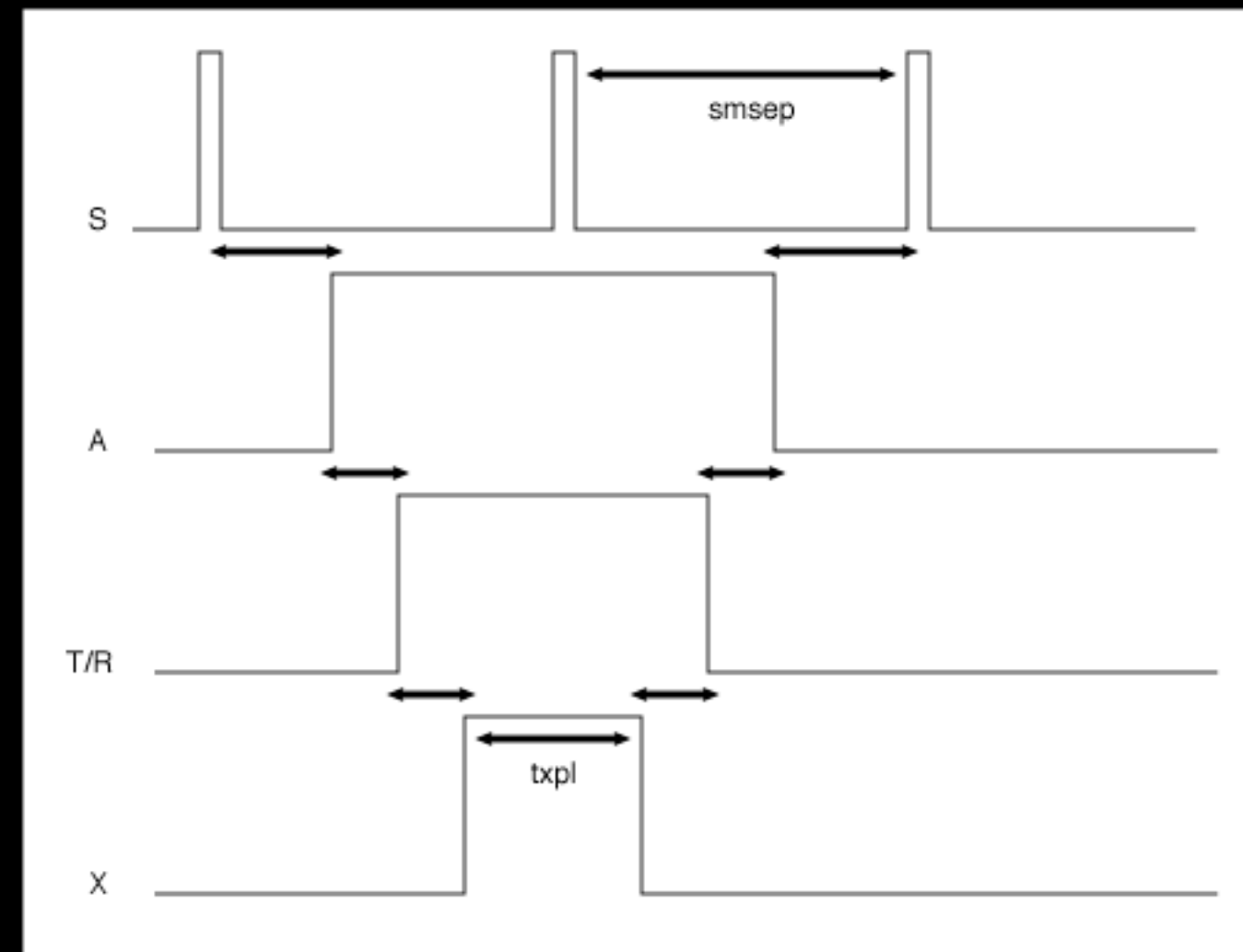
- The pulse pattern is the multi-pulse sequence transmitted by the radar
- It allows the radar to determine both the distance and velocity of targets
- The software must use this pulse pattern to control the radar hardware using four control signals

Software Design: The Timing Sequence

- (S) Sample
 - Sample the received signal (not used for digital receivers)
- (A) All attenuators
 - force all attenuators to on and protect the receiver from spurious signals
- (T/R) Transmit /Receive switch
 - Switches from transmit to receive mode
- (X) Transmit
 - transmits burst of RF signal - the transmitter pulse

Software Design: The Timing Sequence

- The timing sequence is constructed from the pulse pattern using the timing sequence generator
- This is done by a state machine that calculates the necessary hardware control bits based on the radar operating parameters



Software Design: The Timing Sequence

- Timing Sequence parameters:

Parameter	Typical Value	Definition
smsep	300	Sample separation
lagfr	1200	Lag to first range
txpl	300	Length of transmitter pulse
mpinc	1500	Multi-pulse increment
nrang	75	Maximum Range
mppul	8	Number of pulses in pattern

