VLBA Correlator Memo No. 95

Software Architecture for the VLBA Correlator

D. Wells, J. Benson, C. Broadwell, J. Horstkotte, J. Romney
National Radio Astronomy Observatory, Charlottesville, Virginia

September 29, 1989

Abstract

The architecture for the tasks, shared data structures and job scheduling in the VLBA correlator is described. Relations in a commercial database management system describe all aspects of the observations and data. Job scripts using a table syntax are generated from the relations. The data structures for the real-time system are loaded from the scripts. Real-time tasks control job processing, tape drive actions, hardware configuration, model tracking, and archiving. The archive is written in a near-distribution format, FITS except for blocking and error-recovery features; automatically scheduled jobs extract users' data for conversion to distribution media.

Contents

List of Figures

List of Tables

1 Introduction

The software design for the VLBA correlator is presented in this document in three major sections. Several appendices include further detailed information, including various script formats and table definitions.

First, in a comprehensive overview, the architectural strategies are elaborated in Section 2. A commercial database management system (DBMS) and a job generator with associated "batch" tasks comprise the "correlator control" system which runs in a general-purpose computing environment. In the real-time system, actual correlator operation is supported by the job control and scheduler tasks and by their many subordinated tasks, either dedicated to particular hardware elements or associated with transient jobs. SQL queries embedded in the job generator code access the DBMS. The job control task receives a human-readable script from the job generator, and constructs globally shareable tables accessible to the real-time tasks.

Sections 3 and 4 discuss the most important tasks in the correlator-control and real-time systems. The level of detail in the descriptions reflects the current state of completion of these tasks. The job loader and model/CALC tasks have been coded and are near final form. The hardware control bus (described in Appendix I) is already being used in prototype form with test fixtures for the correlator's printed-circuit boards; the low-level functions used in this work will evolve into the real-time drivers for the bus. The remaining tasks are in various stages of planning and implementation; important aspects of each are described in enough detail to present a functional description of the individual tasks and of the software architecture design as a whole.

Each individual section is intended to be relatively self-contained, and redundant expository material has been included where necessary to achieve this goal, with liberal cross references pointing to additional detail on major concepts. The authors have found it necessary to adopt rather precise language to manage the conceptual diversity and complexity embodied in this document, and it has not been feasible to cross-reference this terminology. Accordingly, a glossary is provided in Appendix A, which the reader is advised to consult when encountering unfamiliar vocabulary or usage.

2 Architectural Strategies

The objective of this chapter is to provide background and justification for the principles and goals that guide the detailed design of the correlator software. In some places this strategy discussion becomes more detailed than would normally be appropriate because the corresponding portions of the detailed design have not yet been done.

2.1 Database Management System

VLBA Memo 469[1] made the case for a commercial relational database management system (DBMS) for the VLBA; Memos 485[2], 568[3] and 569[4] have added weight and detail to the arguments. The DBMS is the foundation of the software architecture of the correlator, the unifying concept that pervades and determines almost everything else.

The DBMS will support the full "relational" calculus (see Memo 469), the ANSI-standard SQL (Structured Query Language) user interface, with an "embedded" SQL extension language for application programs coded in C.

The VLBA Computer Coordination Group (Memo 497[5]) discussed the desirability of choosing the VLBA DBMS technology such that it had the possibility of being selected as an Observatory-wide solution. Recently client-server architectures in DBMS packages have shown signs of becoming standardized. A key component of these architectures is the use of SQL as the client-server interface; the user interface software (the client layer) need not be made by the same manufacturer as the DBMS server software. The client layer can even be on separate machines under different OSs, e.g. DOS or VMS. The vendor-supplied automated mechanisms of a typical DBMS can move information from one server to another almost transparently. Because of the long-term potential for Observatory-wide DBMS functions the correlator utilizes a DBMS which has this "distributed" capability.

---

[1] VLBA Memo 469, "The VLBA Database", Jonathan D. Romney, 23 July 1985.

[2] VLBA Memo 485, "Selection Criteria for the VLBA Database", Jonathan D. Romney, 9 September 1985.

[3] VLBA Memo 568, "Relational Databases and VLBA Operations, or, Is the VLBA ready for MIS?", Martin Ewing, 7 August 1986.

[4] VLBA Memo 569, "Relational Databases and VLBA Operations, Revisited", Martin Ewing, 15 August 1986.

[5] VLBA Memo 497, "Review of 24 Sept. 1985 Meeting", Craig Walker, 25 September 1985.

The query tools provided by the DBMS vendor will assist in constructing applications for entering, editing and displaying information in the DBMS. The application-building tools, often called "4GLs" (Fourth-Generation-Languages), appear to be likely to facilitate development of such applications.

The DBMS holds *tables* of information. The SQL language manipulates these tables using what is essentially a notation of sets and set operators. This notation is terse and powerful; most algorithms coded in this notation take *much* less text, and therefore much less effort to code and debug, than if they were written in C (a "third-generation-language") to access the same information stored as ordinary data files on the disks. It is difficult at this time to estimate the cost advantage of SQL over C (in programmer-time); it may be as much as an order-of-magnitude. Not only will the gains be realized during construction, but they will also accrue over time as decreased maintenance costs.

Logging information flowing back from the real-time system to the DBMS will be encoded as SQL `UPDATE` and `INSERT` transactions. These can be processed by a batch task as "dynamic" embedded-SQL operations. This means not only that the logging information is a plain ASCII text file, but also that the batch data-entry/transaction task will not have to contain code, in the form of SQL commands coded as `sprintf` statements, for the endless variety of transactions which we may wish to invoke. This approach is so attractive that it may also be advantageous for the VLBA array control system to output its logs as SQL transactions.

The DBMS need not have particularly high performance; typical transaction rates in the correlator system are likely to be measured in units of transactions-per-minute rather than in transactions-per-second. The dynamic SQL transactions discussed in the previous paragraph will need to be compiled and optimized at execution time, but at the low transaction rates expected this will probably not create an unusual loading on the DBMS server.

2.2 Tuples-Scripts-Structs

> "Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious."[6]

[6] F. P. Brooks, Jr., "The Mythical Man-Month (Essays on Software Engineering)", 1975, p. 102, in a section titled "Representation is the Essence of Programming".

The relations in the Database Management System will contain complete descriptions of all aspects of all observations made by the VLBA (and other stations jointly observing with the VLBA), and the sequences of commands and parameters for the processing of the data for these observations will be generated from these relations. It is desirable that correlator processes receive their marching orders by means of a human-readable script, rather than by making real-time queries of the DBMS server, which will be running in the general-purpose environment. It is also desirable that logging output from the real-time (RT) system pass back to the DBMS on the GP system in the form of a script.

We are considering the interface between two large subsystems of the correlator (RT part versus GP part) which will be built by somewhat different people on somewhat different schedules, using somewhat different technology. The main reason for wanting plain-ASCII scripts for this interface (command input and logging output) is that during development of the RT system test scripts can be built and used even if the GP system is not ready to generate scripts, and logging output properties can be verified before the GP system is ready to accept logging. The GP system will gain the analogous advantages during its (decoupled) development.

The script input interface will not be used in production for people to generate or edit jobs for the correlator. Instead, hand-editing will occur in the DBMS, by changing values in the attribute fields using the front-end tools that all DBMSs have. The automatic job generator will then produce a new job script which will pass to the correlator automatically.

The script will be a succession of keyword-equals-value pairs, and will use the syntactic conventions of the array control script mechanism for data values. The notation will be a concise representation of the contents of the relevant relations in the DBMS, with the separate relations delimited, and the tuples within those relations also delimited. The relations and keywords needed, and the precise notation of the script language, are discussed in Section 4.2 and a sample script is shown in Appendix B.

The attributes (columns) of the relation (table) have names. Keyword-equals-value pairs in the script will be generated mechanically (i.e., transliterated) from the names and values of the attributes, probably one pair per line. Each new tuple will be signaled by a "`!row!`" command. When the generator processes a new tuple it will only emit those pairs for which the value has changed since the last tuple. Note that because a relational DBMS has system directory (metadata) relations that contain the description of the attributes of all other relations, and because these descriptive relations can

also be queried in the standard manner, the generator can be a generalized utility, knowing nothing about the applications.

The script and the DBMS relations are completely equivalent notations. Another equivalent notation is the FITS tables extension formats (the IAU-standard ASCII version or Cotton's "3-D" binary table design). The script format will be used for the GP/RT interface, while Cotton's (FITS) table format will be used to store these tables, as well as other tables, in the archive.

When the script files are processed by the RT system it will generally be true that the keyword-equals-value pairs of the script will be stored into instances of C data structures in our application code, with one instance per tuple. Thus, discussing the schema for the correlator DBMS is nearly equivalent to discussing the C-structures of our RT application.

Mechanical translation of tuples into script code does not imply that the two parts of the project must have identical naming conventions. They don't, because of the availability of the "view" mechanism of relational DBMSs. A view is a virtual relation, and the attribute names of the relation may be different from those of the underlying relations from which the virtual relation is synthesized. This view mechanism also means that the array control group and correlator group are not obliged to agree completely on the normalization and naming conventions of their relations, which may be in different servers. Of course minimization of the entropy of our name-spaces and architectural concepts is desirable, and so we should attempt to coordinate our subsystems, but it is also true that, as Frost said, "good fences [interfaces] make good neighbors".

Every aspect of what the correlator does will be driven by what the scripts contain, and the scripts will be driven by what the DBMS contains. Therefore the DBMS schema must ultimately be a representation of everything that the correlator, the array, the operational staff and the end-users are doing in the observational process. It is in this sense that "the DBMS is the foundation of the software architecture of the correlator, the unifying concept that pervades and determines almost everything else".

2.3 Quantum of Work

A fundamental principle of the design of the correlator software is that the overall system should be as "robust" as possible. Not only should the system initiate processing jobs automatically, without any human intervention, but it should also recover gracefully from a wide variety of events which will

occasionally interrupt normal processing. In particular, *it should automatically re-initiate processing with as little redundant work as is possible.* In addition, the system should positively verify the integrity of archival data *before* raw data tapes are released for re-use.

These desiderata lead to the notion of identifying a "quantum of work". The output quantum of the correlator will be a physical block, about one-half megabyte in size, containing data for a single project. As each block is written a logging message will be generated to record the time of the last visibility in the block. This logging message will eventually cause the time to be written into a field in the DBMS tuple associated with the array of the project. The amount of work represented by the block (the amount which can be lost in a crash or other interruption) will depend on the number of baselines, spectral resolution and integration time.

If the power fails, or the operator peremptorily halts processing, or any other cause halts processing, the last logging message will eventually arrive back at the DBMS and will update its field. The RT system will generate another logging message for any job which was in progress at the interruption in order to mark the overall job as aborted (this message can be generated after a boot by checking a local "jobs-in-progress" disk file which will survive the crash). In any case, a time-out rule will apply: any job for which no logging messages are received to mark it as either complete or aborted will be considered to be aborted, and will be regenerated. The new script for an aborted job will specify processing only for the time range not already processed.

2.4 Job Scheduling

Two principles of scheduling jobs in the correlator are

- *observations should be processed as soon as possible after their tapes become available and*
- *the computing resources of the correlator should be utilized as fully as possible.*

The real-time job loader task will produce a queue of job structures which are to be processed. The first principle implies that this queue should be time-ordered, and that the oldest job should be processed first. The second principle implies that successive jobs should normally be processed in time order, as long as they represent contiguous blocks of observe time,

because this minimizes tape changing overhead, except that jobs must be run in parallel whenever doing so will not imply excessive tape changing. These two desiderata (in time order versus in parallel) are contradictory, and finding the optimum balance between them is what makes the scheduling problem of the correlator "interesting".

A *job* will consist of observations made for one or more *projects* during some range of time. Each such project consists of one or more *arrays*. Each array is a set of stations observing the same object, for a single project, with compatible observing parameters, over some range of time; individual stations may join or leave randomly. The duration of a job will be sufficient to minimize tape change overhead, but short enough to allow the job scheduling algorithms in the RT system to have some flexibility.

2.5 Automated Processing

A fundamental assumption behind the architectural plan is that the normal mode will be automated processing of observational data. "Automated" in this context includes correlation of observations exceeding the correlator's station or channel capacity by making multiple passes, which will be submitted automatically to the RT system as separate jobs. In general, input tapes will be recycled as soon as the archived results have been verified (by automatic processes). In order to permit special cases to be handled a flag will be provided to inhibit the recycling of data tapes, so that new parameters can be entered into the DBMS manually to cause new jobs to be queued for re-processing the data tapes.

2.6 Clocks

The fundamental principle is that clock offsets needed for any regular processing job must be known to an acceptable tolerance before that job may be submitted to the queue of the real-time system. The intent is to avoid wasted processing time due to improper delay models caused by improper clock estimates.

