Distributed Acquisition and Data Analysis (DADA) Software Specification

Willem van Straten

September 29, 2011

Contents

This document describes the software component of the Distributed Acqusisition and Data Analysis (DADA) system. This software has been developed to support the recording and processing of timeseries from one or more digitizers using one or more computers. The first version of the code was written for the 2nd generation of the Caltech-Parkes-Swinburne Recorder, CPSR-II, installed at the Parkes Observatory in 2002. This code was updated and revised for the 2nd generation of the Dutch Pulsar Machine, PuMa-II, installed at the Westerbork Synthesis Radio Telescope (WSRT) in 2005. The next revision of this code will be used to implement the ATNF-Parkes-Swinburne Recorder, APSR, to be installed at Parkes in 2007.

The DADA software must be run on one or more data acquisition machines, called the *Primary* nodes, and may optionally include the use of a (possibly distributed) processing cluster of *Secondary* nodes.

Each Primary Node interfaces directly with the data acquisition hardware at the telescope; for example, via a Direct Memory Access (DMA) card, a PCI card, or a Network Interface Card (NIC). From RAM, the Primary Node may write the data to a local file system, forward the data on to one or more Secondary nodes, or process the data directly from RAM.

The communications plan and protocol used for sending the data from Primary to Secondary node may change over the life of an instrument. Therefore, an adaptable and modular design philosophy has been employed in the data flow control software, based on the use of a ring buffer in shared memory. This document contains an overview of the software model, including a discussion of the design decisions made at each level of command, control, and configuration.

1. Introduction

1.1. Design Goals

1. modularity
2. separation of data transport, control, and data reduction
3. near real-time data reduction
4. elegant scaling to slower data reduction speeds

1.2. Overview

1.2.1. Workstation Cluster

The functionality of DADA may be divided across multiple workstations. These are divided into two main classes:

1. Primary nodes - workstations equipped with Direct Memory Access (DMA) card, large data storage, and high-speed interconnect.
2. Secondary nodes - workstations equipped with high-speed interconnect and modest data storage facilities.

A single Primary node may have multiple Secondary nodes associated with it. To these, it will send the observed data either during an offline, post-recording stage or in real-time.

1.2.2. Software

The major functionality of DADA is divided into five categories:

1. Data Flow Control - high-bandwidth data transfer and storage
2. Data Reduction - process and archive the observed data
3. Command and Monitoring - communication channels for external control
4. Control Interface - centralized access to Command and Monitoring
5. Configuration and Scheduling - files and databases for automated control

Communication between each of the levels of software will take place primarily through shared memory, semaphores, and internet socket connections.

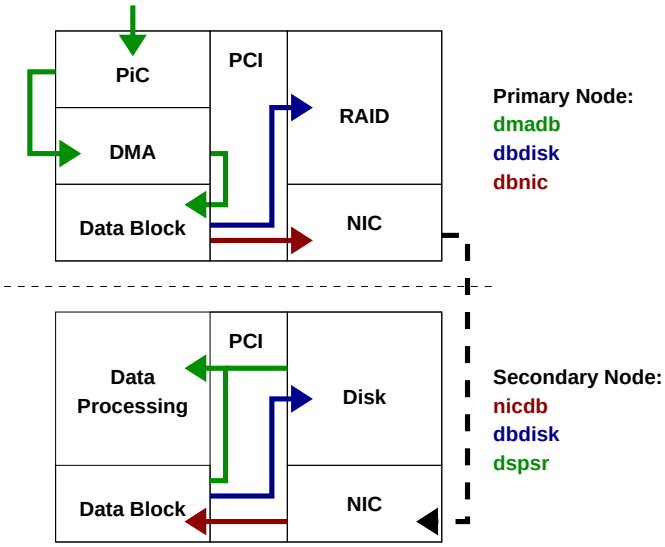

Figure 1.1.: Schematic overview of DADA data flow; note the parallelism and symmetry of the operations on Primary and Secondary nodes.

Data Flow Control

Data Flow Control software will control all aspects of the high-bandwidth data recording, including DMA control, interim storage on internal RAID, and high-speed transfer between the Primary and Secondary nodes. The transfer to Secondary nodes will include the overlap required to compensate for data reduction losses (owing to dispersion smearing, filter rise times, etc.) as specified by the Configuration and Scheduling software.

On both Primary and Secondary nodes, data will be stored in a large buffer established as shared memory, called the Data Block. The various tasks that must run in parallel will be implemented as unique processes, as opposed to multiple threads of a single process. Therefore, access to the Data Block will be controlled by an inter-process communication (IPC) locking method, known as a semaphore.

Rationale: It is better to begin with multiple processes and IPC in the early stages of development because this paradigm is more modular. For example, the process that reads data from the Data Block and writes it to local file storage may be run on either a Primary or Secondary node. If data reduction can later be performed in real-time, the disk writing client may be replaced by a data processing client.

Data Reduction

On both Primary and Secondary nodes, one or more Data Clients may attach to the Data Block and operate on the data. A single, high-priority Data Client will be given permission to flag sub-blocks as processed. Initially, this client will be part of the Data Flow Control software that writes the data to local file storage. Later, this client may be a data processing client.

Data Clients may perform any number of tasks, including various forms of data reduction, calculation and display of data quality statistics such as the bandpass, storage of the data to some form of medium, or farming the data out to a grid. The data reduction client will process the data according to the specifications of the Configuration and Scheduling software.

Command and Monitoring

Command and Monitoring software includes the Command software that establishes low-bandwidth network communication channels between Primary and Secondary nodes and the Control Interfaces. These channels are used for sending high-level control commands, such as start, stop, and information about the source. These communication channels may be implemented as a control thread in each component of the Data Flow Control software.

The Monitoring software will perform any tasks required to maintain proper operation of the instrument and present useful information to the user. This includes monitoring telescope status information from TMS, disk space consumption, network traffic, CPU load, etc.

Control Interface

The Control Interface software defines the centralized command/control point, and will be connected to the various communication channels established by the Control and Monitoring software on each of the Primary and Secondary nodes. The Control Interface software should be run on a single designated workstation, as it will provide the means through which other processes may treat the various components as a single instrument. For example, a text or graphical user interface and/or automated scheduling program may connect, issue commands, and inquire about the status of the instrument. A textual user interface (TUI) will be developed to connect to the Control Interface and:

- allow DADA to be configured, started, and stopped;
- display various status variables; and
- create diagnostic plots, such as the passband and digitizer statistics

Configuration and Scheduling

Configuration and scheduling software will be make use of database information to configure the instrument and schedule data reduction operations, based upon the parameters of the observation.

2. Data Flow Control Software

Data Flow Control software running on the Primary and Secondary nodes must handle the flow of data in a modular and extensible manner, enabling future developments by replacement of a single component. The required modularity is met by basing all data transfer on a single ring buffer protocol, which will be known as the Data Block.

2.1. Data Block

The Data Block is a ring buffer that will be allocated as a shared memory resource, logically divided into a header block followed by a number of sub-blocks. Each sub-block will have an associated byte count that may be used to calculate the time offset from the start of the observation. Only one contiguous stream of data may be represented in the Data Block at any one time; therefore, the size of the Data Block will determine the maximum amount of time required to flush the ring buffer between stopping an observation and starting the next observation.