Software Design: The Timing Sequence

- The timing sequence generator is a state machine
 - All possible states of the hardware lines are defined
 - The possible transitions from one state to another are defined
 - The timing parameters are used to compute the actual output bytes using the state transition rules

Software Design: Clear Frequency Determination

- Two forms of clear frequency search are used:
- The original “sampling” search
 - Used on analog receivers
 - Radar cycles through a range of frequencies looking for the lowest noise
- FFT search
 - Used on digital receivers
 - Receiver is switched to a broad frequency band samples are taken
 - An FFT is used to compute the power spectrum and the frequency is selected from this

Software Design: Clear Frequency Determination

- Analog receiver mode:
 - The radar sweeps over a range of frequencies taking a few samples at each
 - The five frequencies with the lowest power are searched for longer to determine the final frequency
 - The noise value is directly determined by this search

Software Design: Clear Frequency Determination

- FFT Clear frequency search
 - Digital receiver is switched to a broad frequency band and samples are taken
 - An FFT is computed to get the power spectrum
 - This is repeated a number of times and the spectrum integrated
 - The lowest power FFT bin is used as the clear frequency
 - Noise is computed from the power in the FFT bin

Software Design: Integration

- Beam selection and Attenuation:
 - The DIO card controls the BAS box which converts the a 4 bit number into one of sixteen beam directions fed to the phasing matrix
 - Attenuators are also controlled via DIO lines and are switched on when the observed lag-zero power exceeds a noise threshold during the integration
 - For digital phasing implementations (like Wallops), 4 additional DIO lines are used instead of the beam bits to get a phase shift - this allows extra beams and steering of the array off the original boresite
 - Other DIO bits are used to select the AGC and Power status information for each antenna

Software Design: Integration

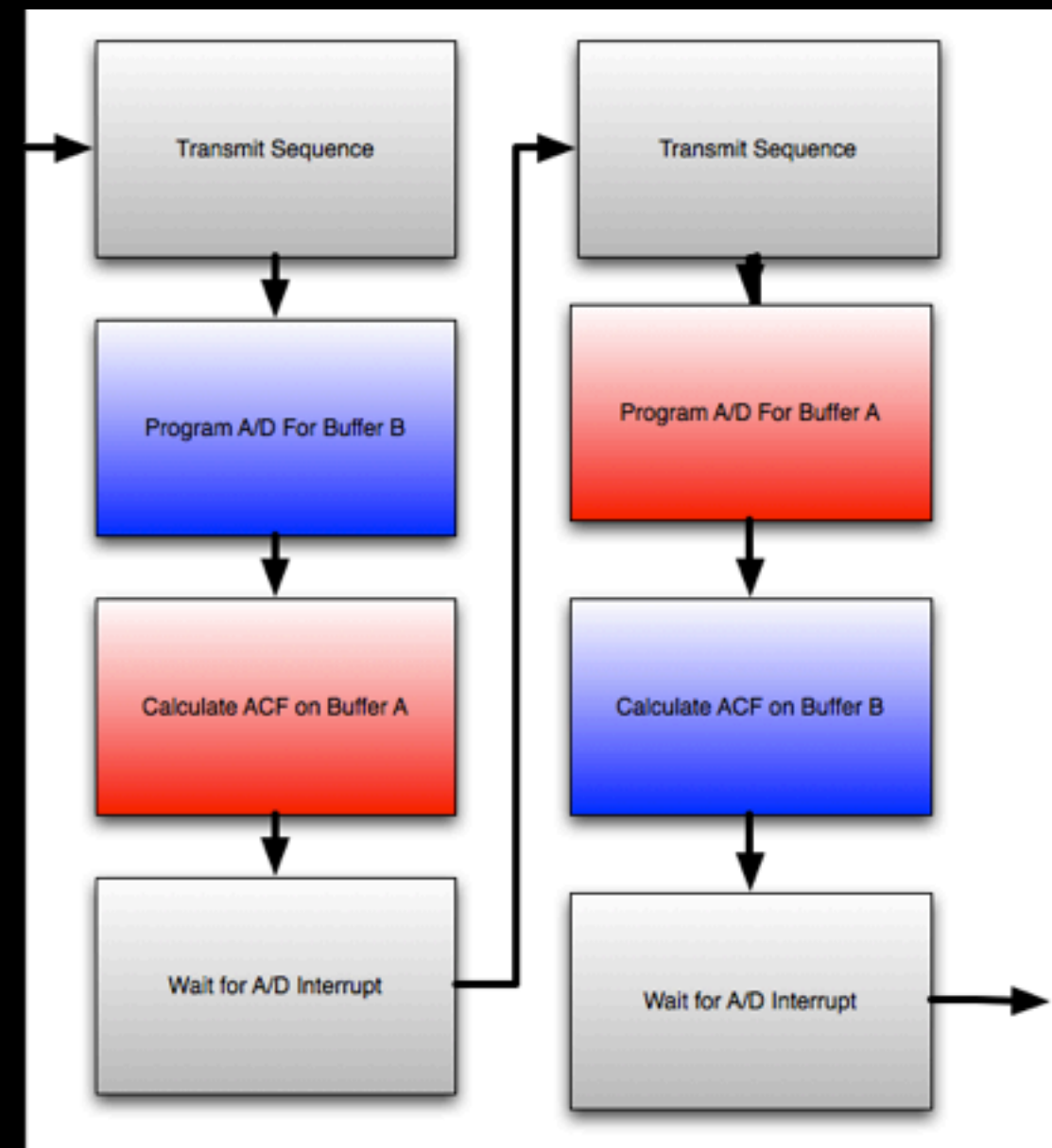
- Timing Sequence output
 - The timing sequence is transmitted by the timing computer
 - Originally a 10MHz external clock was used to drive a counter on the DIO board. This was replaced by a direct crystal oscillator that feeds a TTL input line
 - The change in state of this line is read by the software and used to clock through the timing sequence
 - The timing sequence is extremely time critical and requires that interrupts on the timing computer are turned off
 - This is why we have a separate timing computer - with interrupts turned off, the computer cannot respond to any other stimuli or process anything other than the timing sequence code
 - Originally written in assembly language the timing sequence code is now written in C