If at all possible the correlator system will avoid requiring operator intervention to determine proper clock offsets. The clock predictor batch job will apply a predictive filter, probably with constraints, to available clock measurements, producing estimates of future offsets and derivatives, and their errors. From these results the job submission batch process can determine whether the clock model has acceptable uncertainty for each observation.

This fitting process will run at regular intervals.

The clock measurements will come from several sources:

- GPS and/or Loran measurements at some stations
- Real-time fringe checks for VLBA stations
- Calibrator observations from regular processing
- Calibrator observations from special clock jobs

The predictive filter batch job will utilize any and all of these data types which may be available, with appropriate weighting factors. The job will insert computed clocks tabulated at regular intervals into the database for use by the job submission batch job.

What will happen if a station clock "glitches"? The real-time fringe check observations, and GPS or Loran measurements, will produce offset estimates sufficient to recognize that a glitch has occurred. These estimates will be available several days before the associated tapes arrive at the AOC for regular processing. The clock predictor program will recognize that a probable glitch has occurred. It will insert predicted clock values with large error estimates in the database beyond the point of the glitch; this will inhibit job submission beyond the last good clock measurement. The fitting program will also (automatically) submit a special clock job for that station as soon as the required tapes have arrived at the AOC. The special job will involve the station in question plus one or more other stations with well-known clocks. The jobs will be relatively cheap to process because only the calibration observations on the tapes from those stations will need to be analyzed, and the PBDs will slew with average latencies of about 4 minutes.[7]

Results from the clock analysis jobs (either special or regular processing) will come back to the database from the real-time system and will be analyzed in the next execution of the clock predictor batch job. Note that the choice of offset to be used during the interval between two calibrator observations which bracket a glitch may require operator intervention. Indeed, although it is highly desirable that clock analysis proceed in a "hands-off" fashion, human intervention may be necessary to make decisions and resolve ambiguities in a variety of cases.

The clock strategy assumes that the clocks at the VLBA stations will be generally well-behaved on a time scale of about 24 hours. Calibration

[7]The tape length and the slew speed of the PBDs will result in end-to-end time of about 12 minutes.

observations which are being correlated during regular processing will be analyzed to determine clocks for all VLBA stations. These offsets will be entered into the database and used to extrapolate into the future in order to submit subsequent jobs. Normally only non-VLBA stations with no recent clock history in the database will need the special clock analysis jobs.

2.7 Archive

The archive will be in a near-distribution format, essentially a multi-source *uv*-FITS format. The only differences from the distribution format will be large block sizes, interleaving of blocks from various observations, and a redundant-block scheme for error correction. This means that the distribution writer task will have an easy job — the heavy lifting will be done by the archive task (with the aid of special hardware associated with the Long-Term-Accumulator). The purpose of these policies is to assure that the archive format is independent of the software and hardware of the correlator real-time system in order to assure portability.

The archive data will be *completely self-documenting:* all values in the DBMS which might ever be needed in the processing or interpretation of the data will be recorded in the FITS data structures. The intent of this policy is to assure that the completeness of the archive format does not depend on the continued existence of the DBMS, and also to assure portability.

All blocks in the archive will be the same length, with the length chosen to achieve high packing efficiency on whatever medium is selected; probably the length will be 576 kB/block (200 FITS logical records of 2880 bytes). A redundant block scheme will be employed to protect against checksum errors in the blocks: for every N (say 10) data blocks written an extra block will be written with the exclusive-OR of the $N-1$ preceding blocks. This will be a defense against unrecoverable read errors in a physical block; it is expected to be a rare occurrence because archive devices defend the individual physical blocks with elaborate, robust error-correcting codes.

Each data block in the archive will be a complete, self-contained FITS file. Sufficient information will be present to reconstruct the catalog of observations from the archive media alone if necessary. Blocks will be numbered consecutively on each volume. The VSN (ANSI volume serial number) and block numbers will be tabulated in the DBMS for each processed observation for use by the distribution task.

The job generator will submit an archive verification job to the RT system for each archive volume which is recorded. Such a job cannot execute

until its volume has been filled. It will read every bit on the volume, rigorously verifying the integrity of the data and its conformance to FITS syntactic rules as much as possible. Each block which has been verified will result in an update transaction in the DBMS to mark the block as "done and verified". Bad archival recordings will be checked to decide whether a re-run should be submitted, or the operator alerted to an unusual condition. The release of the input data tapes will be inhibited until verification is complete.

The desire that the archive should be completely self-contained leads to the notion that the DBMS relations which represent catalogs of archival data should also be written into archive volumes. FITS tables extensions are completely satisfactory for this purpose of course. The issue of whether old observation catalogs should be retained on the disks of the DBMS server forever or rather stored in the archive itself (with batch jobs to fetch tables back to the DBMS) will be considered later in the project.

Another strategic issue is whether the VLBA archive should contain the maps (post-processing results) in addition to the visibilities produced by the correlator itself. It appears that by the mid-90's bandwidths available over the LAN in the AOC, and often even over wide-area networks, will make it practical to offer an archiving service for post-processing results. The design of the archive-writer, DBMS schema and distribution service will not preclude implementation of such a service.

Most discussions of archives these days presume that some sort of robotic mechanism (a "jukebox") will be used to manipulate archive volumes. *The VLBA correlator archive plan is based on the assumption that an operator will be available to mount any archive volume within minutes of the posting of a request at any time, 7 days a week, 24 hours per day.* This "human-jukebox" assumption is the only means by which an archive of nearly unlimited capacity can be constructed and operated within the current budget.

It is not possible to predict at this time what devices will be best for the archive medium. It is not even possible to predict whether the device should be optical or magnetic. Preliminary cost estimates indicate that magnetic media are cheaper if rapid random access is not required (it probably isn't in this application). Probably one of the magnetic cartridge media (4, 6, 8 or 12 mm width) will prove to be best. It is fascinating that a direct lineal descendent of Mark II VLBI techniques, the Digidata VHS cassette technology, appears to be the cheapest archival medium at the present time. We will defer the procurement of the archive hardware as long as possible in order to take maximum advantage of hardware evolution.

Output recording capability of 0.5 MB/s is specified for the VLBA correlator in order to support processing of worst cases requiring high spectral resolution covering wide fields of view, notably observations of water masers in Orion A. Therefore, it is desirable that the archive writing devices have sustained data rates for long blocks of at least 0.5 MB/s. Some of the devices available at present cannot sustain this rate (e.g., the new Sony DAT drive appears to be limited to 183 kB/s); "striping" could be used if we choose one of these devices for some compelling reason. An alternative strategy is to utilize one or more disk drives as buffer devices, stopping the input tapes periodically to allow the disks to be dumped to the archive. This scheme can probably utilize a part of the code which will support clock analysis, which will analyze files saved on disk in parallel with normal archive-writing.

Various media will have different projected lifetimes. The medium with the longest lifetime is not necessarily the best choice; the cost of the medium and the costs of periodic re-recording must be factored into the analysis. History has taught us that media have sometimes outlived the effective lifetime of the read-write hardware that they depend on, and this observation casts grave doubt on the relevance of all lifetime estimates of more than about 15 years. *This observation implies that re-recording logic should be implemented in any archive which proposes to maintain its bits longer than about a decade.* The correlator archive system will include a batch task which will periodically randomly select archive volumes for read-back verification and will log "soft" errors (note that the internal error correction of the medium plus the exclusive-OR blocks are expected to protect against "hard" errors, actual data loss, and that even if they fail the loss of some data will not cause loss of a whole observation). Statistical analysis of the soft read-back errors as a function of time will be used to set the time-constant for submission of jobs to copy volumes to new media. This copy mechanism can also be used to gradually migrate the data to new media as technology evolves over the decades. Storage space for archive media should be available for projected production over about 20 years (the increasing density of recording media and the technological lifetime of about 10 years will lead to steady-state size in about this range). The steady-state costs of copying volumes will probably raise the annual media budget of the correlator by 5-15% (for lifetimes ranging from 6-20 years). *This whole strategy, which is simply prudent policy for an eternal archive, is critically dependent on the assumption that the Observatory management will continue to fund the personnel, media and technology upgrade cost of the evolving archive forever.*

2.8 Distribution

The distribution writer will accept batch jobs to extract desired data from the archive volumes for distribution, either via LAN/WAN or on physical media (tapes, cassettes, etc.). Such jobs will be submitted in response to requests submitted by users over various networks interfaced to the correlator. Probably by the mid-90's at the AOC itself the requests will come from an AIPS task executing in one of the post-processing computers and the results will be passed to the task over the AOC LAN. Observers who lack network access or who want physical media will be able to request the tape output in their original observing requests and batch jobs will be generated automatically. Facilities will be available, similar to those at the VLA, to select subsets of a project's data at the cost of additional processing.

Visibilities from more than one array may be interspersed in an archive volume; each block will be a separate FITS "file", able to stand alone, apart from other blocks. The distribution writer will generally need to concatenate related blocks and suppress redundant header and extension-file information in order to obtain files of convenient size. The optimum distribution file size has not yet been determined, and may be allowed to be a user-specified option; sizes of 10 MB or more are plausible. Note that AIPS is able to accept and effectively concatenate multiple files of visibilities. The job generator will be able to generate a script which will specify which blocks (from tuples in the DBMS) should be concatenated into files, and it can also compute the total number of visibilities (from tuples in the DBMS) and supply it in the script so that the distribution task can generate a proper FITS header before concatenating the blocks.

The multi-source *uv*-FITS files produced by the distribution task will be in conformance with the FITS standards, will be compatible with AIPS task `UVLOD`, and will be similar in style to those produced by the AIPS task `FITTP`.

The job generator in CCC which will submit jobs to the RT system to produce distribution media from files must enforce NRAO's data-ownership policy until the files enter the public domain (18 months).

At the present time it is not possible to predict which *physical* distribution media will prove to be most popular as a function of time. Probably we will have to support 6250 bpi nine-track tape as a distribution medium, even though it is already technically obsolete and will be embarrassingly so by the mid-90's. Several types of magnetic and optical cartridge drives will be supported and the mix of devices will evolve over time with the technol-

ogy. *The distribution mechanism must be capable of supporting an evolving ensemble of multiple media devices.*

2.9 Screens

It is desirable that the operator interface of the correlator be very similar to that of the array operator interface in order to facilitate control of both systems by the same set of operators. This will be accomplished by utilizing the "screens" interface package which was developed for the array control system. The screens package assumes VT-100 terminals; one or more such terminals will be attached to the the RT complex for status display and operator input.

2.10 Bar Codes

Each reel of recording tape will have a bar-code sticker bearing a unique identifying serial number. Bar-code scanners will be located in the mail-room of the AOC, at each PBD of the correlator and at each recording tape drive at each VLBA station. When a reel of tape is mounted on any tape drive the associated bar code wand will be used to scan the bar code sticker on the reel. When a tape is shipped from or received at the AOC the code will be scanned again. It may even be useful to have wands at the "mailroom" of each VLBA station to scan tapes in and out of the stations. *The intent is that the VLBA DBMS will show the status (erased-at-AOC, erased-in-transit-to-station, recorded-at-station, recorded-in-transit-to-AOC, recorded-at-AOC, etc.) of every reel of tape owned by the VLBA, at all times.* DBMS queries on this status information are sure to be a regular operational activity. Queries on "recorded-at-AOC" are a condition for the submission of jobs to the correlator, and a tape cannot transition to "erased-at-AOC" status until processed results in the archive have been verified.

Checking bar codes at PBDs is redundant: the station/recorder codes and times recorded in the tape frame headers are sufficient to identify the reel of recording tape. But the bar codes are a valuable end-to-end consistency check.

2.11 Concurrency

It is likely that the correlator will evolve during its lifetime. Some new astronomical problem may be presented that will produce higher computational loadings in the real-time system. New media with higher data rates

are likely to appear, and be attached to the correlator. These thoughts imply the strategic principle that *not only should the correlator real-time architecture avoid CPU dependence* (so that we can use faster CPUs in future) *but it should also demonstrate concurrent RT CPUs* (so we can add more CPUs at will). Another strategic principle that is implied is that *the architecture should also demonstrate concurrent RT operations across more than one VME crate* (so that we can avoid bandwidth limitations due to contention on the VME backplane by adding more crates at will).

VxWorks (see Section 4.1.1) has already been ported to several different architectures. A notable example is the "SPARC" RISC (Reduced Instruction Set Computer) architecture; another example is the Motorola 88000 RISC CPU, a port recently said to be in progress. This continuing effort to port to the latest fast CPUs assures that we could gain speed by CPU swap in the future.