At the beginning of an observation, every sub-block of the ring buffer will be empty and the header block will be initialized with the relevant observation information (such as bandwidth, centre frequency, source, start time, etc.). In order that data acquisition may be started before the data are valid, data may be written to sub-blocks before the start-of-data flag is raised. Data may be read from sub-blocks only after the start-of-data flag is raised.

Data will be written to sub-blocks in sequential order until the end of the observation, at which point an end-of-data flag will be raised, the last full sub-block and the number of valid bytes written to this sub-block will be recorded. Data will be read from sub-blocks in sequential until the end-of-data flag is raised and the current sub-block is equal to the last full sub-block.

Data that are written to the Data Block may not necessarily be valid. Therefore, each sub-block will have associated variables to indicate:

- the state of the block: empty or full.
- the byte offset at which data became valid
- the byte offset at which data became invalid

Note that data can transit from valid to invalid (or vice versa) only once per data block. However, these transitions may occur an arbitrary number of times between the start and end of data.

2.1.1. Write Client

A single, high-priority process, called the Write Client, will be given write access to the Data Block; only the Write Client can change the state of a sub-block from empty to full. The Write Client can write data to the ring buffer before flagging the start of the observation. In this way, it can clock data without activating the Read Clients, and may change the state of the validity flags before raising the start-of-data flag.

After raising the start-of-data flag, the Write Client will not write data to a sub-block until its state is empty; after filling a sub-block, it will change its state to full and set the data validity byte offsets. If the Write Client cannot obtain an empty sub-block, an overflow condition will occur; this condition will be handled according to the mode of operation:

- **contiguous**, a contiguous stream of data is required (as is often the case in search observations). In this case, data overflow is treated as an error that is propagated to the Command and Control software, and data acquisition is stopped.
- **discontinuous**, an end-of-data is written to the Data Block, and the Write Client continues to receive data from its source. When the ring buffer has been cleared by the Read Client, the Write Client starts a new observation and data acquisition continues.
- **tolerant**, the Write Client will wait indefinitely for the next empty sub-block.

Other specifics of overflow handling will depend upon the application. For example, discontinuous overflow handling may include a sending a signal to to the Primary Node that instructs it to move on to the next Secondary node.

2.1.2. Read Clients

One or more Read Clients may attach to the Data Block and read the data from sub-blocks marked as full. Only the bytes designated as valid will be used from each sub-block. Only a single, high-priority Read Client will be given permission to change the state of a sub-block from full to empty. Read Clients will access sub-blocks in contiguous order after the start-of-data flag is raised and until the end-of-data condition is encountered.

Example

Consider a tight schedule, in which the time required to synchronize and start data acquisition is considered too costly. In this case, some time may be saved by starting data acquisition before the telescope is on source and continuing to acquire data while the telescope slews between sources.

In this case, the Write Client will begin writing data to the Data Block before raising the start-of-data flag. The Write Client knows the UTC time at which it started clocking data into the Data Block. When the signal is given to the Write Client that the data became valid at a certain UTC time, the Write Client can go back to that time in its

buffer, flag all data from that point to the present buffer as valid, and raise the start-of-data flag. In this way, the Write Client can retroactively flag data as valid **before the start of the observation**.

In order to slew to the next source, a data invalid message must be sent to the Write Client before the data becomes invalid. The Write Client will continue to clock data into the Data Buffer, but the data will be marked as invalid and therefore will be ignored by the Read Clients; the designated Read Client will simply mark the buffers as empty as they are encountered. (This is why retroactive validity flagging can be done only before the start of the observation.)

2.2. Data Flow Write Clients

Write Client software will read data from a device and write it to the Data Block.

2.2.1. DMA Client: `dmadb`

The DMA Client software, `dmadb`, is responsible for transferring data from the telescope to the Data Block. It will talk directly to the PiC through its PCI interface, start and stop the data transfer, and record the UTC start time of the observation. Data from the PiC will be transferred to Primary node RAM via a Direct Memory Access (DMA) card that is commercially available from Engineering Design Team (EDT). The DMA Client software will:

1. allocate a number of fixed memory buffers of a size and number to be determined during the testing stage;
2. send start and stop instructions to the PiC via the PCI/DMA interface;
3. determine the UTC start time of the first sample recorded
4. copy filled DMA buffers to the Data Block; and
5. monitor the number of DMA buffers filled and copied, ensuring that no data overflow occurs.

The DMA buffers will be separate from the Data Block buffers and accessed only by the DMA card driver and `dmadb`. Once started, DMA transfer will continue uninterrupted until a stop flag is raised or an overflow occurs.

2.2.2. Network Interface Client: `nicdb`

The software for network I/O will run on both Primary and Secondary nodes. The Data Flow Control software running on the Secondary nodes, `nicdb`, will open a port and listen for incoming Data Flow Control connections from `dbnic`, which will run on the Primary nodes. A single incoming channel will be connected and used to establish a high-bandwidth data communication channel between a single Primary node and a single Secondary node. The protocol for the network communications will be a simple, custom-built design on top of internet sockets. This may change in the future to some sort of grid-based protocol. Data received via this communication channel will be copied

to the Data Block in contiguous order. Each packet of data will be preceded by a copy of the Data Block header from the Primary node. This header will be copied to the Secondary node Data Block.

The `nicdb` software has the responsibility to monitor the Data Block and ensure that there is sufficient space to hold incoming data packets. It will send a message to the Primary node if there is insufficient space, and the Primary node will cease data transfer, possibly initiating data transfer to the next in Secondary node in the ring.

2.3. Data Flow Read Clients

Read Client software will read data from the Data Block and write it to a device.

2.3.1. Data Storage Client: `dbdisk`

Writes data blocks to disk, breaking up data into files of arbitrary length. Each file will be preceded by the header block from the Data Block. Runs on either Primary or Secondary nodes, depending on the mode of operation. After each file is written to disk, an entry will be added to an ASCII text log file; each entry will describe:

- the full path to the file
- the time it was written
- the size of the file
- the time required to write the file
- the observation identifier

This log file will be polled by the Configuration and Scheduling software, which will add the information to a centralized database.

2.3.2. Network Interface Client: `dbnic`

This software runs on the Primary nodes; it reads from the Data Block and writes to one or more Secondary nodes, breaking up data into packets of arbitrary length. The total length of data sent to an individual Secondary node will be independent of the Data Block buffer sizes, and may depend on the overlap specified by the Configuration and Scheduling software. Header information (including all available observation information as well as offset byte counts) will be sent with each block of data transmitted to the Secondary nodes.

3. Data Reduction Software

3.1. Operational Phases

The data reduction software supports three models:

- **offline** processing: performed after the recording has finished
- **simultaneous** processing: performed during the recording
- **diskless** processing: as above, but completely in memory

3.1.1. Offline Data Reduction

The data reduction software operates only on data files stored on local disk space, and only after the recording has completed. After the observations have been completed, the data files from each Primary node will be farmed out to the Secondary nodes. After each file has been copied, an entry will be added to the log file that is monitored by the Configuration and Scheduling software, which will attend to the data reduction as described in Chapter 6.

3.1.2. Simultaneous Data Reduction

The data reduction software runs during the observation. In this mode, data are sent directly to Secondary nodes and written to disk. As before, an entry will be made in a log file and the Configuration and Scheduling software will coordinate the data reduction.