Software Design: Integration

- The Integration loop is the heart of a control program:
 - It transmits a pulse sequence,
 - Samples the signal
 - Computes the ACF and XCF

Software Design: Integration

- The integration loop uses double buffering
- While one set of samples are being taken, the ACF and XCF are being computed on the previous set



Software Design: Organization

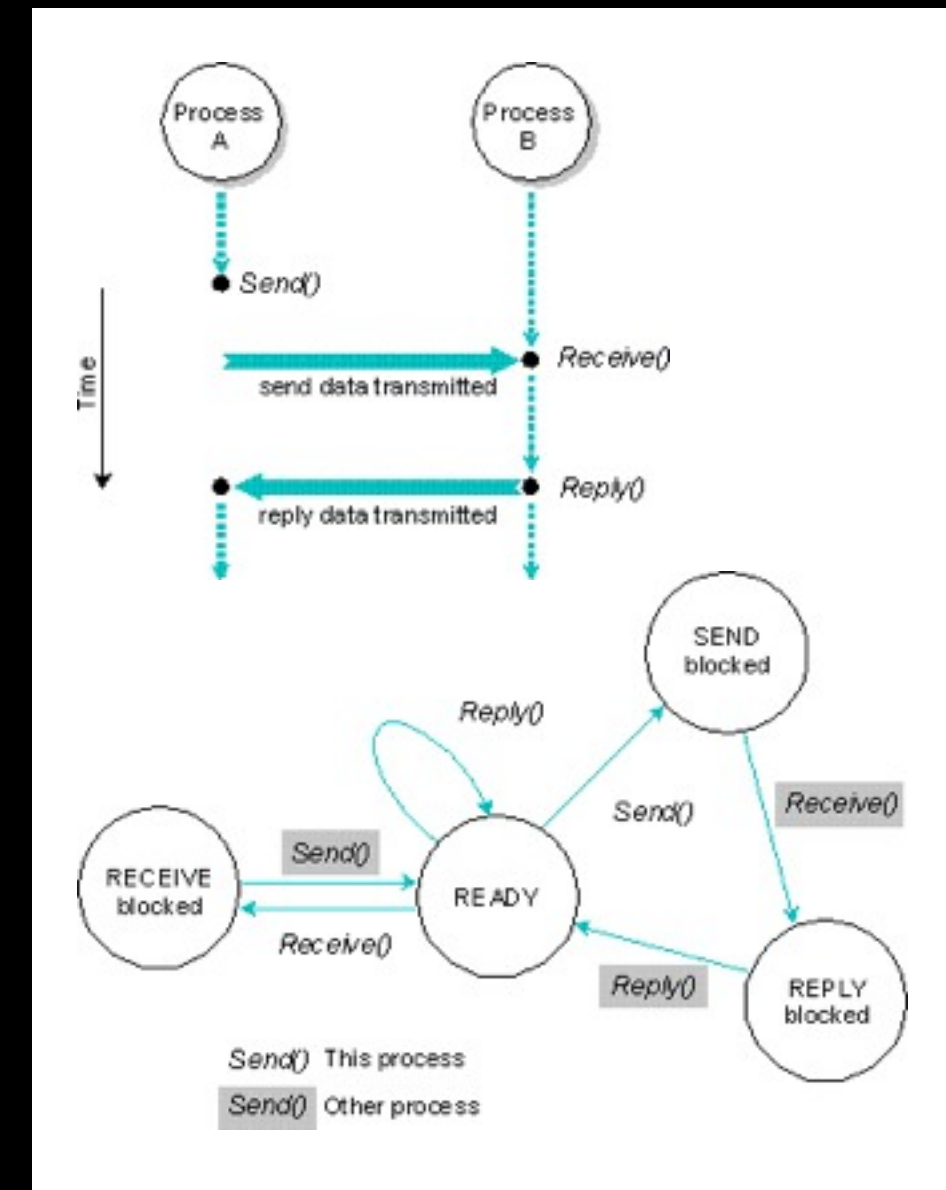
- Key to the Radar Operating System is the distributed design
- The ROS is divided into multiple processes
 - Scheduler
 - Control program
 - Data Loggers
 - Displays
 - Shell
 - Scheduler