The ability of the VxWorks RT OS to support concurrent RT tasks is a key component of this strategy. It gives us the freedom to allocate RT tasks across multiple CPUs and even to invoke multiple instances of tasks in multiple CPUs to increase aggregate performance. The latter concept implies that the number-crunching power of a classical array processor can probably be effectively duplicated by adding more VME-module CPUs and invoking concurrent processes in them to use their aggregate FP coprocessor power. Such concurrent number-crunching could be used to implement the option for a time-domain filter at the output of the LTA.

The availability of VME bus repeaters means that CPUs in multiple VME crates can access each other's memories while minimizing impact on the independent, concurrent backplane bandwidths available in the separate crates.

2.12 Error Messages

> "...I *still* don't know what the paradigm should be for debugging a distributed and concurrent program... I mean, when you've got an application running across a number of machines, how should you think about it? When errors messages come from deep in the bowels of the system, asynchronously, how are you going to know what they mean?"[8]

The question asked in the quote above will be relevant in the correlator

[8] Bill Joy, in an interview in Unix Review, April 1988, p.67.

when we are running concurrent processes in several CPUs and crates. Error messages will stream to a common error log. With several CPUs and numerous processes executing at various priorities how will we untangle the sequencing of (correlated) error messages? We will establish a 1 kHz clock which will be synchronous across the RT complex. This can be done by establishing a high priority interrupt at that frequency in one CPU and a counter variable. The variable will be visible to other CPUs, and tasks in all CPUs will use the variable to time-tag error messages.

Numerous messages are likely to arise in the correlator with so many tasks interacting with so much hardware in such a sophisticated way. These messages will need to be displayed on a terminal attached to the real-time complex itself, and will also need to be passed to CCC to be appended to an error-message relation in the DBMS. The messages will have a format with various fields — time stamp (to 1 ms), originating "job" (if relevant), originating task and source-line-number, an error level code, and the message text (perhaps containing other parameters). The time stamp and error level will be used to sort messages in the real-time display (highest levels to the top, most recent cases at top of group of messages of same level, similar to the one at the operator console of the VLA). The stamp and level will also be used to delete older messages from the DBMS error log when some numerical combination of age and level exceeds a specified threshold.

It may make sense to combine multiple error messages from the same subroutine/source-line/job combination into messages like "19 messages of the preceding type were detected during last 4 minutes".

Periodic summary reports derived from the error logs will be prepared by a CCC batch process and E-mailed to appropriate operations personnel.

3 CCC — Correlator Control Computer

CCC[9], a Sun-3/280S with a 25 MHz Motorola 68020 CPU plus a 68881 FP coprocessor, is the server for the "CX.NRAO.EDU" Yellow Pages and Internet domains. Its MIPS rating is about 3.5 (about 3.5× a VAX-11/780). Currently it has 16 MB of RAM and one 892 MB SMD-style disk drive; probably it will have a second drive by the time it reaches the AOC. It has a high-resolution monitor (1600 × 1280), keyboard and mouse. It also has two terminal ports and a SCSI (Small Computer Systems Interface) QIC-24 cartridge tape drive. A unique peripheral is the IPC (Integrated Personal Computer), a PC-AT on a VME card plus a dual-floppy drive which supports DOS-in-a-window on any workstation in the CX complex (certain engineering and project management tools used heavily in the correlator project are DOS applications). CCC has two Ethernet controllers; it supports a private thin Ethernet for the correlator laboratory and gates it to NRAO's Edgemont Road building LAN. The real-time complex is attached to this private Ethernet. CCC is a *file server*—it acts as an NFS (Network File System) server to enable client workstations and real-time CPUs to mount its disk partitions. The IPC plus several PCs used by the correlator project utilize the PC-NFS client software to enable them to also mount and share access to CCC's disk partitions.

CCC will be the server for the DBMS, "Ingres" from Relational Technology. The DBMS server on CCC will support multiple client interfaces on the workstations of the CX complex, and will also be able to interconnect with other DBMS servers and their clients (i.e., it will be a "distributed" DBMS).

3.1 Batch Processes on CCC

CCC will utilize the Unix job-scheduling commands ("cron", "at") to cause the periodic execution of a large number of Unix processes which will account for the majority of the CPU time on CCC. Throughout the strategy discussions in Sections 2.x there are references to these batch processes.

Several processes will be concerned with the transfer of various pieces of data from the the array control computer (ACC[10]) to CCC for entry into the DBMS. A list of various types of records to be transferred is given in Table 1. Although the table applies specifically only to VLBA observations, similar

[9] named ccc.cx.nrao.edu.

[10] a Sun-3/260,-named vlbacc.aoc.nrao.edu.

- For the VLBA as a whole:
  1. List of all scheduled stations whose data will be correlated.
  2. List of valid project codes and attributes.
- For each station to be correlated:
  1. Log of observing events, including *null* observations and all observational parameters.
  2. Log of tape events (bar-codes).
  3. Amplitude- and phase-calibration measurements.
  4. Bad data flags.
  5. Weather records.
  6. Clock offsets from GPS & fringe checks *(required daily)*.

Table 1: Items to be Provided by the Array Control System

information will be required for joint observations by "global" arrays, and for any external arrays which may be correlated. These data transfer processes may have to involve some format conversions. It would be better for the data to arrive as SQL `INSERT` statements than as a flat text table; even better, at least within the VLBA, would be to fetch it using the distributed DBMS mechanisms from relations in a server in ACC.

A major duty for CCC is the preparation of job scripts for the real time system. The important log-entry and job-generation processes are described in some detail in Sections 3.1.1 and 3.1.2 below.

Another major duty is the analysis of clock offset data and prediction of future offsets; this was discussed earlier in Section 2.6. Batch processes must enter real-time fringe check results from the array control system, must execute a predictive filter batch process, and must enter clock analysis results arriving from the RT complex.

Section 2.12 on error-message strategy described how a combination of the age and priority of error messages in the DBMS on CCC will be used to delete older, lower priority messages. It also suggested a batch process to collapse voluminous error logs to a smaller size by combining redundant messages.

Another earlier section (2.3) described how the archive task will generate a logging message which will record the time of each output block that is written. A batch process on CCC will check periodically check for jobs which aborted or which failed some portion of their verification job; these jobs must be resubmitted. Finally, yet another batch process will detect when all observations from a given input tape have been processed, and will then release that tape to be erased and shipped to another station.

Batch jobs will generate E-mail messages to users to inform them of completion of the processing of their data, and of its location in the archive.

3.1.1 Log Entry Task

The station control systems will record the beginning and end of each observation, and these log records will arrive at the correlator through the VLBA communications system, passing through ACC on their way to CCC (see Table 1). The Log Entry task will insert the records into the appropriate relations in the DBMS. At this point each station which is participating in a given observation will be represented by its own record in the observation relation. Various data-range and consistency checks will be performed on the input data.

The algorithms which will automatically recognize and schedule the processing of observations depend on three assumptions being satisfied:

1. *a complete chronology of station schedules will be available,*
2. *the observation records from each station will give a complete chronology of the state of that station for all times, including periods when it is slewing or idle or is down for some reason, and*
3. *a complete list of valid project codes will be available.*

The first item enables us to know whether our data are complete, i.e., whether we have observation chronology data from all stations which need to be correlated during some range of time. The third item allows a consistency check of the project codes which will be included in the observation chronology. Use of stop as well as start times in the chronology will enable us to know that the chronology is complete, and is also a consistency check. These consistency checks are a desirable redundancy.

Whenever a station is not recording supposedly valid signal on a program source its control computer should record a log record indicating that it observed the "null source" from the ending time of the last valid program source observation until the beginning time of the next source observation. When the computer crashes it must generate such a null-source observation record when it comes up again (which implies that it must note what it is doing plus the current time in a non-volatile place at regular intervals, say once a minute), in order to assure the continuous coverage of the chronology. Multiple contiguous null-source observation records are acceptable, of course.

The preceding discussion applies in particular to VLBA stations, but similar arrangements will have to be negotiated with non-VLBA stations involved in joint observations.

Observation logging records will arrive in arbitrary order from the various stations, and it is always possible that they may be lost or temporarily delayed in transmission, or be duplicated. The Log Entry task will look for "holes" in the time coverage of the logs (due to missing or corrupted records). A separate test will detect duplicate logging records (which may easily arise in the transfers between various computers), and will suppress the redundant copies. Either case is an indication of failures of logging and/or communications algorithms and will inhibit job submission until it is corrected.

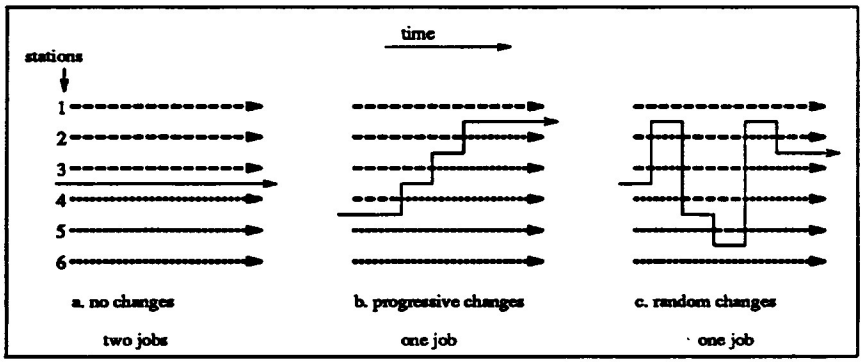


Figure 1: Jobs and the Time Variation of Array Membership

The Log Entry task will also enter other logging information arriving from the VLBA array control system. In particular, the tape chronology will be entered, with various data integrity and consistency checks being applied.

3.1.2 Job Generator Task

If we draw station timelines and construct a polygonal outline around an array, the outline will have "jagged" ends as stations progressively join (or leave) the array due to local horizons, different slew times, and, possibly, arbitrary user scheduling (see Figure 1). Arbitrary patterns of change of stations from project to project and array to array are allowed. Case "c" in Figure 1 shows the worst case: stations leaving one array for another at random, and then returning later.

Assume that we have determined that a contiguous observation chronology is available for all stations that observed for a certain range of time. A list of project codes will be SELECTed (with SQL) from the chronology. For each project a list of unique combinations of source and observing parameters will be selected.[11] For each combination an array will be declared,

[11] A significant technical difficulty is the "Mark III Hybrid Mode", in which the VLBA plus other stations will operate as an inhomogeneous array, with not all bandpasses in common. This will complicate the recognition of related observations; precise procedures have not yet been worked out. The authors speculate that the problem of recognizing common bandpasses will prove to be analogous to the problem of automatically recognizing arrays, and that a similar algorithm may be feasible. Two other technical difficulties are Doppler tracking, with different frequencies at different stations of the same array,

and a unique code number assigned, in an array relation in the DBMS, and in the observation chronology tuples. For each array a list of stations that observed will be selected, and will be tabulated in a relation. The time duration of each array will be from the earliest time to the latest time in the observation chronology for all stations determined to be in the array.

The Job Generator task selects arrays for which jobs have not been submitted and for which all of the tapes needed for the time ranges are available. A *job* consists of one or more *arrays*, belonging to one or more observing *projects*, observing for a range of time. A sequence of SQL SELECT commands will extract the set of tables which describe such a job.

For each array, a list of other arrays which include common stations in the time range will be selected. If this list is empty (i.e., case "a" in Figure 1), a unique job code number will be assigned; otherwise, the job code number of the first array of the list will be assigned. At the end of the process a count of the job code numbers will indicate whether arrays fall into two disjoint sets (analogous to case "a"); such cases can be submitted as two distinct jobs in order to increase the flexibility of scheduling the processing in the correlator.

If the duration of a job is greater than some specified limit (perhaps six hours), it must be split into two jobs. Arrays in the second half of the time range will be given a new unique job code.

The automatic array recognition algorithms discussed in previous paragraphs produce tables (relations) of jobs and arrays, and of stations which are in those arrays. The station chronologies are also tables. A job *is* these tables. Each job will be converted to the script syntax which the Job Loader Task in the real-time system expects. The conversion can be done by an application-independent utility which can work from nothing more than the names of the relations to be translated, using the attribute names and formats specified in the DBMS schema. If a relation contains columns which should not be written to the script a "view" (virtual relation) will be defined which does not contain these columns.

The script files will be written into an agreed directory on CCC with some agreed name rule, and the RT Job Loader Task will access them via an NFS mount of the CCC directory. The job queue in memory will be generated from these disk files. As long as a job has not been initiated in the RT system its disk file can remain in the CCC directory so that it can be

---

and pointing offsets at different stations; presumably the array recognition algorithm can invoke the source name plus some tolerances to resolve these ambiguities.

reloaded after a crash. But once a job has begun to execute its associated disk file must be removed. A logging message will be sent to mark the job in the DBMS as being in progress. This message will be time-tagged in the DBMS; a job file will be re-generated by CCC if the time-tag is not overridden by a job-completion within about 24 hours.