3.1.3. Diskless Data Reduction

The data reduction software is able to keep up completely with the flow of data on the Primary nodes. In this case, the data is never written to disk, and `dspsr` operates as a Read Client directly connected to the Data Block. Another Read Client may be written that will farm data reduction tasks to a grid computing facility. In this case, the data reduction Read Client will have to incorporate the Configuration and Scheduling tasks.

4. Command and Monitoring Software

4.1. Command

Each element of the DADA software must be coordinated to operate as a single instrument. Therefore, many of the processes described in the previous chapter will have to be synchronized and configured through communications channels. Some degree of synchronization will be achieved through the hand-shaking protocol of the Data Block specification. Other communication requirements will be met through internet socket connections.

4.1.1. Data Block Communications

Many of the Data Clients can be implemented as a persistent process, like a daemon, that is configured once during an initialization stage and runs automatically from that point onward. Apart from configuration, the behaviour of these automatic processes will depend completely upon the state of the Data Block.

Two Read Client programs that can run in this manner are `dbdisk` and `dbnic`. These automatic processes need only start reading from the Data Block when it is active, as determined by the behaviour of the Write Client. They start when the header is initialized and stop when the end of data flag is raised. Also, the `nicdb` Write Client can be run as an automatic process that starts when it receives packets from the Primary Node and ends when the end of data message is received. If it is shown that the operation of these Data Clients may have to change from observation to observation, then there are two possible solutions:

- Stop and restart the daemons with different configuration parameters
- Enable socket communications that set configuration parameters

4.1.2. Internet Socket Communications

Certain processes will require internet socket communications in order to be configured between observations and to start and stop the observation. In order that communications may be sent and received during normal operation, the processes that require socket communications will be multi-threaded. The communication threads may have lower priority than the main thread, if required. More than one communication channel may be opened to each process; however, only one channel will be able to issue control commands. The others may only inquire about the status of the process.

All communications will be human readable, ASCII text. This enables interface testing using standard tools like telnet. If large amounts of binary data must be sent between

the Control Interface and Data Flow Control software, then it should be done using a separate communication channel designated for this purpose. Text commands will be sent on a single line of text. After every command received, the process will respond with `ok` or `fail`, followed by any additional information, and ending with the command prompt.

DMA Data Client

The `dmadb` processes running on each of the primary nodes require careful synchronization with the Control Interface software, especially if they are all to be started on the same UTC second. For this reason, it will not suffice to remotely start the processes at the desired beginning of each observation. The processes must be persistent and must maintain socket communications with the Control Interface. The Command Interface to `dmadb` is described in Section 4.3.

4.2. Monitoring

Monitoring processes will be run on all nodes in the instrument, reporting on remaining disk space, CPU load, network traffic, etc. At the time of this writing, it is not clear if standard cluster monitoring tools like Ganglia will suffice. For example, it may prove useful to have a regular report on which Secondary nodes are currently receiving data. This information would have to come from `dbnic`, possibly via a socket connection to this process.

In addition to live monitoring, it may also prove useful to maintain a database of relevant statistics, such as the average time required to write a block of data to file or over the network. These monitoring tasks would be performed by the relevant process, `dbdisk` and/or `dbnic` and communicated to a central database via some protocol.

4.3. Primary Write Client Command Interface

This section describes the behaviour of the Write Client software that will run on each of the Primary nodes, known as the Primary Write Client (PWC) software. In the case of PuMa-II, the PWC is `puma2_dmadb`.

4.3.1. Operational States

The PWC has four main states of operation:

- **idle** waiting for configuration parameters
- **prepared** configuration parameters received; waiting for start
- **clocking** clocking data but not recording
- **recording** recording data

Idle State

In the idle state, the PWC sleeps until configuration parameters are sent from the control software. All configuration parameters are sent in a single ASCII header, and at any time before entering the **recording**, the duration of the recording may be specified

- **HEADER text** copy text to the Header Block and enter the **prepared** state.
- **DURATION time** record for the time specified as *HH:MM:SS* or number of samples.

Prepared State

In the prepared state, the PWC sleeps until a start command is sent from the control software. There are two different start commands that can be received in this state:

- **CLOCK** enter the **clocking** state
- **START [YYYY-MM-DD-hh:mm:ss]** enter the **recording** state

If the optional UTC time argument is specified, the PWC will enter the **recording** state at the specified time. Otherwise, the PWC will enter the requested state at the next available opportunity (for PuMa-II, on the next `SYSTICK`).

Clocking State

In this state, the PWC software clocks data into the Data Block but does not flag the data as valid. The PWC will overwrite the data in each sub-block, and will remain in this state until one of the following commands is received:

- **STOP** enter the **idle** state immediately
- **REC_START YYYY-MM-DD-hh:mm:ss** raise the valid data flag at the specified UTC time in the data stream and enter the **recording** state

Note that the UTC time specified in the first argument to `REC_START` may be any time in the future. If it is in the past, then the difference between the specified UTC and the present cannot be greater than the amount of time corresponding to the length of the Data Block.

Recording State

In this state, the PWC software clocks data into the Data Block, flags the data as valid, and will not overwrite a sub-block until it has been flagged as cleared. The PWC will remain in this state until one of the following commands is received:

- **STOP [YYYY-MM-DD-hh:mm:ss]** enter the **idle** state
- **REC_STOP YYYY-MM-DD-hh:mm:ss** raise the end of data flag at the specified UTC time in the data stream and enter the **clocking** state

Note that with the exception of `REC_START` all UTC times **must** be in the future. If this is not true, the command will take effect immediately. If `DURATION` has been specified, it will not be corrected to reflect the difference between requested and actual start times.

The command interface described in this section is implemented by the `dada_pwc` software, as documented in Appendix E.

4.4. Primary Write Client Main Loop

In addition to providing the command interface described in the previous section, the Primary Write Client must implement the transfer of data to the Data Block. This section gives an outline of how this will be done.

4.4.1. Top Down Description

The following describes the behaviour of the Primary Write Client

- Initialization
  - read configuration file
  - parse command line options
  - initialize DMA and PiC cards
  - connect to Data and Header Blocks
  - open a port and listen for command connection
- Main Loop
  - Idle State
    - wait for configuration
    - set configuration
    - enter **prepared** state
  - Prepared State
    - wait for a command
    - if command=`CLOCK`, enter **clocking** state
    - if command=`START`, enter **recording** state
    - if command=`STOP`, return to **idle** state
  - Clocking State (loop)
    - check for a command
    - if command=`STOP`, return to **idle** state
    - if command=`REC_START`, enter **recording** state
    - copy buffer from DMA to Data Block
  - Recording State (loop)
    - check for a command
    - if command=`STOP`, flag end of data (EOD) and return to **idle** state
    - if command=`REC_STOP`, flag EOD and enter **clocking** state
    - wait for next free Data Block sub-block

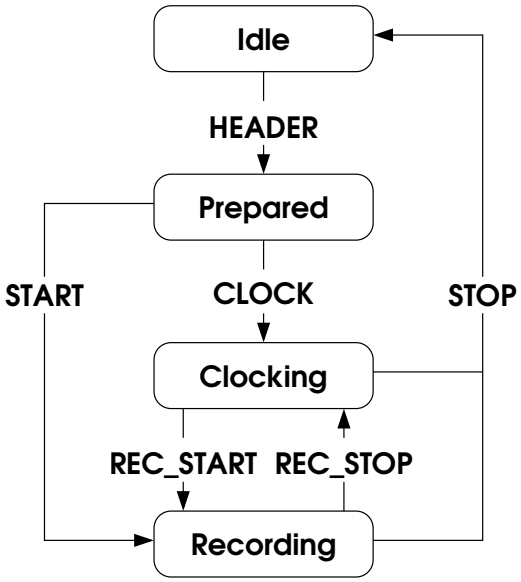

Figure 4.1.: Schematic overview of DADA control flow. Note that the state can be toggled between clocking and recording.