Software Design: Inter-Process Communication

- Each process passes information to another using Inter-Process Communication (IPC)
- In QNX, IPC is implemented using message passing
 - Messages are blocks of data that can be sent between processes using QNX Operating System Calls
 - They are extremely efficient and easy to implement
 - QNX Message passing can also be done across a network connection
 - This is used to communicate with the timing computer

Software Design: Inter-Process Communication

- Process A sends a message to process B using `Send()`
- Process A then blocks (suspends)
- Process B receives a message using `Receive()`
- Process B takes action and replies to process A using `Reply()`
- Process A unblocks when it receives the reply from process B
- Processes block on both `Send()` and `Receive()`



RST 1/2

RST I

- The RST I Architecture defined:
 - A filesystem hierarchy (directory structure) for classifying software
 - RCS for revision control
 - Strong versioning of binaries and libraries
 - A set of standard compilation tools

RST I :Architecture

- Filesystem Hierarchy
 - RST I was installed in a top-level home directory
 - It defined several top level directories to classify code

RST I :Architecture

Directory	Description
bin	Binaries for main software
code	Source code
doc	Documentation
lib	Libraries
log	Compilation and installation logs
script	Shell scripts
tables	Data tables and configuration files
usr	Secondary Hierarchy

RST I :Architecture

- Run through of FHS hierarchy document and diagram...

RST I: Version Control

- RST used the Revision Control System (RCS) for version control
 - Each component has an RCS subdirectory that contains the controlled source
 - RCS operates on individual files so each source code module is under revision control
 - Source code is checked out for modification and checked in to mark a revision
 - RCS automatically applies version numbers to revisions

RST I : Version Control

- RST added a “master” version number
 - The version number of each module was stored in the file “version.info”
 - This file was also maintained under RCS and had a master version number
 - The RST utility `rCSV` was used by the makefile to check that the version numbers in each module agreed with “version.info”
 - This prevented code that had not been checked in from being distributed

RST I : Version Control

- Every component has a version number of the form $X.YY$
- eg. `fitacf.2.21`
- X is the major version number
 - Software with the same major version number is functionally the same
 - Binaries operate the same way
 - Library functions have the same calling sequence
- YY is the minor version number
 - Software with different minor version numbers usually represents bug fixes or minor feature enhancements

RST I : Libraries

- RST uses many small libraries that accomplish very specific task
 - eg. time conversion, file I/O, graphics rendering etc.
- Larger “meta” libraries can be constructed by combining the smaller libraries together
- Library function names will always include the library name in them

RST I : Source Code Directories

- Source code in RST I is located in the “code” directory
- This is further sub-divided into “src.lib” for libraries and “src.bin” for binaries
- Each element, whether a library or binary, always has a top level directory that consists of the element name followed by the version number:
`acf.1.12`
- This allows multiple versions of an element to be installed at the same time.

RST I : Source Code Directories

- Libraries are further sub-divided into a “src” and “include” directory
 - “src” contains the C source code
 - “include” contains the header files for the library
 - When compiled symbolic links to the headers in the “include” directory are created in the top-level “include” directory

RST I : Compilation

- Every library or binary has an associated makefile
- This makefile uses a template in “code/make” to compile the code
- Lets look at a makefile...

RST I : Compilation

- Both static and dynamic versions of libraries are compiled
- A compiled library always has the version number appended to the name “libaacgm.1.22.so”
- A Symbolic link is created with just the major version number: “libaacgm.1.so”
- This link points to the most recent version of the library: “libaacgm.1.so -> libaacgm.1.22.so”
- This allows multiple version of the library to coexist. If a binary requires an old version of a library, it can use the fully specified version name.

RST I: Compilation

- The helper utility `makeall` searches a directory tree looking for makefiles and compiling any code that it finds
- `make.lib` invokes `makeall` to compile libraries
- `make.bin` invokes `makeall` to compile binaries
- Any interdependencies are resolved by either ordering the directories or by explicitly telling `makeall` which directories to search in.