Note that the correlator must be able to process observations whose arrays contain more stations than the number of correlator station inputs, not only during the construction phase when the number of station inputs will be increasing, but also to permit processing of arrays combining the VLBA with numerous other stations. Thus the Job Generator must automatically generate multiple jobs in order to make multiple passes on the input tapes so as to get visibilities for all — or only the most valuable — baselines. Multiple jobs will also be necessary for cases where more channels or more phase centers are needed than the correlator supports.

3.2 Interactive Processes on CCC

Although most CPU time on CCC will be consumed by batch processes executing embedded SQL statements on the DBMS server, there will be a variety of interactive facilities available to operators and users.

Operators will have windows in which warning messages will be posted ("data tape in transit more than 6 days", "observation inconsistent with array assignments", "archive volume read-back failure", "low blank tape stockpile at Brewster", etc.). They will also need to execute various queries to check status of jobs, tapes, etc., as reflected in the DBMS contents. The latter will generally be "canned" queries; it is probable that they will be invoked by menu items in a "screen". To speed complex queries batch jobs can periodically perform such queries and cache the results. Common operations which modify the database can also be coded into applications using the screen interface. Alternatively, it may be better to use the forms interface tools of the DBMS to produce such applications. Because the correlator is being designed to be totally automated very little thought has gone into this whole class of interactive operations so far; they will be devised as needed during the course of the project (obviously they will be most needed during the development phase of the project to generate test cases and work around bugs).

In principle the users should have no need to communicate with the correlator because the correlator will produce data for them on the basis of their original observing requests, which will include processing parameters.

However, network access will be feasible, and it is plausible to suppose that observers and array management might be able to perform some of the following functions:

- Enter proposals using a proposal-entry application (perhaps similar to the "MIPS" facility which supports ROSAT?).
- Enter proposal reviews.
- Enter detailed observation information.
- Enter station scheduling.
- Enter valid project codes.
- Modify processing parameters before the Job Generator process runs.
- Query the status of processing of jobs.
- Request production of distribution media.
- Search archive catalogs for observations of interest.

Because the DBMS will contain a *complete* description of all VLBA observations, it will be effectively equivalent to the array scheduling assignment plus the detailed observing commands which are given to the array control system by the observer. In which direction should the information flow — from the DBMS to the array, or from the array to the DBMS? And what tools will be provided to aid observers in generating the detailed observing commands? Should such tools interface to the DBMS? These questions cannot be answered satisfactorily until the project has more experience with the DBMS and the schema design is at least partially available.

4 The Real-Time Complex

4.1 The Hardware/Software Configuration

The real-time complex consists of two "6U"-height VME crates (Figure 2). One crate contains two Motorola MVME147 CPUs. Each of these is two VME slots wide and contains a 25 MHz 68030 CPU and a 68882 FP co-processor, 8 MB of RAM, SCSI channel controller, Ether controller, two serial ports and miscellaneous other devices. These CPUs have about 5-5.5 MIPS of speed. Each CPU will have a SCSI disk drive, probably with about 300 MB each; the on-board SCSI controller does not impact VME backplane bandwidth.

A special CPU will be present in the crate with the two 147 CPUs and will be the channel controller for communication with the correlator racks. This device is discussed in Appendix I (Hardware Control Bus Interface) and appears in Figure 2.

Two DMA channel controllers will also be present in the crate, and will interface to the LTA (Section 4.3.10) and to the archive devices (see Figure 2). They will read into and write from the memory of the second 147 CPU. That CPU's own SCSI controller can transfer portions of its RAM to and from its SCSI disk drive simultaneously. Multiple autonomous I/O channel controllers operating concurrently are needed in order to support multiple data flows each up to 0.5 MB/sec between the various devices, while the CPUs are performing various computations.

An MVME331 serial-line controller plus MVME705A transition module will support an MCB (Monitor & Control Bus) connection to the 24 PBDs; this will be used by the 24 tape tasks. Each PBI will have an RS-232 serial connection to its associated PBD which will be used for the tape-speed servo loop. Presumably commands arriving simultaneously on the two inputs of the tape controllers will be processed sequentially.

The second VME crate will contain a Motorola MVME130 CPU, which uses a 68020 chip with about 2 MIPS integer speed and a 68881 FP coprocessor. Several dual SCSI channel controllers will enable numerous archive and distribution devices to operate simultaneously. Because the archive is in nearly the distribution format, and has large block sizes, the MVME130 CPU will not have to do much actual work and may be lightly loaded even with several concurrent DMA operations in progress.

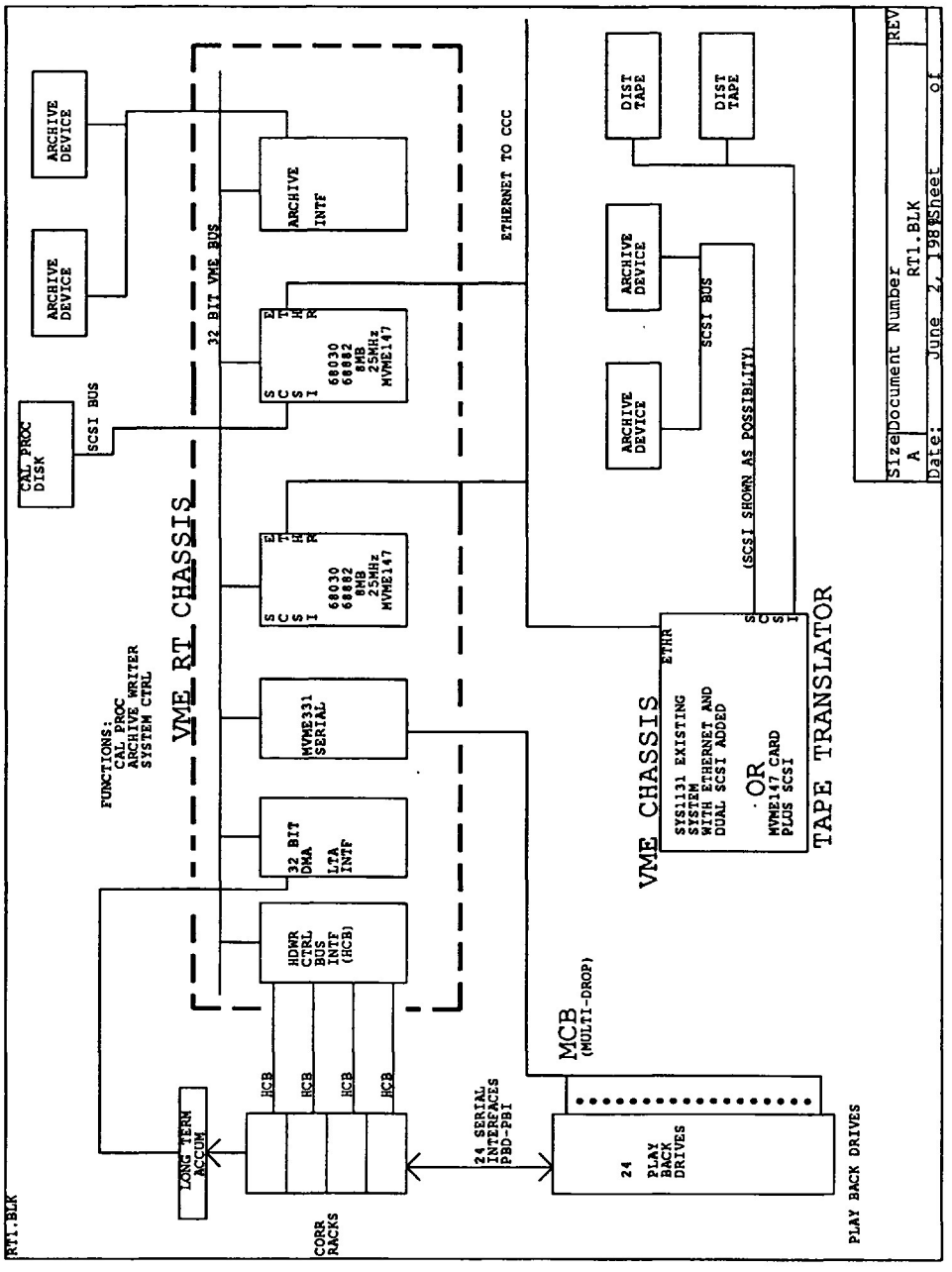


Figure 2: Real-Time Hardware Configuration

4.1.1 About VxWorks

The correlator will use the VxWorks[12] real-time "kernel" as its RT operating system. VxWorks is not intended to be a complete OS; it is not intended to support timesharing (only one login at a time) and is not intended to execute a compiler. It is notable for containing a rich networking environment, permitting remote logins, file transfer, Berkeley socket calls, remote procedure calls and NFS-client mounts. It supports both Ethernet and VME backplane as network media. The workstations of the CX complex complement VxWorks and the real-time CPUs to form a complete RT development environment.

VxWorks claims to support up to about 250 real-time processes. *The proposed correlator software architecture depends critically on the assumption that it will be practical to dynamically (at will) create and destroy up to about 100 real-time tasks.* This assumption simplifies much of the logic by allowing, for example, each of 24 PBDs and each of 20 stations to have its own task, each job in the queue (even jobs which are waiting for a previous job to complete) to have its own task, plus tasks for delay-model calculations and observation control.[13] Because we will have tasks spread across several CPUs, *the architecture assumes that invocation of remote processes really does work in VxWorks.* The two assumptions mentioned here have not yet been verified experimentally by us (although we have no reason to doubt the claims).

VxWorks does not support memory mapping. All real-time code and all variables declared outside C-functions are shared across all tasks (i.e., there is only one name space). This is both a blessing and a curse: one gets the effect of shared libraries and shared COMMON blocks automatically, but private global variables in a package of functions are risky, especially if the functions can be invoked more than once. The latter is a real possibility — any function can be spawned as a task (with arguments passed) and a function can be invoked more than once (usually with different arguments, such as the PBD and station tasks). The answer to this problem is to define a structure containing the shared variables, define a single shared pointer variable, allocate memory for an instance of the structure for each instance of the task and set the pointer appropriately for each instance. The multiple pointer values are kept distinct by declaring the pointer to a "task-variable" for the tasks so that it will become a part of the context of the tasks.

[12] VxWorks^{TM} Software was developed by Wind River Systems, Inc., Emeryville, CA.
[13] at least $24 + 20 + (5 \times 4) = 64$ processes are expected.

Almost all RT code will be coded in C, initially in the K&R dialect and eventually in ANSI C. Some imported modules (notably CALC, see Section 4.3.9) will be in Fortran.

4.1.2 Performance Guess-timates

The CPUs and I/O channels of the correlator RT system are much faster than the RT systems used previously at NRAO; they are also more complicated. It is difficult to extrapolate their performance from prior experience. We hope to make a variety of measurements in the months to come, but must make do with estimates for now, estimates which may be uncertain by a factor of two.

There will be two CPUs, executing at about 5 million instructions per second. They can probably make a VxWorks context switch from one task to another in about 50 μs. Typical basic I/O operations cost perhaps 200 instructions (40 μs) in a RT OS like VxWorks; therefore we expect that each CPU can respond to an interrupt frequency up to about 10 kHz, doing some useful work.

Latency on a remote procedure call from one CPU to the other over the backplane will probably be less than 500 μs; it would be nice if it turns out to be as small as 200 μs.

Backplane bandwidths are difficult to predict. The theoretical limit of a VME bus is 40 MB/sec, but it is rare to sustain more than about 10 MB/sec in practice. This is probably enough for plausible enhancements of the correlator, because the current estimate for the ultimate limit of LTA output rate is about 2.8 MB/sec and we only need four times that rate even if we buffer to and from a disk system on the way to the archive devices.

4.2 Real-Time Tables and Structures

The real-time software reads control data through shared data tables that are globally accessible to all tasks in the system. The data tables are arranged in a hierarchical order, and are connected by pointer variables in each table that point to the next group of tables further down the hierarchical tree. A master table is locatable via a global pointer variable that contains the address of the first element of that table. By following a linked list of pointers, any table in the system can be accessed quickly and simply. The table system and the real-time tasks are cross referenced in Table 2.