    - ∗ copy buffer from DMA to Data Block and flag as valid
- Shutdown
  - close command connection
  - disconnect from Data and Header Blocks

The behaviour described in this section is implemented by the `dada_pwc_main` software, as documented in Appendix F.

5. Control Interface Software

The Control Interface software will implement the single instrument look and feel of a DADA-based instrument. A single process, `dada_nexus`, will:

- connect to the Command interface of each Primary Write Client (PWC)
- issue commands as necessary to each PWC
- collate monitoring information from all processes
- provide Command and Monitoring connections to outside world
- start and stop DADA processes on any node as required

In addition to maintaining the required control connections with processes on each of the Primary nodes, the Control Interface software will also present a Command and Monitoring connection to the outside world. To this connection, an operator can connect to issue commands and check on the status of the instrument/observation. As before, multiple communication channels may be connected, but only one will be able to send control commands. The operator will connect using a textual user interface program, `dadatui`, which will present all information from the Control Interface in an organized manner, possibly employing the curses terminal control library.

Another process, `dadatms`, will be written to provide automated control of the DADA instrument by the WSRT TMS. Commands sent by TMS will be received by `dadatms`, converted to text commands, and sent to `dada` through the operator interface.

The Control Interface software will enable complete initialization of the DADA instrument through a single command. That is, on any or all nodes, it will be able to start and stop various processes, create and destroy the Data Block shared memory and semaphore resources, and perform whatever other tasks prove useful in the initialization and configuration of the DADA instrument.

5.1. Data Flow configuration

The `dada_nexus` software will distribute Data Flow configuration options to each Primary Write Client (PWC). Data Flow configuration describes the parameters of the Data Flow Control software that may need to change between observations. These currently include:

- `FILE_SIZE` requested size of data files
- `OBS_OVERLAP` the amount by which neighbouring files overlap
- `TARGET_NODES` the nodes to which data will be sent

These parameters configure the behaviour of the `dbnic` or `dbdisk` processes. As there may be a long delay between the time at which the data was acquired on the Primary node and that at which data reaches the Secondary nodes, these processes must be able to operate independently of the Control Interface software. Therefore, the Data Flow configuration parameters will be stored as attributes in the ASCII Header for each observation (see Chapter B). The Header Block for each PWC is unique and each data path may be configured independently.

The `dada_nexus` software will read all configuration parameters from observation specification files. This file will contain all information about the observation, including the Data Flow configuration configuration parameters, as described in the following section. The `dada_nexus` will parse the specification file and create a unique header for each PWC.

5.2. Observation Specification

Each observation must be completely specified before it may be started on the DADA instrument. Specifications will be created using the specification tool, a graphical user interface designed to ease the creation, duplication, and verification of observation and data reduction configurations.

5.3. Control Interface Configuration

Will also be read from a text file. Parameters include:

- `PWC_PORT` the port to connect with the PWC Command Interface
- `NUM_PWC` the number of Primary Write Clients
- `PWC_`N the name of the Nth Primary Write Client
- `COM_POLL` the polling interval for communication connections
- `HDR_SIZE` the size of the Data Header
- `HDR_TEMPLATE` file containing the header template
- `SPEC_PARAM_FILE` file containing the specification parameter list

6. Configuration and Scheduling Software

Depending upon the objectives of the experiment, DADA configuration may change from observation to observation.

6.1. Instrumental Configuration

Based upon the parameters of observation, the `dada` control program will set up various operational parameters, including:

- buffer sizes: DMA, Data Block, file and network I/O
- number of Primary nodes
- number of Secondary nodes
- assignment of Secondary to Primary nodes
- operational mode: offline, simultaneous, or diskless
- overlap required between Secondary nodes

These operational parameters will be stored in a **configuration database**, which will specify the instrument configuration and data reduction requirements for various combinations of source, receiver, centre frequency, band width, etc.

Entries in the configuration database may have an expiration date associated with them. In this manner, the observer may specify special configuration and/or reduction options for a specific experiment without permanently changing the default behaviour.

6.2. Data Reduction

The Configuration and Scheduling program, `dadaskd`, will be used to configure and schedule all data reduction operations. Before the **diskless** mode of data reduction is implemented, all data will exist as a file on either the Primary or Secondary nodes. Whenever a file is written to disk, an entry will be registered in a centralized **observations database**, which will contain basic header information such as

- source name
- start time (UTC)
- centre frequency (MHz)
- band width (MHz)

as well as the location (machine and file name) of the data. Each entry will also contain a time-stamped list of **performed operations**, describing when the data was written, when and how it was processed, when it was deleted, etc. The header information will be used to find matches in the configuration database. An observation may be processed in multiple ways, as specified by the list of **requested operations** in the configuration database entry.

If it is possible to achieve two requested operations in one execution of `dspsr` then this will be done. Otherwise, data reduction operations will be performed one at a time. After each operation is completed, it will be recorded in the list of performed operations of the observations database entry.

The scheduling software will periodically check or poll the observations database. Any entries that require data reduction will be scheduled according to the data reduction parameters of the requested operations in the corresponding configuration database entry. An observation will be considered completely processed when the list of performed operations is equal to the list of requested operations. At this point, the raw data will be deleted or archived.

7. APSR Objectives

APSR will depend upon the following new techniques, which must be implemented, demonstrated, and rigorously tested.

- streaming UDP packets from board to nodes
- optimal use of multiple cores
- processing directly from RAM
- simultaneously processing and writing to disk
- processing data written to disk

In addition, the following management software must be written/adapted:

- Data reduction control: scheduling and logging
- User Interfaces: text, gui, web-based?
- TCS Interface
- Collection, presentation, and archival of reduced data

7.1. Streaming UDP packets from board to nodes

Can it be done while processing? How should dropped packets be handled?
Who: Grant and Willem
Status: hardware ready, no software

7.2. Optimal use of multiple cores

Is a multi-threaded solution the most efficient?
Who: Willem
Status: implemented but not optimized, currently debugging

7.3. Processing directly from RAM

Should multiple threads be triggered to return results before reaching end-of-data, or should end-of-data be induced at regular intervals?
Who: Willem
Status: implemented, tested on single archive only

7.4. Simultaneously processing and writing to disk

Should a monitor be used to schedule the two tasks intelligently, or should priorities be assigned to the processes?
Who: Willem *and Andrew?*
Status: requires redesign of ring buffer

7.5. Processing data written to disk

Can the redistribution and processing of data written to disk take place during easy observations, or only offline? How should different data sets be scheduled?
Who: Willem *and Andrew?*
Status: simple plan: process easiest (lowest DM) observations first

8. Summary

The following table summarizes the DADA software, its basic functionality, and the machine on which it will run.