RST I : Software Packaging

- Software is packaged using the rpkg scripts
- Code is checked out from the RCS directories in the main repository
- Source code directories have their version numbers appended to them
- The distribution tree is then converted to a self-extracting archive that installs the software on the target system
- The code must still be compiled after the software has been installed

RST I

- Scheduler
- A/D Driver
- Digital Receiver Drivers
- Digital IO Driver
- Data Logging
- Radar Shell
- Real-Time Data
- Libraries
- Site Specific Code
- Walk Through of a Control Program
- Data formats

RST I : Scheduler

`code/src.bin/radarqnx4/main/schedule`

- The scheduler controls which control program is run at any given time
- It is the only process that has higher priority than the control program
- The schedule file contains the start time when a program should be started
- The scheduler spawns a new process when a control program starts
- A running control program sends a message to the scheduler to see if it needs to stop
- If a program does not behave properly, the scheduler will kill it off
- The scheduler periodically reloads the schedule for updates
- If a control program dies, the scheduler will fall back to a “safe” default program

RST I :A/D Driver

`code/src.bin/radarqnx4/drivers/dt2828`

- The A/D driver was originally written for the ISA DT2828 card
- It uses DMA to transfer samples from the card to an area of shared memory.

Text

- An interrupt tells the driver when the DMA transfer has been completed
- The shared memory is accessed by the control program to calculate the ACF, store the raw I&Q samples etc.
- An implementation of the A/D driver for the PCI A/D card was written by Leicester

RST : Digital Receiver Drivers

```
code/src.bin/radarqnx4/drivers/gc214TShf
code/src.bin/radarqnx4/drivers/gc214TSif
code/src.bin/radarqnx4/drivers/gc214hf
code/src.bin/radarqnx4/drivers/gc214if
```

- Four different digital receiver drivers exist for two different cards (GC214, GC214TS) and two modes of operation (IF,HF)
- The digital receiver driver programs the card to take samples at a given sampling rate at a specific frequency and bandwidth
- The receiver includes two digital filters, and depending on the card a re-sampler and a data frame mechanism
- Digital receivers do not use the sample pulse and are either gated or triggered to start taking samples
- Samples are stored in FIFOs and transferred to main memory either by DMA or by simple memory mapped IO
- The output from the receiver is still the equivalent of the I&Q samples taken by the A/D card

RST I : DIO Driver

code/src.bin/radarqnx4/drivers/diopio48c

```

Configuration : Basic Configuration
Registered Name : /dio_drive
Version : 1.14 debug
Sequence No. : 15
Sequence Id. : 0
Sequence Stat. : Ok

Frequency : 10807
Beam : 8
Antenna Mode : AUTO
Synthesizer Status : REMOTE

AGC status : -----
LOPWR status : -----
Attenuation : 000
Test Mode : OFF

```

RST I : DIO driver

- The DIO driver runs on the timing computer
- It communicates with the main computer using QNX4 message passing over the network
- The driver can be run in either a mono or stereo mode, supporting either one or two DIO PIO48C cards
- It contains the timing sequence output code for either mono or stereo mode
- The driver can be configured using a text based configuration file that determines the location and mapping of the DIO ports to the radar hardware control lines

RST I : Data Logging

```
code/src.bin/radarqn4/main/fit_write  
code/src.bin/radarqnx4/main/fitacfwrite  
code/src.bin/radarqnx4/main/raw_write  
code/src.bin/radarqnx4/main/rawacfwrite  
code/src.bin/radarqnx4/main/iqwrite
```

- The data loggers all function in a similar fashion
- They receive a series of IO control messages from the control program to open files, write data and close files
- All use the `fio` library to construct a standard format superdarn filename: “YYYYMMDD.HHMM.SS.stid.xxx”

RST I : Data Loggers

- The control program creates a series of message blocks that contain the radar parameter block, ACF data, FITACF data and IQ data
- The message passing library sends this message block to all the processes registered to receive data
- Each data logger decides what it wants to do with the data contained in the message, store it, ignore it, or further process it

RST I: Data Loggers

- The IQ sample data is handled slightly differently
- As the volume of data is large, it is inefficient to send this as a message.
- Instead the IQ data is stored in an area of shared memory that can be accessed by any registered tasks such as IQwrite
- The IQ data messages contain information about the samples stored in the shared memory, such as the number samples, the interleaving, timing etc.
- IQ write uses this information to store the actual IQ data