The real-time data tables for a single correlator job are constructed and

Tasks	Real time tables																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
array	r	r	r	r					r	r		r						
clock	r	r	rw															
cross	r	r	r															
init_active	w		c															
init_model	r	r	w						r	r	r	r						cw
init_queue	c																	
job	rw	r	r						r									
job_control	r	r							r									
job_loader	w	cw		cw	cw	cw	cw	cw	cw	cw	cw	cw	cw	cw				
LTA	r	r	r	r	r				r	r		r						r
model	r	r	r	r			r	r	r	r	r	r		r	r	r	r	w
sched	rw	r			r				r									
station	r	r	r		r													r
tape	r	r				r				r			r					
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Keys:

c	create
r	read
w	write

	Tables:
1	queue_table
2	job_table
3	active_base_table
4	channels
5	correl
6	formatter
7	clocks
8	constants
9	job_header

	Tables:
10	observations
11	sources
12	stations
13	tapes
14	UT1
15	polar
16	solar
17	constants
18	model

Note: There are some minor inconsistencies between the names of tasks and tables given here and as used in the text; the names in this table represent work-in-progress, *actual code as built.*

Table 2: Real Time Tasks *vs.* Real Time Tables

loaded by the Job Loader Task, which reads a correlator script file on a VME disk, and directs the contents into the appropriate data tables. The Job Loader Task essentially copies the script file into the real-time tables with no modifications. It also loads addresses into pointer variables that link the data tables for that particular correlator job.

4.2.1 The Queue Table and the Job Base Tables

The table at the highest level in the hierarchy is the Queue Table. A pointer variable, "queue_base_ptr", contains the address of the first element in the Queue Table, and it is the only globally common variable in the tables system (in the VxWorks symbols table). The Queue Table contains a row for each correlator job that is active (being correlated) or waiting its turn in the job queue. The correlator operators will be able to observe and manipulate the job order in the queue (table) through the screens package. The contents of the Queue Table are defined in the include file QUEUE_TABLE.H (see Appendix C). Three of the elements in each row are particularly important: "job_index", "job_ptr" and "active_ptr". The "job_index" is an integer number that uniquely identifies each job in the Queue Table. The job index number provides a link from any task in the system to that job's entry in the Queue Table.

The Queue Table pointers, "job_ptr" and "active_ptr", provide links to the two base tables that in turn connect to all of the various data and control tables associated with a particular job. The two base tables are the Job Table and the Active Base Table (see Figure 3). The Job Table points to the data tables which contain essentially literal copies of the script file tables. There is one row in the Job Table for each data table. The Job Table structure is defined in TABLES.H (see Appendix D) and the Active Base Table structure is defined in ACTIVE.H (see Appendix G).

4.2.2 The Description Tables

The Job Table row that corresponds to each data table also contains a pointer to a description table. The description tables exactly describe the individual data table formats. There is a separate description table for each type of data table. The description tables are declared in TABLES_DESC.H (Appendix E). A single set of description tables are used by all jobs. The description tables contain one row for each script file keyword: the keyword string, its data type and position offset in its data table row. The description

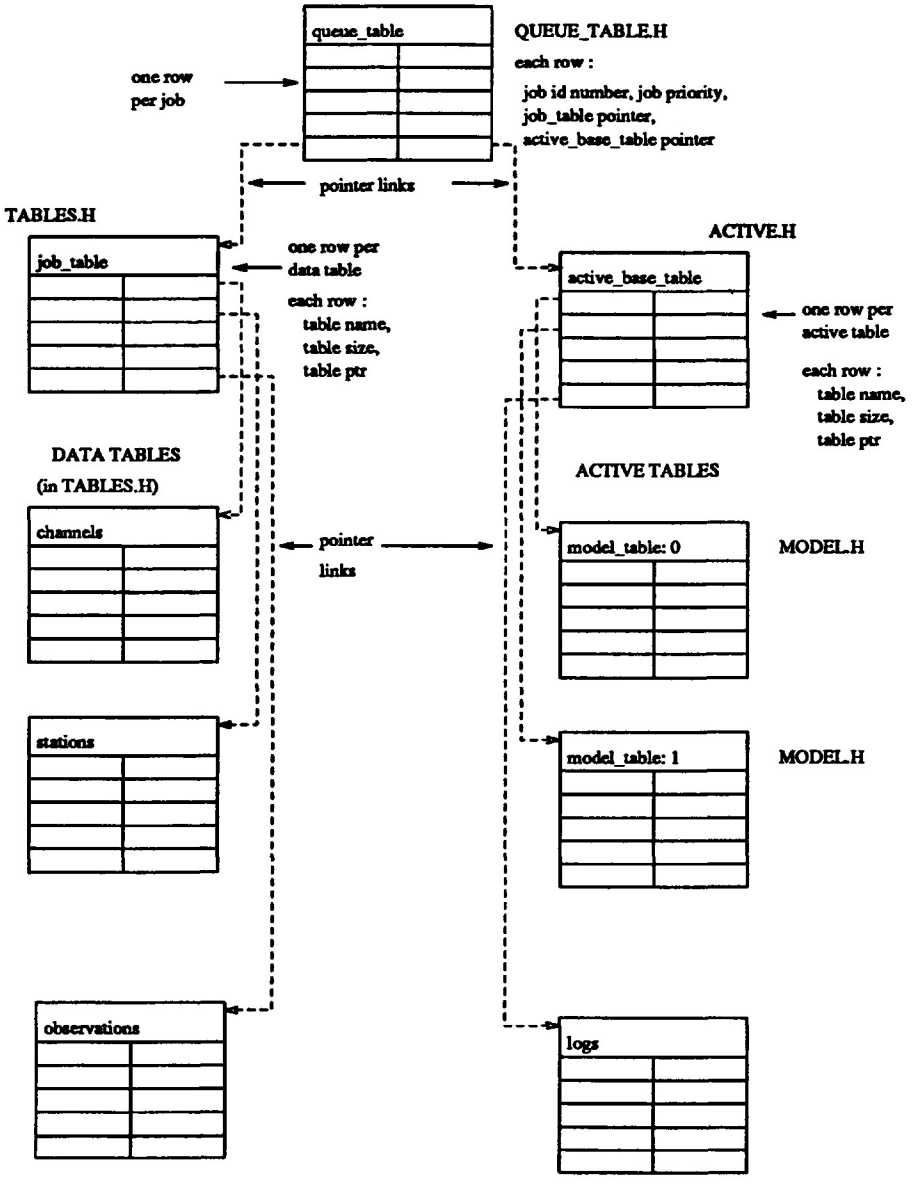


Figure 3: Table Hierarchy in the Real-Time Environment

tables allow easy and safe modification of existing data table formats, and the creation of new ones. The Job Loader Task does not refer to data table names or keywords by name. Rather, script file keywords are matched against strings in the description tables and values are stored via pointer directions.

4.2.3 The Data Tables

There are currently over a dozen data tables associated with each correlator job. They contain values loaded directly from a single script file. The data tables are described in the include file `TABLES.H` in Appendix D along with a sample script file (Appendix B). Each table is allowed a variable number of rows depending on the number of rows in the corresponding script file table. A row count is kept for each table in the Job Table. Two tables have only one row: the Physical Constants Table and the Job Header Table.

There is a one to one mapping of the script file parameters into the data table elements. In fact, the structure member names are the same as the script file keywords. The row elements are zeroed or blanked when the tables are created. While the script file contents are being transferred into the tables, unspecified elements are given the value of the same element in the previous row (for the first row the "previous row" is filled with nulls).

4.2.4 The Active Tables

The active tables are in a separate hierarchy because they are fundamentally different from the data tables. They may be created and destroyed at any time during the lifetime of their controlling job, their size is not fixed (nor necessarily known in advance), and new rows may be added at arbitrary times in the job processing. Some active tables will contain row pointers that link the rows in the individual tables. For example, this will be necessary in the Logs Table where its ultimate extent cannot be known at the beginning of the job processing cycle.

4.3 Real-Time Tasks

When the correlator real-time system is booted it will initiate the main Job Control Task which will run forever. The Job Control Task will then initiate an instance of the Tape Task (Section 4.3.3) for each PBD, the Crossbar Task (Section 4.3.4) which manages the crossbar switch, and an instance of the Station Task (Section 4.3.5) for each station input of the correlator

(eight FFT engines per input). The Job Control Task then initializes the basic structure of pointers for the job queue and spawns the Scheduler Task whose duty is to load and arrange the queue into a schedule which the Job Control Task will follow. The Job Loader Task reads scripts into the job queue structures (see Section 4.2). The Scheduler Task then manipulates these structures using the strategies which were discussed in Section 2.4.

The Job Control Task initiates the next job in the queue at the time specified in the queue (the time is chosen by the Scheduler Task). Jobs are initiated in advance of the time they will execute in order that they may prepare delay model calculations (Sections 4.3.8 and 4.3.9) and acquire their data tapes.

The Job Control Task initiates a job by spawning an instance of the Job Task, which then initiates instances of other tasks to perform the work of the job. The relationships of the various tasks to each other are shown schematically in Figure 4. The Job Task spawns the Model Task (Section 4.3.8) which, in turn, spawns the CALC Task (Section 4.3.9). The Job Task also spawns an instance of the Array Task (Section 4.3.7) for each array in the job. The Array Task interacts with the Tape, Crossbar and Station Tasks to acquire the data tapes for the job and connect them to the station inputs of the correlator. The Job Task also spawns an instance of the Archive Task (Section 4.3.10) for each project in the job. The Array Tasks interact with the Archive Task for their project in order to control the averaging of visibilities in the LTA, to maintain the index vectors for the LTA and to write the project's visibilities to the archive.

More than one job may be in execution at one time, and each such job will have its own instances of the Array, Model and Archive Tasks. Instances of the Array and Archive Tasks contend with their other instances for access to devices and resources. Such contention involves "critical regions" of control which are interlocked with semaphores.

When the previous job ends and the necessary tapes are mounted and positioned, each Array Task will request a crossbar reconfiguration for its tapes and when all of these are completed the Station Tasks can order a register bank switch in the correlator. The Array Task will also alert the Archive Task associated with it so that the index vectors and integration times can be set appropriately, and the archive output buffers initialized. When all of these operations have completed successfully the correlator can resume processing signals.

The general strategy of the tasks will be that each will strive toward one or more goals, each will present various status signals for the other tasks to

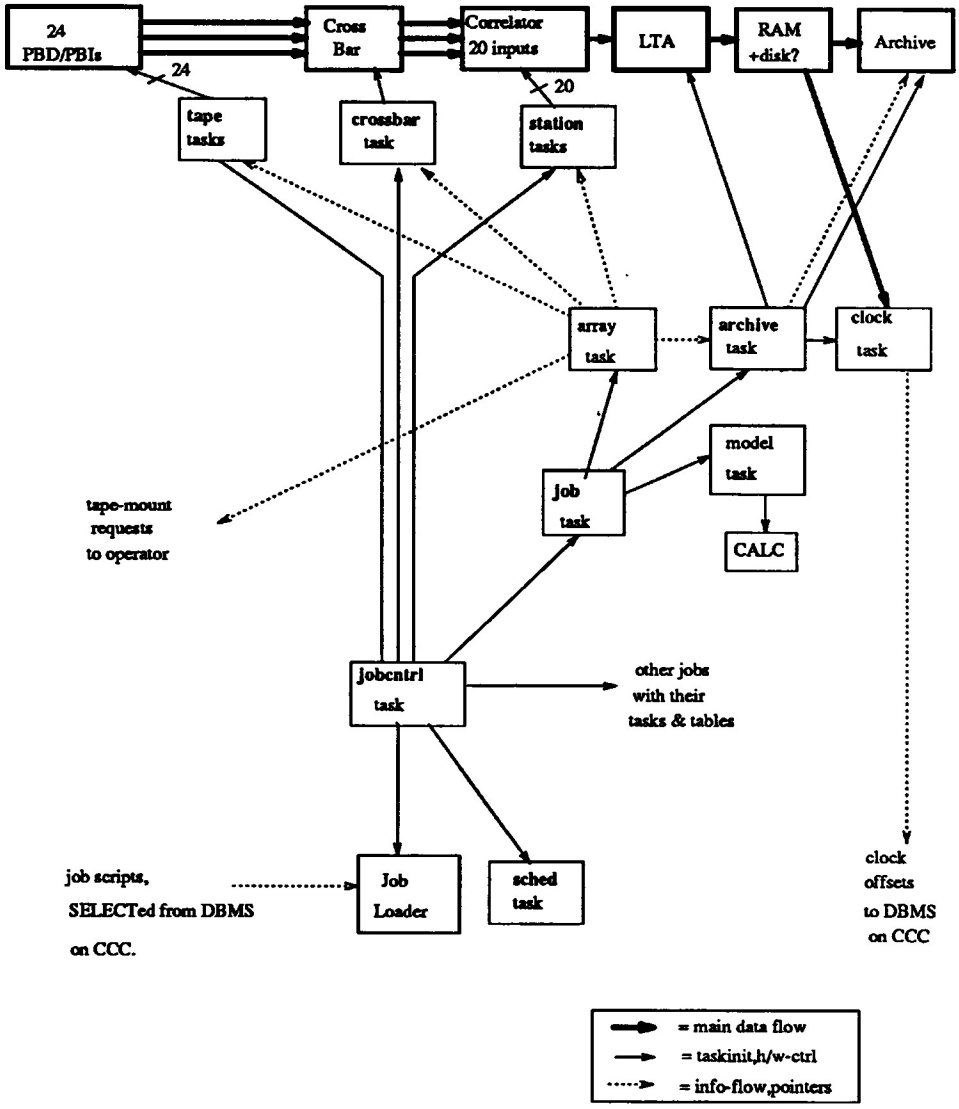


Figure 4: Principal Real-Time Tasks

monitor, and each will monitor those signals and progress toward the goals as combinations of signals permit. The various tasks will execute at various priority levels. They will awaken periodically to monitor their signals, and generally they will also awaken each other whenever signals are changed.

