Pipeline Processing

Rick Hobbs

Revision: 1.11
Date: 2005/01/29 00:24:12

1 Functionality and Scope

1.1 Overview

Pipeline Processing will involve various stages of data manipulation starting with the collection of data from the correlator bands to the output of data files. Input rates from the digital correlator will be 2 Hz and output into flat files will be controllable with the maximum rate being 2 Hz. Control of the processing stages will occur directly from the control system. Blanking and flagging information will be received from the Fault System. A list of options will be described in the Control Input section.

The processing stages implemented in the Pipeline will be as follows:

1. Collect data from each correlator DO and combine them into a single data object(finished) - class name: CatchData
2. Data decimation/windowing - class name: Decimator
3. Correct for passband gain - class name: PassBand
4. Apply Tsys - class name: Tsys
5. Convert to flux units - class name: Flux
6. Blank or flag data - class name: BlankFlag
7. Linelength delay correction - class name: LinelengthCorrection
8. IF delay correction - class name: IFcorrection (This may not be significant enough to worry about)
9. Apply atmospheric delay correction - class name: Wvr
10. Integrate data to requested astronomical integration time. Create additional band consisting of band averages. This processing stage needs to know which contiguous frequency channels to use for the band average. - class name: Integrator
11. Write visibilities to files in Data Brick format. This processing stage may need to know which frequency channels to write. - class name: VisBrickWriter
12. Also, continuum data published in the monitor stream. - class name: WbPipelineSubsystem

1.2 Distributed Objects

The Distributed Objects(via CORBA) that will be available to other systems are described in Table **??** as well as those objects needed by this package.

Table 1: Pipeline Distributed Objects

DO	Function	Used By	Expected From
WpipelineControl SpipelineControl	Controls how data are processed through pipeline stages.	Control	
carma.correlator.Wband1-N carma.correlator.Sband1-N	Dishes out correlator data	This package	Digital Correlator
Logging	Log errors and any startup info	This package	Logging Services
Monitor	Pipeline processing status	This package	Monitoring
FaultSystem	Contains blanking/flagging information	This package	FaultSystem

1.3 Coding Requirements

The software framework from the existing Wideband Pipeline used for COBRA can partially be reused and a new data object will need to be constructed. New code will be required to handle the new pipelineControl API.

1.4 Outstanding Issues

1. Would like to have another 'pipeline' machine to run a parallel development Pipline Processing stream.
2. More design work needed to better specify control/functionality requirements for each stage. Algorithms/equations need to be written for each stage and reviewed.
3. Pipeline interaction with other parts of the system such as the monitor averager. Need for an integration number.
4. Need to specify blanking/flagging object sent by Fault System. Mode of transmission - I'm now leaning towards Notification instead of DO method call.
5. Do we need IF delay correction
6. Is there a need for a continuous non-decimated pipeline or is the on/off flag sufficient?
7. Performance for CARMA type correlator will need to be looked at. Number of Bands = 8, Number of Channgels = 256.

1.5 Administrative Summary

1.5.1 Estimated FTE Effort

The tasks include purchasing and configuring of hardware as well as the standard software development cycle. The effort below is in FTE months.

Computer purchase/setup:	1.0
CDV upgrade:	1.0
Design:	1.5
Implementation:	1.5
(Estimate # of lines of new code)	2700
Testing:	2.0
Integration:	2.5
TOTAL:	8.5

1.5.2 Schedule

Package status:	open	
Conceptual design:	completed:	2003/08/27
Preliminary design:	completed:	2005/01/27
Nominal closing dates:	Winter 2004	(SZA)
	Spring 2005	(CARMA)

2 Design

The software design can be broken up into a part which requires no control and a part which does. The no control part consists of the data combiner whose only function is to collect data from each individual correlator object and merge it into a single data object. This break is consistent with SZA needs as they will pick up the combined object at this point in the processing stage. This stage will most likely be the one to publish the Control DO however.

The control DO will manipulate the functionality of items 2-11 in section 1.1 above and will be handled via the following API and implemented using CORBA. A single CORBA object will be instantiated for each subarray and will be identified as WpipelineControl for Wideband Pipeline Control and SpipelineControl for Spectral Pipeline Control.