RST I : Radar Shell

`code/src.bin/radarqnx/main/radar_shell`

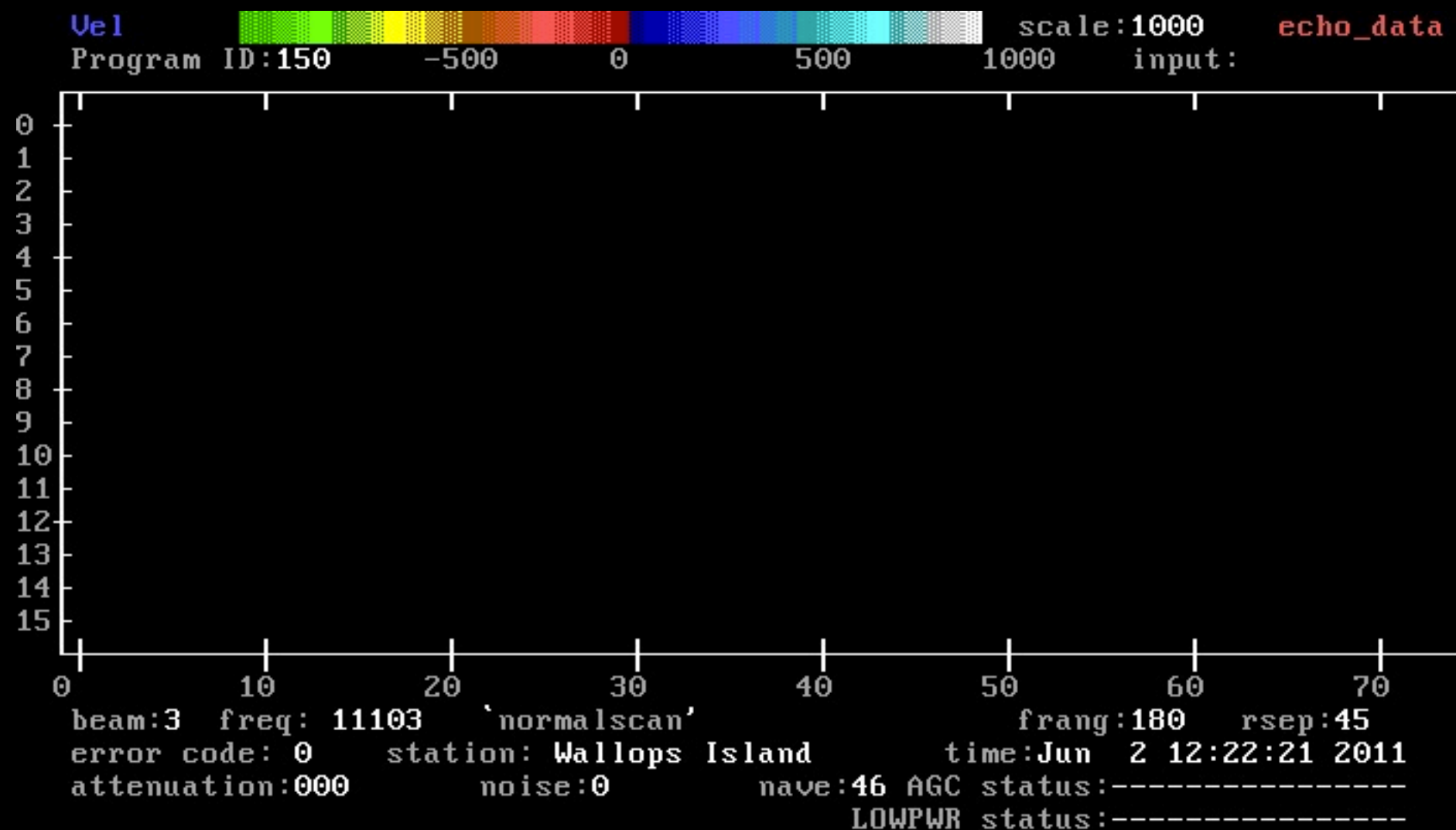
- The radar shell provides a mechanism for the user to interact with the control program
- The shell is designed to allow the user to inspect and change variables without having to stop the control program from running

RST I : Radar Shell

- Every integration loop, the control program polls for the existence of the shell task. If it does not exist, the program continues
- If it does the control program will receive a message either asking for a set of parameters or sending an updated list back
- The radar shell library constructs a message containing a set of parameters and sends them to the shell
- The shell can then operate on these internally without interfering with the control program
- When the user types “go” in the shell, the updated parameters are sent back to the control program and updated

RST I : tdisplay

code/src.bin/radarqnx4/displays/tdisplay



RST I : `tdisplay/echo_data`

- `tdisplay` is a terminal based shell program
- it runs as part of the `echo_data` data relay mechanism
- `echo_data` allows programs to register to receive radar data without interfering with the operation of the control program
- This allows `tdisplay` to be started and stopped as needed.
- `tdisplay` can display a simple schematic of FITACF data or ACF data
- The latest version supports >75 range gates

RST I : Real-Time System

`code/src.bin/radarqnx4/main/rtserver`

- The real-time system is implemented using the `rtserver` task
- It packages FITACF data into packets that are served over TCP/IP
- The process listens for connections on port 1024 and spawns child processes to handle incoming connections
- A pipe is established between the parent and child process to relay the data packets
 - The parent receives data messages from the control program, converts them into TCP/IP packets and passes them onto any child processes connected to external clients
 - This provides a mapping from the QNX messaging system to TCP/IP

RST I : Libraries

- The ROS libraries consist of a mix of general purpose libraries, SuperDARN specific libraries and ROS libraries
- The SuperDARN specific libraries define the standard data structures such as the radar parameter block, fit and dat data structures and the file read and write routines
- The ROS libraries define the low level interfaces to the hardware drivers, the message passing between processes and the basic functions of the radar such as the integration loop and the clear frequency search

RST I : Libraries

- A higher level ops library calls the lower level libraries to implement the majority of functions for the control program and to implement some of the common timekeeping and scheduling functions

RST I : Site Specific Code

- The site specific code is handled by the site library
- The site contains functions that act as wrappers for the actual radar operations such as the clear frequency search and integration loop
- This allows the different hardware to be controlled by common shared control programs

RST I : Writing a Control Program

- Read through of normalscan...