A variety of registers in the correlator must be loaded by the Station Tasks to set the processing parameters. Some registers and RAMs need only be loaded at startup time and it may be appropriate for an initialization task to do this job.

As processing proceeds new Array Tasks are spawned by the Job Tasks and old Array Tasks terminate. When a job (a set of Array Tasks) completes, its Job Task will terminate after some cleanup work. But, in general, the Array Tasks associated with some other Job Task will then acquire the correlator resources that are freed by the termination.

Thus, a variety of tasks will be created and destroyed continuously during job processing. Only the Job Control, Scheduler, Crossbar, Station and Tape Tasks will live forever.

Operators will interact with the Job Control Task to adjust the priorities of jobs in the queue, terminate jobs which are in execution, query status of jobs, etc. The Array Tasks (Section 4.3.7) will post messages to the operators about tapes that they would like to have mounted.

Probably the Tape, Crossbar and Station tasks will operate in the first MVME147 CPU in the first VME crate, the Archive and Clock Tasks will operate in the second MVME147 CPU in the first crate, and the distribution and verification tasks will execute in the MVME130 CPU in the second crate. See Section 4.1 for hardware details.

4.3.1 Job Control Task

The Job Control Task spawns the Job Task to process a job, which is a linked list of C-structures which the Job Loader Task has produced, in turn, from a set of table descriptions produced by the Job Generator in CCC from SQL SELECTs on the DBMS. The job queue contains a list of pointers to the jobs (linked lists). For each job the queue structure also carries the time at which the job should be initiated. The queue structure will always be sorted in time order. The Job Control Task only needs to watch the clock and start the first job in the queue at the proper time.

Special categories of jobs may be defined, and there may be commands available to the operator to control whether these categories can initiate. The principal example of this will be clock jobs (see section 4.3.11). Such

jobs may require operator intervention and the operational staff may want to run them only when certain operators are on duty, or only at certain hours of the day.

4.3.2 Scheduler Task

The Scheduler Task decides what starting time to assign for the jobs in the queue, and it maintains the queue in time order. Access to the queue structure by the Job Loader, Job Control and Scheduler Tasks will be interlocked by a semaphore. The Scheduler Task will execute at lower priority and, in general, it will be making most of its decisions far in advance (tens of minutes to hours) of the job starting times that it specifies.

If the correlator has enough resources (station inputs) to process multiple arrays it will be the duty of the Scheduler Task to find work in the queue to utilize the resources as fully as possible (see Section 2.4). That is, it should command the initiation of two or more jobs. The constraints and optimization criteria for this problem are complex, and poorly understood at this time, if a *perfect* solution is desired. It does appear that searching for optimum solutions will always involve examination of only a modest number of sequences of jobs arranged in order of observe time, *not* the $n!$ possible permutations of the jobs, and that therefore a simple approximation to the optimum solution will achieve a reasonably high efficiency.

Probably the first versions of the Scheduler Task will implement simple brute force approaches to the scheduling problem, and the algorithms can then gradually become more sophisticated. Because of the interest of this problem we will sketch some preliminary ideas for several of the needed algorithms here.

The Scheduler Task can first sort the queue of jobs into order of observation. Generally the oldest job should be scheduled for immediate initiation. If the next job in the queue follows it immediately in original observing time, with many tapes in common, it should be scheduled for initiation somewhat before the predicted termination of the first job. "Somewhat before" means long enough to carry out necessary initializations.

The hard problem is to decide what other job(s) should be initiated in parallel with the first set of jobs that are handling the oldest unprocessed observations. In general, we want to schedule parallel execution of a job which has no tapes in common with currently executing jobs and their immediate successors. This can occur either because of two unrelated time sequences in the job queue or because a sequence extends over a time range

longer than about 1.5 tape durations.

A particularly important concept is the "width" of each job — the maximum number of correlator station inputs which the job will need. A job can only be initiated (scheduled) if the width is less than the number of uncommitted station inputs. For example, the Scheduler Task might choose the "widest" job which can be initiated in parallel and which will not conflict with the tape needs of the previously scheduled jobs for at least 1.5 tape durations, and whose execution will not extend beyond the predicted termination of the current jobs. The purpose of the latter rule is to assure that the oldest job can always be initiated (assuming, of course, that its width does not exceed the maximum width of the correlator).

A variety of constraint rules apply in this scheduling process. It is most likely that the Scheduler Task can simply explore the possibilities by exhaustive search, because it has plenty of time and CPU speed at its disposal (at low priority) and because the number of jobs to test is modest enough. If search speed proves to be a problem the solution will probably be a rule-based engine approach, probably using neural-net techniques. An interesting technical question is whether the problem should be solved ab initio whenever a new job appears in the queue, or can the schedule be *revised* at lower cost to introduce the new job.

4.3.3 Tape Tasks

The Tape Tasks will be initiated by the Job Control Task at boot time, with one instance per PBD-PBI pair; there are 24 such pairs. Each Tape Task acts as an agent (a server) for an Array Task when the PBD is acquired by that array.

Each Tape Task watches its PBD and PBI to detect any anomalies of operation. It also listens to the associated barcode reader. When a barcode input occurs the assumption is that a reel of tape is properly threaded and ready to be loaded. The task will load the tape and attempt to read some signal from the tape in order to determine the station, recorder and original time of the recording. The barcode input will indicate the VSN of the reel.

At this point the Tape Task will examine the job structures to determine if any job is requesting its reel of tape. If a match is found, the task will attach itself to that job. *Operators may mount requested tapes in any order and on any drive and the Tape Tasks will automatically assign the tapes to the jobs, without any manual intervention.*

The Array Task of the job will be requesting data for a certain time.

Each Tape Task will slew its tape to the proper time and will prepare to deliver data to the correlator.

The Tape Task will help with servoing the tape to track the delay model (a local servo loop will tightly couple the PBI to the PBD). It will continually monitor the status of the PBD and will monitor and log tape errors.

When a tape is no longer needed by an Array Task, its Tape Task will look for a new array to which it may attach. It will often be true that another array is waiting — the next consecutive observation in the job — and that that array will want the tape to continue from its current location. It may even be possible that the tape can be kept in motion during this transition from one array to another.

Eventually the Tape Task will find no array for its reel of tape. The task will rewind its reel, but will not unload it until absolutely necessary (or upon operator command, of course). The intent of the strategy is to minimize dismount/mount actions by the operator. Whenever some array wants a tape that is not currently mounted on some PBD, there must be at least one PBD available if at all possible. The Tape Tasks will all monitor the request list and will note when an idle drive needs to be freed. One of them will rewind and unload its reel and resume waiting for another barcode input to signify the mounting of a new reel on its PBD.

The Tape Tasks will share the common MCB interface to the PBDs. Probably colored lights will be mounted on the PBDs to signify their status (assigned, mounted, free) as an aid for the operators.

4.3.4 Crossbar Task

The crossbar has a set of connections to the station (FFT) inputs which form an elegant pattern (Figure 5). The purpose of this pattern is to create a variety of opportunities for PBDs to be available for mounting new reels of tape. When an Array Task detects a new combination of PBDs in its array it will alert the Crossbar Task. The Crossbar Task will display a "please-pause" status, and soon all Array Tasks will pause (at the end of a current integration), at which point the Crossbar Task will work its way down the current PBDs and will assign station inputs to them. It will note the assignments in their array tables. Such a reconfiguration will be required at most tape changes.

Note that the crossbar station input assignments control where in the LTA the data for the various baselines will appear. Tapes will be mounted on PBDs in random order, with tapes from various arrays intermingled.

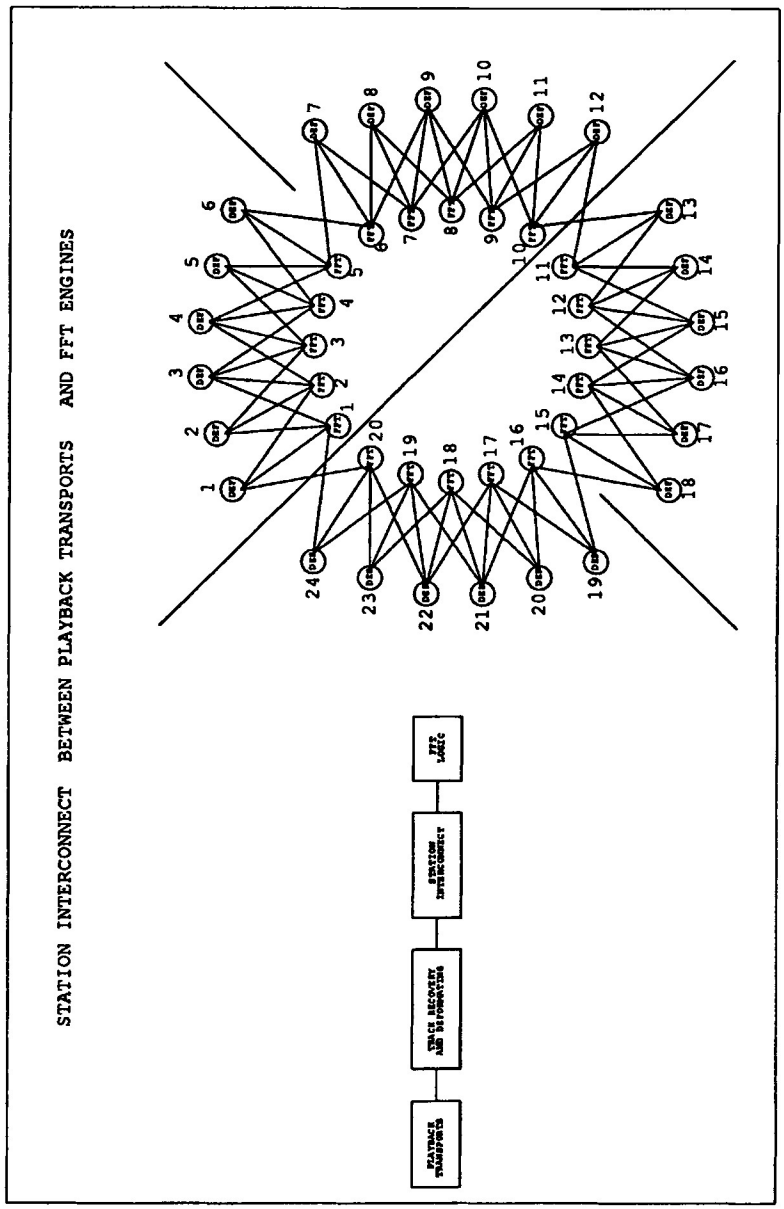


Figure 5: Crossbar Connections from PBDs to FFT Station Inputs

The Archive Task (Section 4.3.10) will be able to determine which LTA cells contain the data for its baselines.

4.3.5 Station Tasks

The Station Task controls all of the hardware associated with one station input of the correlator. There are 20 such station inputs, each eight channels deep, connectable in various configurations to the eight hardware FFT engines with associated delay and phase model generators. Station Tasks are initiated by the Job Control Task at boot time.

A station input is connected to a PBI-PBD pair by the crossbar. The PBD has a tape with data from some station. The station is a member of some array. The array observed for some project. By following the chain of pointers, which will generally change at every tape change (and thus require a crossbar reconfiguration), a Station Task can find the observing and processing parameters for its current station, array, project and job.

The Station Tasks begin preparing for the next source as soon as the current source is in progress; this enables new register settings to be prepared in advance of need. Access to the registers of the correlator will be interlocked with semaphores. Such registers will be organized in dual banks, thus permitting preloading. It may be expedient for the Station Task to "remember" previous settings of its station input so that it can transmit only changes to the registers.

4.3.6 Job Task

Instances of the Job Task are spawned by the Job Control Task. The Job Task begins execution in order to perform various initialization operations before it and its minions can acquire tapes and process signals. The Job Task spawns the Array and Archive Tasks.