A single process model is beig proposed to facilitate a clean messaging protocol between the DO and ojects in the pipeline chain (ie. object method calls). This will eliminate the need to attach a control structure to the CorrelatorData object which is also being published for CDV. Therefore, I'd like to keep it as small as possible. This single process may start after the catchData process in order to leverage off IPQ alignment between incoming correlator data and blanking information. In this case, the pipeline will become a 2 process model. One process catches data from the Correlators and blanking/flagging information from the fault system and the second process listens for control information and processes the time aligned data. Each stage of the pipeline will output results to an IPQ in the format of a CorrelatorData object to allow publishing for CDV as well as for noninvasive interrogation.

The pipeline, like the Correlator, will initialize itself from a configuration file. Some parameters will be changable in real-time in order to study performance characteristics. For example, the time-out period for catching bands can be modified in real-time to study how this influences the percentage of caught bands.

2.1 Control Input

Each Pipeline Processing object will implement the following methods:

- startIntegration(integTime, numRecords)
- stopIntegration()

- applyPassbandCorrection(on/off/noise)
- applyTsys(true/false)
- applyFlux(true/false)
- applyAtmosphicDelay(true/false) (May not need this since 2 parallel streams will be processed)
- applyBlanking(true/false)
- Decimate(true/false) (May not need this since 2 parallel streams will be processed)
- addChannelAverage(bandNumber, startChannel, endChannel) (Only 1 range will be allowed and applied for all baselines)

Tsys processing will require additional information from control:

- calibrator position[ant] (ie. SKY, AMBIENT, FIXEDTEMP, PARTIAL)
- downconverter total Power[band,ant,sb]
- Outside Air Temperature?
- Sky freq., LO freq.(if correct Sideband is still unclear)
- Sideband ratio[band?, baseline] composed of receiver and atmospheric sideband ratios.
- Kelvin to Jansky Conversion needs: Area[ant], illumination effeciency, surface error[ant]. Surace error is used to compute Ruse efficiency[f]. Geometric mean used.

2.2 Data Output

Data headers(ie. fields pertaining to processing) will be stored in the CorrelatorData object along with the visibilities and written into files using the Data Brick format. Currently this format is just the CorrelatorData object written as separate binary records. Continuum data at a 2 Hz rate will be published in the monitor stream. The integrated continuum data will also be published in the montior stream.

2.3 Monitor Output

Each stage will be capable of publishing monitor points to indicate their status as listed in the following sections.

2.3.1 Data Collector

Monitor Point	Description
numBandsCaught	Number of Correlator Bands Caught for current half second.
bandsMissed	band numbers missing for current half second.
pctCaughtTot	Total % of data caught since start of process.
pctCaught[band]	% caught as a function of band.

2.3.2 Decimation/Windowing

Monitor Point	Description
state	on or off.
window	window type.
finalChannels	final number of channels.

2.3.3 Passband Correction

Monitor Point	Description
state	on or off.

2.3.4 Tsys

Monitor Point	Description
state	on or off.
spectral	yes or no.
Tsys	(DSB, LSB, USB)[band,baseline,sb]
lastMJD	last time a valid tsys was applied.

2.3.5 Atmospheric Delay

Monitor Point	Description
state	on or off.
delay[ant]	Amount of delay being applied.

2.3.6 Integrator

Monitor Point	Description
state	on or off.
mode	blanking or flagging.
integrationNumber	unique number for this integration.
numRecords	Number of records being integrated.
desiredIntegTime	Desired integration time.
nominalIntegTime[band,baseline]	Nominal integration time.
pctTime[band,baseline]	% of integrated time in record. (actual/desired).
blankedTime[band,baseline]	amount of data blanked in msec
flaggedTime[band,baseline]	amount of data flagged in msec

2.3.7 VisBrick Writer

Monitor Point	Description
integrationNumber	current integratin number
filename	current filename being written.

3 Implementation

3.1 Data Collector

The non control functionality consists of only 1 stage which collects data from each correlator band and combines it into single data object as well as collecting blanking/flagging information from the fault system. This will most likely run as a separate process.