Name	Function	Machine
`dmadb`	Transfers data from EDT buffers to the Data Block.	Primary
`udpdb`	Transfers data from UDP stream to the Data Block.	Primary
`dbdisk`	Reads data from the Data Block and writes it, with header information, to disk.	Both
`dbnic`	Reads data from the Data Block and sends it, with header information, to a Secondary node.	Primary
`nicdb`	Receives data from Primary node and writes it to the Data Block.	Secondary
`dspsr`	Attaches to the Data Block and processes raw data according to specification, writing results to disk.	Secondary
`dadanexus`	Connects to the Command and Control interface of the various Primary Write Clients. Accepts a text-based control connection.	fixed
`dadatcs`	Translates telescope control system commands into text-based commands as accepted by `dada` control connection.	fixed
`dadatui`	Textual user interface connects to `dada`, displays various quantities of interest, and controls the instrument.	variable
`dadaskd`	High-level code that schedules and records data reduction operations	fixed

A. Data Block Ring Buffer

In this chapter, the Data Block API is specified in detail. The Data Block is the ring buffer through which the primary data flow will take place on both Primary and Secondary nodes in the cluster. Access to the ring buffer shared memory is controlled by an inter-process communication semaphore.

The Data Block API includes software for creating and initializing the shared memory and semaphore resources, locking the shared memory into physical RAM, connecting to the ring buffer, writing data to the ring buffer and reading data from the ring buffer.

A.1. Creation, Connection, and Destruction

The Data Block ring buffer is accessed through a data type named `ipcbuf_t`, which is declared and initialized as in the following example:

```
#include "ipcbuf.h"
ipcbuf_t ringbuf = IPCBUF_INIT;
```

To create a ring buffer, call

```
int ipcbuf_create (ipcbuf_t* ptr, int key, uint64 nbufs, uint64 bufsz);
```

- `ptr` is a pointer to an unallocated ring buffer handle
- `key` is a unique identifier (range of acceptable values???)
- `nbufs` is the number of sub-blocks in the ring buffer
- `bufsz` is the size of each sub-block in the ring buffer

After the ring buffer has been created, it is ready for use. The ring buffer resources will remain available until calling

```
int ipcbuf_destroy (ipcbuf_t* ptr);
```

- `ptr` is a pointer to an allocated ring buffer handle

That is, even if the process that created the ring buffer exits, the shared memory and semaphore resources will remain allocated in computer memory. In order to connect to a previously created Data Block ring buffer, call

```
int ipcbuf_connect (ipcbuf_t* ptr, int key);
```

- `ptr` is a pointer to an unallocated ring buffer handle
- `key` is the unique identifier passed to `ipcbuf_create`

To disconnect, call

```
int ipcbuf_disconnect (ipcbuf_t* ptr);
```

- `ptr` is a pointer to a connected ring buffer handle

Note that, after calling `ipcbuf_create`, the process is connected to the newly-created ring buffer and it is not necessary to call `ipcbuf_connect`. Similarly, after calling `ipcbuf_destroy`, it is not necessary (or possible) to call `ipcbuf_disconnect`. After the process is connected to the Data Block ring buffer, it is possible to write or read data.

A.1.1. Locking into Physical RAM

In order to ensure that the Data Block ring buffer remains in RAM and is not swapped out by the virtual memory manager, call

```
int ipcbuf_lock_shm (ipcbuf_t* ptr);
```

and, to unlock,

```
int ipcbuf_unlock_shm (ipcbuf_t* ptr);
```

A.2. Writing to the Data Block

After connecting to the Data Block ring buffer, it is possible to write data to it.

A.2.1. Locking and Unlocking Write Access

Naturally, only one process may write data to the ring buffer; therefore, the Write Client must first lock write access to the buffer by calling,

```
int ipcbuf_lock_write (ipcbuf_t* ptr);
```

- `ptr` is a pointer to a connected ring buffer handle.

Similarly, write permission may be relinquished by calling

```
int ipcbuf_unlock_write (ipcbuf_t* ptr);
```

- `ptr` is a pointer to a connected ring buffer handle.

A.2.2. Write Loop

After locking write access to the Data Block ring buffer, the Write Client will generally enter a loop in which it

1. requests the next sub-block to which data may be written,
2. fills the sub-block
3. marks the sub-block as filled

Step 1 is performed by calling

```
char* ipcbuf_get_next_write (ipcbuf_t* ptr);
```

- `ptr` is a pointer to a connected ring buffer handle
- RETURN value is the pointer to the next available sub-block

```
int ipcbuf_mark_filled (ipcbuf_t* ptr, uint64 nbytes);
```

- `ptr` is a pointer to a connected ring buffer handle
- `nbytes` is the number of valid bytes in the sub-block

If `nbytes` is less than the number of bytes in each sub-block, as set by the `bufsz` argument to `ipcbuf_create`, then an end-of-data condition is set.

A.2.3. Writing before Start-of-Data

By default, when a Data Block ring buffer is created, the start-of-data state is enabled and any data written by the Write Client will be made available to the Read Client. However, in some cases it may be useful for the Write Client to write data to the Data Block before making it available to the Write Client. For example, the trigger to begin data acquisition may arrive later than the desired data acquisition start time.

To begin writing data before the actual start of valid data, it is necessary to first disable the start-of-data flag by calling

```
int ipcbuf_disable_sod (ipcbuf_t* ptr);
```

- `ptr` is a pointer to a connected ring buffer handle

The Write Client may then enter a loop identical to that described in the previous section: requesting, filling, and marking. However, when the start-of-data flag is disabled, the message that a sub-block has been filled is not passed on to the Read Client and the Write Client will over-write filled sub-blocks as necessary. The Write Client raises the start-of-data flag by calling

```
int ipcbuf_enable_sod (ipcbuf_t* ptr, uint64 st_buf, uint64 st_byte);
```

- `ptr` is a pointer to a connected ring buffer handle
- `st_buf` is the absolute count of the first valid sub-block
- `st_byte` is the first valid byte in the first valid sub-block

Note that `st_buf` is an absolute sub-block count, equal to the total number of sub-blocks filled before the first valid sub-block. Naturally, it is not possible to raise the start-of-data flag for a buffer that has already been over-written. Therefore, the start sub-block count plus the total number of sub-blocks must be greater than the current sub-block count, or

```
st_buf > ipcbuf_get_write_count - ipcbuf_get_nbufs
```

A.3. Reading from the Data Block

After connecting to the Data Block ring buffer, it is possible to read data from it.