4.3.7 Array Tasks

The duties of the Array Task are to procure signals from the stations of its array at the specified time and to coordinate with the Archive Task for its project to get the visibilities of its array written to the archive. It determines which reels of tape it needs in order to obtain the desired signals. These reel serial numbers constitute an "advertisement", and a Tape Task will eventually recognize that its reel of tape will satisfy the request. The Tape Task will be able to supply the station tape recorder code which it has read

from the tape, and the Array Task can verify this against the tape logging information which the array control system has supplied (Table 1).

The Array Task checks to determine whether all of its Station tasks have Tape Tasks attached to them. Until they do it will display a status which will inhibit processing. Note that an Array Task begins *before* the time when it is really processing data, and that this is the means for requesting the pre-mounting of tapes. The task will post the desired data tape VSNs on a display as a request to the operator to mount the reels, and will remove the requests as reels are mounted and the associated Tape Tasks attach to the array. The operator can mount the tapes in any order on any available PBD.

The Array Task will interact with the Crossbar Task to request a crossbar reconfiguration at the earliest opportunity if one is needed. When all tapes are ready, the crossbar configuration is set, the Station Tasks are ready and the Archive Task is ready, the Array Task can set the "ready" status; the correlator will process the signals and write results to the archive.

4.3.8 Model Task

The Model Task generates a Model Table (see `MODEL.H` in Appendix H) that contains six-term polynomial expansions of group and phase delays for each station and frequency channel at 2-minute intervals. The table contents are transferred by the Station Tasks into the delay and phase tracking firmware as the correlator job executes (that is, as data are actually correlated). The Model Task creates a table that is bounded by a specified observing time range. The time range can encompass the entire correlator job, or if desired, it can be a short as several minutes.

The Init Model Task is activated shortly before the correlator job begins. Init Model allocates blocks of memory for one or more Model tables, and inserts their base addresses into the Active Base Table. We will most likely have two separate Model tables per job. This will allow double buffering of time blocks of model data for input to the correlator hardware. Model tables that span one hour will be most convenient. While the correlator is using the data in one table, the Model Task will fill the other.

Since the Model Task is run in advance of correlation, it can build the Model Table by multiple passes of several different functions. This is a safer approach than building a complicated model point-by-point seconds before it is required by the correlator phase generators. By constructing the complete delay model in advance, we are able to do some checking and

raise flags if problems are found. This is worthwhile. The VLBA correlator model is fairly complex.

The Model Task calls a succession of functions that fill various elements in the Model Table. First, an empty table is formatted in the memory block previously allocated by Init Model. Delay solution rows are constructed based on the number of stations and baseband channels listed in the job data tables. The Model Table time range was specified as an input to Init Model. There is one model row every 2 minutes (on the even UTC minute) for each baseband channel at each station. Actually, the row time is that of the next 4.096 ms correlator interrupt after the even 2.0 minutes UTC. The Model Task creates four additional rows that precede each source observation, and four rows that follow the end of each observation. These extra solutions allow accurate polynomial fits at the beginning and end of the observing runs. The newly created rows will contain the date (MJD), time (UTC), the station name, the source name and the baseband channel id. The flag words are also set. At this point the task compresses the table in station-baseband-time order. This eliminates the redundant rows that will occur for repetitive sequences of very short observing runs.

Once the Model Table is formatted, the group delays and group delay rates can be calculated and entered. The delay calculations are performed by the CALC Task (Section 4.3.9). After initializing itself one time, the CALC Task is able to calculate delays for any stations and sources at any time in any order. It is very simple then to spawn a CALC process for each row in the table one after another. The information already in the rows is sufficient to direct the CALC Task.

After the CALC Task has filled all of the group delay/rate elements, the Model Task will calculate the phase delay/rate values for each row. In the absence of any dispersive model components, the phase delays may be produced simply from the group delays and channel observing frequencies.

The six-term polynomial expansion is derived from a series of consecutive delay and delay rate pairs for one baseband/station. The spline fit algorithm is described in Section 4.3.9. Because the table has been sorted and compressed, the polynomial fitting can proceed in a straightforward manner. Separate polynomial solutions are obtained for the group and phase delays, and for the group and phase delay rates.

As its last step, the Model Task checks the completed table, particularly looking for error flags. If everything is in order, the "model completed" flag in the job queue table is set to the affirmative and the task terminates.

As the correlator job progresses, its Model Table is read by the phase and

group delay tracking firmware. The table is also read by the archive writer, and essentially copied into the data archive. The polynomials in the Model Table carry the model accountability into the archive itself. The Archive Task will convert the Model Table into a FITS-formatted AIPS calibration table. The polynomial coefficients will in this way accompany the visibility data into AIPS.

The Model/CALC Tasks and the Model Table will require a substantial fraction of the time and memory of one MVME147. A model table row is 200 bytes in length. Since the correlator has 160 input channels, 32 kB of storage are required for each 2-minute model interval. One hour of full correlator operation will need about one MB for the Model Table, so two (double-buffered) tables will require about two Megabytes. CALC takes just under 150 ms to run on the MVME147. But since the CALC group delay solutions currently are non-dispersive, twenty CALC runs will drive the models for 160 correlator inputs. That is, approximately three CPU seconds are required by CALC for one 2-minute model interval. This is a fairly light load on the real-time system.

4.3.9 CALC Task

The CALC Task produces the VLBA interferometry model used by the correlator for group and phase delay tracking. CALC was written and is maintained by the VLBI group working in the NASA Crustal Dynamics Project at Goddard Space Flight Center. It is part of their VLBI data reduction software package that supports geodetic and astrometric observations. The CALC model is considered to be as reliable and well understood as any sophisticated interferometry model code in existence.

It is relatively straightforward for us to use CALC as the correlator model generator. CALC is written in basic Fortran and requires only a few modest modifications to run under VxWorks. The modifications will be done entirely by a preprocessor. The preprocessor itself will thus maintain an accountability of our modifications. The modifications do not affect the CALC solutions, rather they are concerned with Fortran I/O, include statement formats, and block data naming conventions. The CALC preprocessor will also allow us to upgrade our version quickly as new versions of CALC are released by the GSFC group.

CALC will be used in a station-based manner. Station A in each CALC baseline will always be the origin of the geocentric coordinate system. Group delays are calculated at the arrival time of the same wavefront at both

stations in the baseline. That is, the baseline diurnal aberration is included in the CALC model. Since we place station A at the earth center, the station to earth-center delays are all calculated for the same wavefront.

The current version of CALC (version 6.0) is completely non-dispersive. The solar corona model in version 4.0 has been dropped, and there is no ionospheric model. We will probably add dispersive models to the CALC Task as the VLBA project develops.

CALC will retrieve all of its input parameters from the Data Tables via a set of interface functions. It obtains values through subroutine calls that are identical to its calls to the GSFC VLBI Database. The highest level interface subroutines are Fortran. They in turn call C functions that access the data tables. The values returned from the C functions are checked within range limits before being passed into CALC. Some conversion of units will be necessary, but other than that, no CALC input values are arithmetically derived in the interface from data table values, and no CALC parameters are hard coded into the Fortran interface. The parameters in the script file and hence in the data tables are exactly what CALC uses. Mathematical constants used for unit conversions are taken from the CALC common block CMATH.

The correlator hardware requires that group and phase delay models be expressed as polynomials with enough precision to replicate CALC solutions at any time point within the 2-minute CALC interval. After much experimentation, we have found that a 10-point quintic spline fit to the CALC solutions satisfies our accuracy requirements. We will use the double precision Fortran subroutine QUINDF[14]. QUINDF uses the CALC delays and delay rates as input, and produces a six term polynomial solution; one solution for every CALC entry in QUINDF's input window. The solutions near the center of the input window are substantially better than those near the edges. Since QUINDF runs very fast (in comparison to CALC), we will use only the polynomial solution at the center of the input window, and step the window along the sequence of CALC delays one delay at a time. A few extra CALC solutions will precede and follow each observation thereby enabling QUINDF to get equally good fits near the beginning and end of each source observation.

The six term polynomial coefficients will be loaded into the Model table. One polynomial will accompany each CALC solution in the table,

$$\tau = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

[14] ACM Transactions on Mathematical Software, vol. 9, page 258.

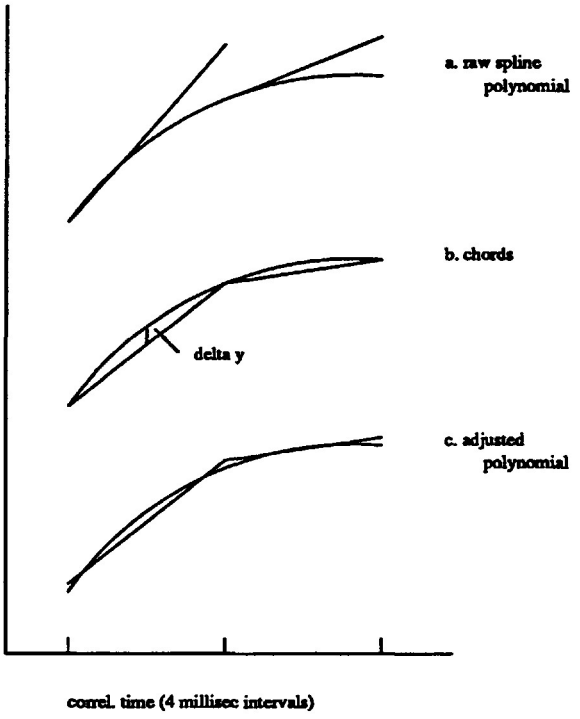


Figure 6: Fitting Splines to CALC Results

The correlator group and phase delay tracking generators read the polynomial table and calculate group and phase delays and delay rates at 4.096 ms intervals. The delay/delay rate pairs initialize number-controlled oscillators whose outputs drive the phase and delay tracking in the correlator hardware. Since the true delay profile is fundamentally a sinusoid with a 24 hour period, the sign of the delay curvature remains the same for hours at a time. The segmented linear delays derived from the spline polynomials deviate from the true delay and produce systematic group and phase delay errors in the correlator output (see figure 6-a). For the delay calculations that are increasingly frequent, the systematic errors will decrease. Evaluating the polynomials at 4 ms intervals reduces the systematic phase delay errors to below one turn of phase at 100 GHz on the earth radius baseline.

However, we have found a simple algorithm that adjusts the polynomial coefficients and essentially eliminates systematic phase delay errors. It is

straightforward to calculate the difference between the model delay and a chord connecting the 4 ms update points. The mid-point difference (dy) is shown in figure 6-b. We use $dy/2$ to offset the phase delay calculated at the 4 ms points, and the slope of the chord is passed as the delay rate. This algorithm is incorporated into the original polynomial coefficients, thereby producing a new set of polynomial coefficients. The adjusted delay and delay rates shown in figure 6-c are then directly derived from the the polynomial with the "throw-back" terms. The new polynomial is shown in terms of the model generator interval Δt (4.096 ms),

$$\tau' = a'_0 + a'_1 t + a'_2 t^2 + a'_3 t^3 + a'_4 t^4 + a'_5 t^5,$$

where the modified coefficients are

$$\begin{aligned}
a'_0 &= a_0 - 8/64 a_2 \Delta t^2 - 12/64 a_3 \Delta t^3 - 14/64 a_4 \Delta t^4 - 15/64 a_5 \Delta t^5 \\
a'_1 &= a_1 - 24/64 a_3 \Delta t^2 - 48/64 a_4 \Delta t^3 - 70/64 a_5 \Delta t^4 \\
a'_2 &= a_2 - 48/64 a_4 \Delta t^2 - 120/64 a_5 \Delta t^3 \\
a'_3 &= a_3 - 80/64 a_5 \Delta t^2 \\
a'_4 &= a_4 \\
a'_5 &= a_5
\end{aligned}$$

The adjusted polynomial replaces the original spline fit polynomial in the Model Table.

The delay rate polynomial is derived by expanding the slope of the chords shown in figure 6-b, such that

$$\dot{\tau} = b_0 + b_1 t + b_2 t^2 + b_3 t^3 + b_4 t^4,$$

where

$$\begin{aligned}
b_0 &= a'_1 + a'_2 \Delta t + a'_3 \Delta t^2 + a'_4 \Delta t^3 + a'_5 \Delta t^4 \\
b_1 &= 2a'_2 + 3a'_3 \Delta t + 4a'_4 \Delta t^2 + 5a'_5 \Delta t^3 \\
b_2 &= 3a'_3 + 6a'_4 \Delta t + 10a'_5 \Delta t^2 \\
b_3 &= 4a'_4 + 10a'_5 \Delta t \\
b_4 &= 5a'_5
\end{aligned}$$

The time variable, t, for both polynomials is a count of 4.096 ms interrupts. That is, t ranges from 0 to 29296. When $t = 0$, the UTC time is the time of the next 4.096 ms interrupt immediately following the even 2 minutes UTC.