RST I : Makefile

- A look at the makefile for a control program...

RST 3

RST 3 :Architecture

- RST3 uses a similar filesystem hierarchy to RST2, however RST is now installed in its own top level “rst” directory.
- git is now used for revision control
- a simplified compilation script
- improved packaging system allows older versions of distributions to be tracked and built
- support for diverse hardware

RST 3 : Architecture

Directory	Description
bin	Binaries for main software
build	Build and documentation scripts
code	Source code
doc	Documentation
include	C header files
lib	Libraries
log	Compilation and installation logs
script	Shell scripts
tables	Data tables and configuration files
usr	Secondary Hierarchy

RST 3: Architecture

- The arrangement of subdirectories has changed slightly under RST3
 - The “build” directory contains the build scripts, documentation building scripts and the packaging scripts
 - Source code is now grouped according to task under “code”
 - There is now a “superdarn” directory with “src.lib”, “src.bin”, “src.idl” etc.
- Walk through of FSH...

RST 3: Version Control

- RST 3 used git as its version control tool:
- RCS is somewhat cumbersome to use
 - Files need to be checked out for modification, checked in for distribution
 - Hard to use with multiple developers
 - Hard to use for distributed development
- Multiple solutions were investigated
 - CVS
 - Subversion
 - git
- git one out on its simplicity of implementation and its successful use in massive projects (Linux kernel)

RST 3 :Version Control

- Using git:
 - A git manages projects using git repositories
 - A directory called “.git” that contains the SCM information: changes,branches,logs etc.
 - The repository goes with the code
 - For distributed development, each developer “clones” the original repository and works on their own copy
 - Changes are reconciled by merging the different developer repositories together
 - Git operates on content, not files
 - Rather than checking in individual source code modules, you commit changes to a whole repository
 - This is a bit like taking a “snapshot” of the code

RST 3 :Version Control

- Git repositories are created by typing “git init”

```
git init
```

- Files are added to a repository by adding them:

```
git add hello.c
```

- Files are not “locked”, you can freely edit them

- Once edits are completed, you commit them to the repository

```
git commit .
```

RST 3 : Revision Control

Text

- Git commits do not have a version number, instead they have a hash code to uniquely identify them:

```
commit ca89e86fab564d9d5d9f177b6eb209d0fe511830
Author: Code Development Account <code@debian.localdomain>
Date:   Mon Jan 4 00:37:18 2010 +0000
```

RST 3 : Revision Control

- However git commits can be “tagged” with human readable names for convenience
- RST uses this facility to apply version numbers
- When we want to mark a commit with a release version number, it is given the tag “version.x.yy”
- Note: this means commits can be made independently of the version number, giving much finer and more flexibly control on when changes are committed

RST 3 :Version Control

- Because of our fine grain version control at the library and binary level, we do not have one top level git repository
- Each library or binary has its own git repository and the git tags are used to duplicate the same version numbering as RCS
- We do not need version.info any more as commits are repository wide - changes to all modules are tracked by the one commit

RST 3: Compilation

- There are slight changes to the compilation process
- The makefile templates now include a “makecfg” file used to setup any top-level variables
- Libraries now update the symbolic links to header files when they are compiled, not when they are installed on the system (This was a major weakness in the old system)

RST 3: Packaging

- The new packaging system uses package descriptions stored in the build/<project>/package/<package> directory. eg “build/superdarn/package/ros”
- A package description consists of:
 - pre and post install scripts
 - a package manifest “module.txt”
 - a build list “build.txt”
 - an commit ID list “id.txt”
 - supporting files such as shell scripts, configuration files etc

RST 3: Packaging

- The manifest “module.txt” lists all of the elements (libraries and binaries) that are included in that software package.
- It specifies the directory level, so elements can either be explicitly listed or whole trees can be defined.
- For example, to include all superdarn code: “codebase/superdarn”

RST 3 : Packaging

- The build list defines the order in which elements should be compiled
- It is used to resolve dependencies
- It also works on directories, so to compile all of the base libraries use “codebase/base/src.lib” (There are no-interdependencies on the base libraries)

RST 3: Packaging

- The ID list is a list of all the git commit IDs for the elements in that distribution, it is generated automatically when the package is updated
- The package directory is itself maintained as a git repository and the ID list is committed whenever a new version of the package is created
- This means that there is always a precise snapshot of exactly what version of each element was included in a package
- It also means that older versions of packages can always be created from the current master repository