Catching and assembling Correlator data from the various Correlator crates is the responsibility of the first stage in the pipeline named catchData. Four classes make up this core functionality:

- CatchData.h[cc]
- DataCollectorN.h[cc]
- BfCollectorN.h[cc]
- DataContainer.h[cc]

CatchData is the main coordinator. It's job is to start up a timer to force a time limit for incoming data into the DataContainer. All data written into the DataContainer within the specified time, will be written to an IPQ. Likewise, Blanking and Flagging information will be received and written to a separate IPQ. CatchData also serves as the object for registering which bands to collect and has the following API:

- absTimerEvent()
- collectBand()
- collectBF()
- run()

DataCollectorN is an object which connects to a specified DO and waits for incoming data. Arriving data are added to the DataContainer object. Each DataCollector object runs in a separate thread. It is designed to handle reconnections to the DO. DataCollectorN has the following public API:

- getRemoteObject()
- processData()
- getName()
- getTypeId()

BfCollectorN is an object which connects to a specified DO and waits for incoming Blanking/Flagging data. Arriving data are written to a separate IPQ. This object runs in a separate thread and is also designed to handle reconnections to the DO. BfCollectorN has the following public API:

- getRemoteObject()
- processData()
- getName()
- getTypeId()

DataContainer is a Singleton Object with mutex locking which is shared between CatchData and the DataCollectorN objects. DataContainer has the following API:

- getCorrelatorData()
- fillCorrelatorData()
- clearCorrelatorData()

3.2 Decimator

Decimation/windowing is handled in a single class. It implements the CorrelatorListener interace and has the following public API:

- decimate(yes/no)
- processData()
- keepEndChannels(yes/no)
- finalNumberOfChannels()

3.3 PassBand

TBD

3.4 Tsys

Tsys calibration's main class will be called Tsys.h[cc]. It implements the CorrelatorDataListener interface and has the following public API:

- processData()
- calibrate(yes/no)
- setTsysParameters() sets an object containing necessary values for cal.

3.5 Flux

Flux converts from Kelvin to Janskys. It implements the CorrelatorListener interface and has the following public API:

- processData()
- applyFlux(yes/no)
- setFluxParameters() sets an object containing necessary values for computation.

3.6 Blanking/Flagging

Blanking/Flagging will be performed in BlankFlag.h[cc]. It will implement the CorrelatorListener interface and has the following public API:

- processData()
- applyBlanking(yes/no)
- setBfParameters() sets an object containing necessary values for computation.

3.7 Atmosperic Delay Corrections

Atmoshperic delays will be performed in Wvr.h[cc]. It will implement the CorrelatorListener interface and has the following public API:

- processData()
- applyWVR(yes/no)
- setWVRparameters() sets an object containing necessary values for computation.

3.8 Integrator

Integration is handled with a single class called CorrelatorIntegrator.h[cc]. It implements the CorrelatorListener interface and has the following public API:

- processData()
- setIntegrationRecrods()
- startIntegration()
- stopIntegration()
- getName()

3.9 VisBrick

Visibilities will be written to flat files using VisBrickWriter.h[cc]. It will implement the CorrelatorListener interface and has the following public API:

- processData()
- writeVisBrick(yes/no)

4 Performance

Table **??** shows timing results for the catchData process catching various number of bands on different platforms. The CPU increase for the Hypertreaded dual CPU platform has been identified as being due to contention in the sub-allocator. The current code contains an additional integer vector in the CorrelatorSideband ojbect which keeps track of channel validity. For 17 channels, this vector is less than 128 bytes which causes the sub-allocator to kick in thus causeing poor performance. Increasing the number of channels to 33(giving 132 bytes) improves performance as compared to using 32 channels as shown in Table **??**.

Table 2: Catch Data Performance

	HT dual 3.0GHz		Single 1.9GHz Athlon	
Bands	CARMA_R1 [%]	Current Code [%]	CARMA_R1 [%]	Current Code [%]
1	.8 - 2.1	.5		.3
5	4.5	6.2		1.5
10	8 - 10	16 - 18		3.5
16	17 - 18	28 - 32	5	6

Table 3: Catch Data Sub-allocator Performance

Bands	Channels	Size of vector<int> [Bytes]	CPU Usage [%]
16	32	128	30
16	33	132	20