A.3.1. Locking and Unlocking Read Access

Only one process may remove data from the ring buffer by flagging it as cleared. This process, the Read Client, must first lock read access to the buffer by calling,

```
int ipcbuf_lock_read (ipcbuf_t* ptr);
```

- `ptr` is a pointer to a connected ring buffer handle

Similarly, read permission may be relinquished by calling

```
int ipcbuf_unlock_read (ipcbuf_t* ptr);
```

- `ptr` is a pointer to a connected ring buffer handle

A.3.2. Read Loop

After locking read access to the Data Block ring buffer, the Read Client will generally enter a loop in which it

1. requests the next sub-block containing data,
2. operates on the data in the sub-block
3. marks the sub-block as cleared

Step 1 is performed by calling

```
char* ipcbuf_get_next_read (ipcbuf_t* ptr, uint64* bytes);
```

- `ptr` is a pointer to a connected ring buffer handle
- `bytes` will be set to the number of valid bytes in the sub-block
- RETURN value is the pointer to the first valid byte in the sub-block

Step 3 is performed by calling

```
int ipcbuf_mark_cleared (ipcbuf_t* ptr);
```

- `ptr` is a pointer to a connected ring buffer handle

A.4. Data Block Abstraction

The fact that data is written to a ring buffer with individual sub-blocks may be abstracted from high-level code, thereby allowing the Data Block to be treated like any other storage device. This abstraction is accessed through a data type named `ipcio_t`, which is declared and initialized as in the following example:

```
#include "ipcio.h"
ipcio_t data_block = IPCIO_INIT;
```

To create an abstract ring buffer, call

```
int ipcio_create (ipcio_t* ptr, int key, uint64 nbufs, uint64 bufsz);
```

- `ptr` is a pointer to an unallocated abstract ring buffer handle

- `key` is a unique identifier (range of acceptable values???)
- `nbufs` is the number of sub-blocks in the ring buffer
- `bufsz` is the size of each sub-block in the ring buffer

After the abstract ring buffer has been created, it is ready for use and the resources will remain available until calling

```
int ipcio_destroy (ipcio_t* ptr);
```

- `ptr` is a pointer to an allocated abstract ring buffer handle

In order to connect to a previously created Data Block ring buffer, call

```
int ipcio_connect (ipcio_t* ptr, int key);
```

- `ptr` is a pointer to an unallocated abstract ring buffer handle
- `key` is the unique identifier passed to `ipcio_create`

To disconnect, call

```
int ipcio_disconnect (ipcio_t* ptr);
```

- `ptr` is a pointer to a connected ring buffer handle

Note that, after calling `ipcio_create`, the process is connected to the newly-created ring buffer and it is not necessary to call `ipcio_connect`. Similarly, after calling `ipcio_destroy`, it is not necessary (or possible) to call `ipcio_disconnect`. After the process is connected to the Data Block ring buffer, it is possible to write or read data.

A.4.1. Read/Write Access

To begin writing or reading data through the abstract ring buffer interface, the Write or Read Client must first call

```
int ipcio_open (ipcio_t* ptr, char rw);
```

- `ptr` is a pointer to a connected abstract ring buffer handle.
- `rw` is either 'W', 'w', 'R', or 'r'

The meanings of the `rw` character codes are as follows:

- **W** open for writing valid data
- **w** open for writing before valid data
- **R** open for reading as primary Read Client
- **r** open for reading as secondary Read Client

Similarly, write or read permission may be relinquished by calling

```
int ipcio_close (ipcio_t* ptr);
```

- `ptr` is a pointer to a connected ring buffer handle.

When the Write Client calls `ipcio_close`, an end-of-data is written to the Data Block. Data are written to the abstract ring buffer by calling

```
ssize_t ipcio_write (ipcio_t* ptr, char* buf, size_t nbytes);
```

- `ptr` is a pointer to a connected abstract ring buffer handle
- `buf` is a pointer to the data to be written
- `nbytes` is the number of bytes to be written
- RETURN value is the number of bytes written, or -1 on error

Data are read from the abstract ring buffer by calling

```
ssize_t ipcio_read (ipcio_t* ptr, char* buf, size_t nbytes);
```

- `ptr` is a pointer to a connected abstract ring buffer handle
- `buf` is a pointer to the buffer to be filled
- `nbytes` is the number of bytes to be read
- RETURN value is the number of bytes read, or -1 on error

A.4.2. Inheritance in C

The `ipcio_t` data type *is a* `ipcbuf_t`. That is, the pointer to the base address of an `ipcio_t` data structure can be passed, after casting, to all of the functions that receive a pointer to the base address of an `ipcbuf_t` data structure. This inheritance allows the `ipcio_t` data type to be treated as though it were a `ipcbuf_t` data type. For example,

```
#include "ipcio.h"
ipcio_t data_block = IPCIO_INIT;
ipcio_connect (&data_block, 0xc2);
uint64 nbufs = ipcbuf_get_nbufs ((ipcbuf_t*)(&datablock));
```

A.4.3. Writing before Start-of-Data

To begin writing to the Data Block before the actual start of valid data, it is necessary to call `ipcio_open` with the 'w' argument. The Write Client may then raise the start-of-data flag by calling

```
int ipcio_start (ipcio_t* ptr, uint64 st_byte);
```

- `ptr` is a pointer to a connected ring buffer handle
- `offset` is the byte offset after `ipcio_open`

To write an end-of-data flag without closing the abstract ring buffer, the Write Client must call

```
int ipcio_stop (ipcio_t* ptr);
```

- `ptr` is a pointer to a connected abstract ring buffer handle

The end-of-data flag is raised after the last byte written. As long as each call to `ipcio_start` is matched by a corresponding call to `ipcio_stop`, these functions can be called an arbitrary number of times between the call to `ipcio_open` and `ipcio_close`.

B. Header Block

In this chapter, the Header Block API is specified in detail. The Header Block is the buffer through which the Write Client communicates auxiliary information about the data stream to the Read Client(s). Access to the Header Block shared memory is controlled through the same semaphore-based interface as the Data Block. The major difference is that the Header Block ring buffer is opened only once, and each sub-block written to the ring buffer represents a unique header.

The Data Block API includes software for creating and initializing the shared memory and semaphore resources, locking the shared memory into physical RAM, connecting to the ring buffer, writing data to the ring buffer and reading data from the ring buffer.

B.1. Setting and Getting Header Values

Each sub-block of the Header Block ring buffer contains an ASCII text description of the data that is currently being written to the Data Block. An ASCII representation offers a number of significant advantages; it is human readable, and there is no need to worry about byte alignment and endian issues. Data attributes are stored as keyword-value pairs, each separated by a new line.

Given the base address of an ASCII header block, keyword-value pairs may be conveniently set and queried using the `ascii_header` API, which includes:

```
int ascii_header_set (char* header, const char* keyword,
                      const char* format, ...);
```

- `header` pointer to a null-terminated ASCII header block
- `keyword` the name of the attribute to be set
- `format` an fprintf-style formatting string
- `...` the value(s) to be printed following the keyword

and

```
int ascii_header_get (const char* header, const char* keyword,
                      const char* format, ...);
```

- `header` pointer to a null-terminated ASCII header block
- `keyword` the name of the attribute to be queried
- `format` an fscanf-style formatting string
- `...` the value(s) to be parsed following the keyword

B.2. Example

```
#include "ascii_header.h"

[...]

  char ascii_header[MY_ASCII_HEADER_SIZE] = "";

  char* telescope_name = "westerbork";
  ascii_header_set (ascii_header, "TELESCOPE", "%s", telescope_name);

  float bandwidth = 20.0;
  ascii_header_set (ascii_header, "BW", "%f", float);

[...]

  double centre_frequency;
  ascii_header_get (ascii_header, "FREQ", "%lf", &centre_frequency);

  float min, max;
  ascii_header_get (ascii_header, "GAIN", "%f %f", &min, &max);
```

Note that the formatting of a "value" is completely flexible; any number of values can be associated with a single ASCII header keyword. If the keyword does not exist when `ascii_header_set` is called, it will be added; if it does exists, its value will be replaced with the new value.