RST 3 : Packaging

- The process is:
- commit all changes and apply the version tags to the source code
- Run “update.pkg <project> <pkg>”
 - eg. `update.pkg supedarn ros`
- Tag the new package with a distribution version:
 - `git tag "version.1.3-beta"`
- Build the distribution package
 - `make.pkg superdarn ros`
- This builds a distribution tree name “superdarn-ros-1.3-beta”
- Compile the package into a self-extracting archive
 - `make.dst superdarn-ros-1.3-beta`

RST 3: Linux/QNX6

- The new ROS is divided into two components
 - Linux “analysis” component
 - QNX6 “hardware component”
 - Communication between the two is via TCP/IP messages

RST3: ROS

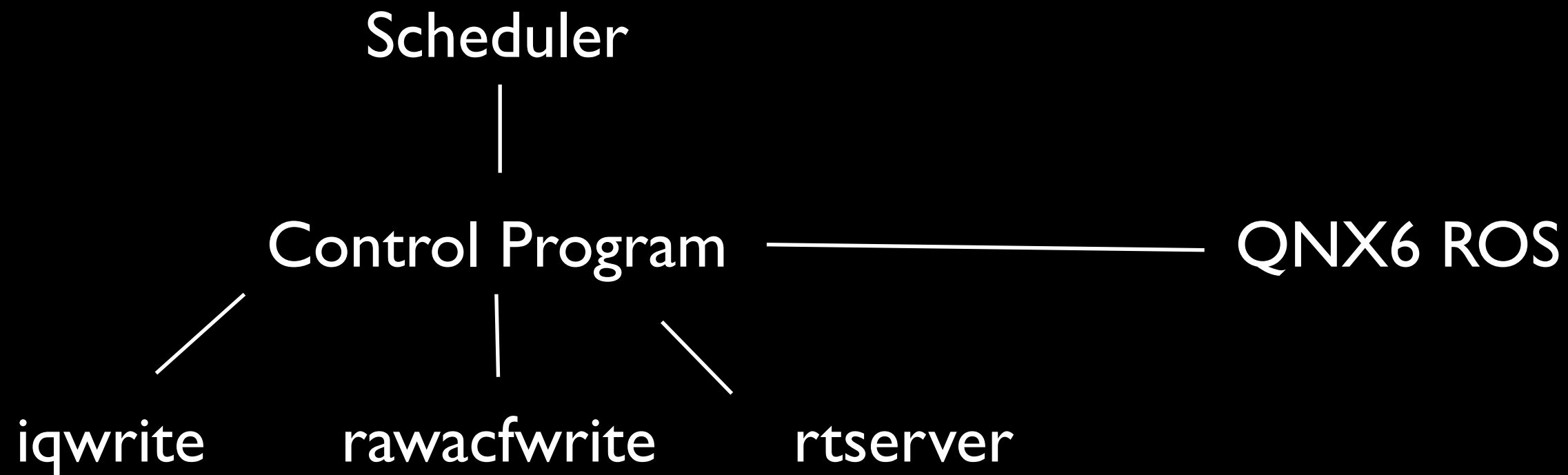
- Linux does have messaging libraries, POSIX etc, but we decided to keep things simple and use TCP/IP messages for all interprocess communication
- Tasks are re-written as “server/clients” - The control program acts as client, connecting to servers such as “errlog” or “fitacfwrite” to send data

RST 3: ROS

- Communication between the Linux and QNX6 components are also via TCP/IP.
- The TCP/IP messages implement the “virtual” radar
- There are messages for:
 - Setup
 - Timing sequence generation
 - Clear frequency determination
 - Transmit pulse sequence/take samples

RST: ROS 3

- The new ROS is greatly simplified:



RST 3: QNX6

- Over to Jeff...

RST 3 : Site Libraries

- The new site library system is designed to support multiple radar sites within one ROS installation.
- This was needed for sites with multiple radars
- Each site has a site library as before, however function names are site specific “SiteFheIntegrate”
- A master site library acts as the interface from the control program and uses pointers to specific functions to resolve the correct site at run time.

RST 3 : Site Libraries

- The master site library uses dynamic library loading to load only the necessary site library at run time
 - control program is started
 - it interrogates the QNX6 system to find the radar ID
 - the site library loads the appropriate site library for that site
 - the site library maps the control program site calls to the appropriate calls in the actual site library

RST 3: Control programs

- Walk through....

RST 3 : Closing remarks

- The new architecture is functionally complete, but still needs some polish
- There is no equivalent of tdisplay as we don't need it - The radar can now run a web server and run the real-time display directly
- The system is now fully distributed - we can run any process on any computer, even across the network

RST 3 : Closing remarks

- The new system has added a lot of “under the hood” enhancements designed for future expansion
- The new engineering processes, git version control, distributed development, packaging, etc. should help us in the future with the rapid recent expansion of the SuperDARN community and the large number of new radars