It is also important to note that in the current API, `ascii_header_set` has no idea about the size of the ASCII header. Therefore, it is up to the user to ensure that new attributes will not cause the length of the string to exceed that of the allocated memory.

B.3. DATA header parameters

The following header parameters will be included in the DADA ASCII header:

- `HEADER` name of the header
- `HDR_VERSION` version of the header
- `HDR_SIZE` size of the header in bytes
- `INSTRUMENT` name of the instrument
- `PRIMARY` host name of the Primary Node on which the data was acquired
- `HOSTNAME` host name of the machine on which data were written
- `FILE_NAME` full path of the file to which data were written
- `FILE_SIZE` requested size of data files
- `FILE_NUMBER` number of data files written prior to this one
- `OBS_ID` the identifier for the observations

- UTC_START the UTC of the first sample in the observation
- MJD_START the MJD of the first sample
- OBS_OFFSET the number of bytes from the start of the observation
- OBS_OVERLAP the amount by which neighbouring files overlap
- TELESCOPE
- SOURCE
- FREQ
- BW
- NPOL
- NBIT
- NDIM
- TSAMP
- RA
- DEC

B.3.1. Notes

The HDR_VERSION parameter should monotonically increase, and change only when a fundamental change in the interpretation of the header attributes is required.

The OBS_ID parameter must uniquely identify the data stream. As multiple data streams will be acquired simultaneously, the unique identifier will describe both the epoch/sequence number and the data stream/band.

The OBS_OFFSET parameter is added only after the data leaves the Data Block of the Primary Node. It counts the number of bytes offset from the start of the observation of the first byte of data described by the header. For instance, if the Read Client is writing files to disk, it would place in the ASCII header of each file written to disk an OBS_OFFSET equal to the number of bytes that preceded the first byte of data in the file. If the Read Client is writing data to a Secondary Node via a network interface, it would set OBS_OFFSET to the number of bytes that preceded the first byte of data to follow the header in the transmission.

The OBS_OVERLAP parameter is used by dbnic to determine the amount by which neighbouring data segments that are sent to different Secondary nodes should overlap.

C. Operational Logging

The various messages produced by the data acquisition software must be logged and/or communicated to possibly more than one listener. Therefore, all messages will be sent using the `multilog` API. A multilog session is opened by calling

```
multilog_t* multilog_open (const char* program_name, char syslog);
```

- `program_name` the name of the program will precede each log message
- `syslog` if non-zero, all messages are cc'd to syslog

and closed by calling

```
int multilog_close (multilog_t* log);
```

- `log` pointer to an open multilog session

Once opened, file streams may be added by calling

```
int multilog_add (multilog_t* log, FILE* fptr);
```

- `log` pointer to an open multilog session
- `fptr` pointer to an open file stream

Messages are written to all file streams (and syslog, if enabled) by calling

```
int multilog (multilog_t* log, int priority, const char* format, ...);
```

- `log` pointer to an open multilog session
- `priority` a `syslog` priority
- `format` an fprintf-style formatting string
- `...` the value(s) to be printed according to the format

Output messages are assigned a priority as described in the man page for the standard C `syslog` utility.

C.1. Example

```
#include "multilog.h"

[...]

  /* open a connection to syslogd using the standard C facility */
  openlog ("dada_dbdisk", LOG_CONS, LOG_USER);
```

```
/* open a multilog session that will use syslog */
multilog_t* log = multilog_open (1);

/* copy all messages to the standard error */
multilog_add (log, stderr);

/* write a message */
char* world_name = "Earth";
int world_number = 1;

multilog (log, LOG_INFO, "Hello %s %d", world_name, world_number);
```

D. Socket Communications

D.1. Examples

D.1.1. Example Server

```
#include "sock.h"

[...]

  char hostname [100];
  int port = 20013;

  /* Ask for the fully qualified hostname ... */
  sock_getname (hostname, 100, 1);
  /* ... or, ask for the IP address */
  sock_getname (hostname, 100, 0);

  int sfd = sock_create (&port);
  if (sfd < 0)
    perror ("Error creating socket");

  fprintf (stderr, "listening on %s %d\n", hostname, port);
  int cfd = sock_accept (sfd);
  if (cfd < 0)
    perror ("Error accepting connection");
```

The open file descriptor returned by `sock_accept` may be used in calls to the standard C I/O routines, `read` and `write` as well as `send` and `recv`. Furthermore, the open file descriptor can be converted into a stream by calling `fdopen`. It is important to note that sockets do not support seeking; therefore, a stream should be opened for only read or write access, never both. For example.

```
  /* two separate I/O streams are required for reading and writing */
  FILE* sockin = fdopen (cfd, "r");
  FILE* sockout = fdopen (cfd, "w");

  /* line buffer the socket stream output */
  setvbuf (sockout, 0, _IOLBF, 0);
```

The server can now read from `sockin` using standard C I/O stream routines such as `fscanf`, `fread`, and `fgets`. It can also write to `sockout` using `fprintf` and `fwrite`.

The call to `setvbuf` is important; without this call, it would be necessary to `fflush` the output stream to ensure that messages are communicated immediately.

D.1.2. Example Client

```
#include "sock.h"

[...]

  char* hostname = "apsr0.atnf.csiro.au";
  int port = 20013;

  /* Connect to the specified host and port */
  int cfd = sock_open (hostname, port);
  if (cfd < 0)
    perror ("Error opening socket");

  fprintf (stderr, "connected to %s %d\n", hostname, port);
```

As with the server, the socket file descriptor may be accessed using standard C I/O stream routines by calling `fdopen`.

E. Primary Write Client Command Interface

F. Primary Write Client Main Loop

In this chapter, the Primary Write Client Main Loop API is specified in detail. The Primary Write Client (PWC) Main Loop implements the PWC Data Flow Control described in Chapter 4.4. The PWC Main Loop is implemented as a function, `dada_pwc_main`, which receives a pointer to a struct, `dada_pwc_main_t`, as its only argument. The member variables of this struct must be properly set up before the Main Loop is entered, as described in the following section.

F.1. Initialization

The Primary Write Client Main Loop structure is created as follows:

```
#include "dada_pwc_main.h"
dada_pwc_main_t* pwcm = dada_pwc_main_create ();
```

After creation, the following member variables must be initialized:

1. `pwc` pointer to a `dada_pwc_t` struct; this struct is used to communicate between the thread that parses external commands from the Command Interface and the thread that runs the PWC Main Loop. The `pwc` member variable can simply be set as in the following example:

   ```
   /* create the command communication structure */
   pwcm->pwc = dada_pwc_create ();

   /* start the control connection server */
   if (dada_pwc_serve (pwcm->pwc) < 0)
     /* report an error message */
   ```

2. `log` pointer to the `multilog_t` struct that will be used for status and error reporting
3. `data_block` pointer to the `ipcio_t` struct that is connected to the Data Block
4. `header_block` pointer to the `ipcio_t` struct that is connected to the Header Block
5. `start_function` pointer to the function that starts the data transfer:

   ```
   time_t start_function (dada_pwc_main_t*, time_t utc);
   ```

   This function receives the UTC start time, `utc` at which the data transfer should begin. If `utc` equals zero, the function should start the data transfer at the soonest available opportunity. This function should take care of everything required to start the data transfer, and should return the UTC of the first time sample to be transfered or zero on error.

6. `buffer_function` pointer to the function that returns the next buffer to be written to the Data Block:

   ```
   void* buffer_function (dada_pwc_main_t*, uint64_t* size);
   ```

   This function should return the base address of the next buffer or the `NULL` pointer on error. The size of the buffer (in bytes) should be returned in the `size` argument.
7. `stop_function` pointer to the function that stops the data transfer:

   ```
   int stop_function (dada_pwc_main_t*);
   ```

   This function should perform any tasks required to stop the data transfer before returning to the idle state. This function should return a value less than zero in the case of error.
8. `context` [optional] pointer to any additional information. Should the implementation of any of the above three functions require access to other information, a pointer to this information can be stored in the `context` member variable and retrieved by casting this member inside the function. e.g.

   ```
   struct puma2_t {
     EdtDev* edt_p;
     pic_t pic;
     unsigned buf_size;
   };

   void* puma2_buffer_function (dada_pwc_main_t* pwcm, uint64_t* size)
   {
     struct puma2_t* xfer = (struct puma2_t*) pwcm->context;
     *size = xfer->buf_size;
     return edt_wait_for_buffers (xfer->edt_p, 1);
   }

   [...]

   struct puma2_t xfer_data;

   pwcm->context = &xfer_data;
   pwcm->buffer_function = puma2_buffer_function;

   [etc...]
   ```

G. Primary Read Client Main Loop

In this chapter, the Primary Read Client Main Loop API is specified in detail. The Primary Read Client (PRC) Main Loop implements the PRC Data Flow Control. The PRC Main Loop is implemented as a function, `dada_prc_main`, which receives a pointer to a struct, `dada_prc_main_t`, as its only argument. The member variables of this struct must be properly set up before the Main Loop is entered, as described in the following section.

G.1. Initialization

The Primary Read Client Main Loop structure is created as follows:

```
#include "dada_prc_main.h"
dada_prc_main_t* prcm = dada_prc_main_create ();
```

After creation, the following member variables must be initialized:

1. `log` pointer to the `multilog_t` struct that will be used for status and error reporting
2. `data_block` pointer to the `ipcio_t` struct that is connected to the Data Block
3. `header_block` pointer to the `ipcio_t` struct that is connected to the Header Block
4. `open_function` pointer to the function that opens the file descriptor to which data will be written:

   ```
   int open_function (dada_prc_main_t*);
   ```

   This function should initialize the following member variables of `dada_prc_main_t`:

   - `fd` file descriptor to which data will be written
   - `transfer_bytes` total number of bytes to write
   - `optimal_bytes` optimal number of bytes to write each time
   - `header` [optional] header to be written to the file descriptor

   The `header` member variable will be initialized before calling `open_function` and will be written to the file descriptor after this function returns. This function should return a value less than zero in the case of any error.

5. `close_function` pointer to the function that closes the file descriptor after data has been written:

   ```
   int close_function (dada_prc_main_t*, uint64_t bytes_written);
   ```

   The `bytes_written` argument specifies the total number of bytes successfully written to the file descriptor. This function should perform any tasks required to close the file descriptor and return a value less than zero in the case of any error.
6. `context` [optional] pointer to any additional information. Should the implementation of any of the above functions require access to other information, a pointer to this information can be stored in the `context` member variable and retrieved by casting this member inside the function.

H. Testing the DADA Software

This chapter describes the various programs that have been designed to facilitate testing and development of the DADA software

H.1. Primary Write Client Demonstration, dada_pwc_demo

The Primary Write Client (PWC) Demonstration program, `dada_pwc_demo`, implements an example PWC interface. It does not actually acquire any data and therefore can be run on any computer. This program has two modes of operation: free and locked.

In *free* mode, `dada_pwc_demo` does not connect to the Header and Data Blocks; therefore, it is not necessary to create the shared memory resources and run a Primary Read Client program. This mode is most useful when testing the command interface and state machine of the Primary Write Client. To run in *free* mode, simply type

```
dada\_pwc\_demo
```

In *locked* mode, `dada_pwc_demo` connects to the Header and Data Blocks; therefore, it is necessary to first create the shared memory resources and also run a Primary Read Client program, such as `dada_dbdisk`. This mode is most useful when testing the interface between Primary Write Client, Header and Data Blocks, and Primary Read Client software. To run in *locked* mode, for example

```
dada\_db -d         # destroy existing shared memory resources
dada\_db            # create new shared memory resources
dada\_pwc\_demo -l   # run in locked mode
```

The first step is particularly useful when debugging. In another window on the same machine, you might also run

```
dada\_dbdisk -WD /tmp
```

To connect with the PWC demonstration program and begin issuing commands, simply run

```
telnet localhost 56026
```

or replace `localhost` with the name of the machine on which the program is running.

As described in Section ???, it is necessary to enter the **prepared** state before recording can begin. This is done by issuing the **header** command.

H.1.1. Primary Write Client Command, `dada_pwc_command`

It is also possible to control one or more instance of `dada_pwc_demo` using the Primary Write Client Command program, `dada_pwc_command`.

This program takes a specification file and creates a unique header for each primary write client.

Therefore, it requires a configuration file that specifies the number of primary write clients, the port on which they are listening, and the full path to a file that lists the parameters that should be copied from the specification to the header block of each primary write client.

A sample configuration file can be found in config/test_dada.cfg. Make a copy of this file and edit the parameters appropriately.

The SPEC_PARAM_FILE parameter must list the full path to config/specification_to_header.txt, which should not require editing.

By default, `dada_pwc_command` uses the same port number (56026) as `dada_pwc_demo`. Therefore, if you are running both programs on the same machine, it will be necessary to specify a different port number for the command interface; e.g.

```
dada\_pwc\_command -p 20013
```

The PWC command program must be configured using a specification file; an example can be found in config/test_specification.txt. Tell `dada_pwc_command` to use this file using the "config" command, giving the full path to config/test_specification.txt.

Example: `dada_pwc_command`

Suppose that you want to simulate control of two primary write clients on two different machines, named dada01 and dada02:

1. Copy config/test_data.cfg to $HOME/test_my.cfg
2. Copy config/specification_to_header.txt to $HOME/spec2hdr.txt
3. Copy config/test_specification.txt to $HOME/spec.cfg
4. Set the environment variable, DADA_CONFIG to $HOME/test_my.cfg
5. Edit the file, setting `PWC_0` and `PWC_1` to `dada01` and `dada02`, and SPEC_PARAM_FILE to $HOME/spec2hdr.txt [you will have to expand the shell variable $HOME]
6. run `dada_pwc_demo` on dada01 and dada02
7. On any machine with network access to dada01 and dada02, say dada, run `dada_pwc_command -p 20013`
8. On any machine with network access to dada, run telnet dada 20013
9. Enter the specification, `config $HOME/spec.cfg` and note the action on dada01 and dada02, ending with `STATE = prepared`
10. Type `start` and note the `Start recording data` messages on dada01 and dada02
11. Type `stop` and return to `STATE = idle`