4.3.10 Archive Tasks

Visibilities for baselines and channels are summed in cells in the LTA. The Archive Task will determine which cells contain its data and will prepare a suitable index vector, which it will load into the LTA. At the end of each integration cycle the LTA switches banks to allow the task to read out the data for each array that it manages.

The precise details of the index vector mechanism of the LTA are still under discussion. At present it appears that the "index vector" will really be a list of stations (an array) and observing modes, and that the LTA hardware will compute the index vector in real time to sum the STA into the LTA. Likewise, the hardware will assist in "gathering" data for each array, as described by a list of stations and the mode, and transmitting the data into the RT system through a DMA (direct memory access) channel controller.

There will be an instance of the Archive Task for each project of each job. The Archive Task will maintain a set of buffers for its project. It will transfer the current integration into one of its buffers by using its index vector and the DMA controller. It will contend with other instances of the Archive Task for use of the DMA controller, with access interlocked by a semaphore. This procedure will rearrange the visibilities into any desired order. Successive integration cycles can be transferred into successive portions of the buffer.

When a buffer is full it will be written to the archive device; to do this the Archive Tasks will contend for access to the device, interlocked with a semaphore. While the contention and I/O are in progress the task will switch to another buffer and will continue transferring visibilities from the LTA.

Each block written to the archive will be quite large, about 500 kB, and will have a complete FITS header, which the Archive Task will construct in the buffer during the integration cycle, plus various FITS extension records.

The effect of these tactics is that the separate Archive Tasks for each job will separately extract and concatenate their visibilities, doing so with their own integration periods. They will write large blocks of these visibilities to the archive in nearly distribution format.

A technical point is that the information transferred from the LTA by use of the index vector should land in the buffer with appropriate weights attached to the visibilities, in order to minimize the computing work needed in the Archive Task. Also, space in the data structure will be provided to hold the baseline codes; this will be handled automatically by the index

vector mechanism.

Another technical point is that the DMA transmissions must be in short bursts to permit experiments with short integration times to transfer their results concurrently with the transfers for large spectral line experiments.

It may be appropriate to buffer the visibility data into even larger sets of multiple blocks. If so, the SCSI disk attached to the CPU containing the Archive Task can be used to hold the data temporarily. This may be especially useful if the archive device has a long start-stop time, which would favor dumping data from the disk in large bursts.

There has been discussion of the need for a digital filter to be applied to the output visibilities in certain cases. Such a filter would be under the control of the Archive Task.

The real-time hardware configuration (Figure 2) implies that DMA bandwidth used to transfer visibilities from the LTA to the RT RAM will contend with DMA bandwidth used to write the archive medium. If either interface is SCSI then the built-in SCSI channel of the MVME147 CPU can be used to avoid the contention.

Note that clock analysis jobs will work from disk, and so the Archive Task must be able to write visibilities to disk as well as to tape.

4.3.11 Clock Tasks

The Clock Task runs in the real-time system and searches for fringes from selected calibrator source observations as they are correlated. The fringe search and fitting algorithms produce station-based delay and delay rate residuals (with error bars) that are fed back into the DBMS, processed and entered as clock offset corrections in subsequent script files.

There will be two modes of operation of the Clock Task. The "routine" mode in which fringe searching will proceed for all stations (per array) simultaneously, and the "wide-search" mode used to search for "lost" clocks. Both modes will use the same software modules. The "wide-search" mode will search fewer baselines in exchange for more spectral channels.

A typical "wide-search" on a single baseline might be 32 time points (1 sec each) by 1024 spectral channels. The execution time would be roughly 10 sec (on a MVME147). This is quite adequate since searching for lost clocks will be a procedure that is controlled manually by the operators and requires iterative re-correlation at stepped clock offsets. A 10 sec pause for the 2-d FFT is an acceptably brief period in the overall process of "wide-searching".

The input data will come from the LTA at roughly 6 kB/sec for the "routine" calibrator mode (190 baselines, 8 channels, 2 sec integrations), and as high as 16 kB/sec/baseline for the "wide-search" mode (1 baseline, 1024 channels, 1 sec integration). The Archive tasks will write the input data to a hard disk.

The Clock Task is located in one of the VME systems, probably sharing a CPU with the Archive Task. Separate instances of the Clock Task will be spawned by Archive tasks whenever they have clock measurement data on disk, ready to analyze. The Clock Tasks will execute at reduced priority.

The fringe search algorithm is based on the traditional FFT search commonly found in VLBI fringe fitting programs. Time sequences of data are transformed into fringe frequency spectra, and the cross-correlation spectra are transformed into delay lags. We will probably pad the transforms by a factor of 2 in the time axis and a factor of 4 in frequency.

Global fringe fitting in the real-time system is an option that is currently under discussion. The global least-squares fit algorithm in AIPS can be made to work in a non-AIPS environment, but it may be a needless complication. The clock calibration sources will be strong and have fairly simple structure. A more rudimentary fitting algorithm will probably be quicker, safer and sufficient.

Failure to obtain a satisfactory clock solution in routine processing will result in an operator warning message, and recycling of data tapes will be inhibited. Such an occurrence will indicate a violation of the fundamental assumptions that the clock delays of the VLBA are well-behaved and that the real-time fringe checks will detect all glitches in advance. The clock data file on the hard disk will be retained on the hard disk to permit an analysis of the cause of the failure of fringe fitting.

A Glossary

The correlator project uses both standard VLBA jargon and its own specialized vocabulary. The following entries include as well some relational database management terminology which is used throughout this document.

archive. Eternal repository of correlator output data, containing all information necessary for any subsequent processing or interpretation.

array. Set of one or more **stations** observing simultaneously and with identical (or compatible) instrumental settings, over a specific, not necessarily contiguous, range of time. Arrays are the atoms in the process of correlator scheduling; one or more comprise a **project** at the next level of structure.

attribute. Relational DBMS jargon for a collection of similar data elements representing different instances or **tuples**; conceptually, a "column" of a **relation**, or more traditionally, a particular "field" in every "record" in a file.

baseline. Geometrically, the vector connecting a pair of **stations**; by extension, the cross-power spectrum obtained by cross-correlation of data from these stations.

CCC — correlator control computer. Somewhat outdated acronym inherited from early correlator plans, referring to the general-purpose computing environment (currently diskless Sun workstations with a single file server) running the Unix(-like) operating system.

channel. Single band of RF frequencies, in a single hand of circular polarization, converted in station electronics to IF and then to baseband, digitized, and recorded. Equivalent to an "IF" at the VLA.

clock. Atomic frequency standard at a **station**; by extension, the time information obtained by counting cycles of this frequency and recorded with sampled data; also, somewhat ambiguously, the offset between this station atomic time and UTC.

correlator time. Time as defined by the correlator's fundamental 32-MHz clock signal. The correlator hardware operates on several fixed cycles in these units, which correspond to equivalent or *longer* intervals of

observe time via the **speedup** factor. Absolute correlator time, presumably coincident with local civil time, can be used to schedule events in the real-time control system.

crossbar. 24-in to 20-out **station** configuration switch, 8 **channels** deep, interconnecting **PBIs** to **FFT** inputs. The name is slightly misleading, implying full matrix switching; only a sparse, quasi-diagonal subset is actually implemented, selecting among four inputs for each output.

FFT — Fourier transform section. 'F' hardware of the 'FX' correlator. Receives data from the **PBIs** via the **crossbar**, into 20 **station** inputs, each 8 **channels** deep. Besides the fundamental FFT operation, also implements fringe phase and fine (i.e., fractional sample) delay tracking. Each of 20 outputs is again 8 channels deep.

job. Set of one or more **project** groups, combined to optimize correlator scheduling. Jobs are the elements manipulated in the real-time system's **queue** by the scheduler.

LTA — long-term accumulator. Integrates **baseline** spectra to limit rate of data transmitted to the **Archive**.

MAC — cross-multiplier/accumulator section. 'X' hardware of the 'FX' correlator. 20-by-20 upper-triangular matrix, 8 **channels** deep, with inputs hard-wired from the **FFT**. Performs pairwise cross-multiplication of two input **station** spectra at each matrix position, and short-term accumulation of the resulting 210 **baseline** auto- or cross-power spectra. Special polarization mode also processes mixed hands from adjacent channel inputs.

MCB — monitor-&-control bus. VLBA communications standard for local control of all VLBA station equipment; used in the correlator for direct control of **PBDs** from the **RT** system.

observe time. "Real" time as it passes at an observing **station**, marked by the sample **clock** and time digits recorded there. Absolute observe time is a good approximation to UTC, differing by the station **clock** offset. Relative intervals of observe time correspond to equivalent or *shorter* durations of **correlator time** via the **speedup** factor.

PBD — playback drive. VLBI data input to correlator; only remaining element of former Data Playback System (DPS). The correlator's final complement is expected to be 24 PBDs.

PBI — playback interface. Recovers recorded samples from **PBD** signals. Formally part of VLBA Data Recording and Playback subsystem, but integrated into correlator for design efficiency. Each of 24 PBIs is dedicated to a partner PBD, and delivers up to 8 **channels** of 32-Msmp/s data to the correlator proper through the **crossbar**. Also implements coarse (i.e., integral sample) delay tracking.

project. Set of one or more **array** observations arising from a single scientific proposal. VLBA scheduling requirements may necessitate subdividing a proposal into several projects. One or more projects are integrated into a **job** during correlator scheduling.

queue. Set of **jobs** available for initiation by the real-time scheduler.

relation. Relational DBMS jargon for a "table" linking **tuples** of **attributes**. Relations should be *normalized*, meaning roughly that tuples should be unique in all attributes, without redundant entries.

RT — real-time computer system. System of single-board 680*x*0-based VME computers (mainly MVME147 modules), running the VxWorks real-time operating system.

script. Intermediate data format, used to transmit the results of database queries to the correlator control system. Encoded in printing ASCII characters, human-readable and (if necessary) editable.

speedup. Factor (= 1, 2, or 4) by which **observe time** is accelerated when played back at higher speed.

station. Fundamentally, a radio telescope at some location, either one of the 10 VLBA sites or elsewhere; by extension, often used to refer to data originating at a station along its entire path through the correlator, in particular through the **FFT**, **MAC**, and **LTA** sections.

tuple. Relational DBMS jargon for a collection of logically related but dissimilar data elements representing different **attributes**; conceptually, a "row" of a **relation**, or more traditionally, one of many identically-formatted "records" in a file.

B model_script.tx Example Job Script

```
-------------------------------------------------------------------
-r--r--r--  1 jbenson  vlb         97799 Sep 26 11:06 s.model_script.tx
   in directory /home/ccc/vlb/fxcorr/src/code/SCCS
-------------------------------------------------------------------


!* Model script file for VLBA Correlator Job *!
!* 3 stations, 8 baseband channels, 2 minute scans  *!
!* 30 July 1989 -- JMB  *!

!*----- Job Control Card  ----------------*!
!table 'job'!
  jobid = 11021    program = 'VW23G'
  n_drives = 3  n_chans = 8   n_fft_pts = 512  n_stns = 3    fft_factor = 8
  date_start = 80Sep22  start = 12h00m00.0s
  date_stop  = 80Sep22  stop  = 12h30m00.0s
!row!
!endtable!

!*----- Observations Table -----------*!
!table 'observations'!
  date = 80Sep22  start = 12h00m00.0s  stop = 12h02m00.0s source ='BLLAC'
  name = 'HY'  array_id = 100 !row!
  date = 80Sep22  start = 12h02m00.0s  stop = 12h04m00.0s source ='VIRGO'
  name = 'HY'  array_id = 100 !row!
  date = 80Sep22  start = 12h04m00.0s  stop = 12h06m00.0s source ='BLLAC'
  name = 'HY'  array_id = 100 !row!
  date = 80Sep22  start = 12h00m00.0s  stop = 12h02m00.0s source ='BLLAC'
  name ='MPI'   array_id = 100 !row!
  date = 80Sep22  start = 12h02m00.0s  stop = 12h04m00.0s source ='VIRGO'
  name ='MPI'   array_id = 100 !row!
  date = 80Sep22  start = 12h04m00.0s  stop = 12h06m00.0s source ='BLLAC'
  name ='MPI'   array_id = 100 !row!
  date = 80Sep22  start = 12h00m00.0s  stop = 12h02m00.0s source ='BLLAC'
  name ='GB'    array_id = 100 !row!
  date = 80Sep22  start = 12h02m00.0s  stop = 12h04m00.0s source ='VIRGO'
  name ='GB'    array_id = 100 !row!
```

```
  date = 80Sep22  start = 12h04m00.0s  stop = 12h06m00.0s source ='BLLAC
  name ='GB'    array_id = 100 !row!
!endtable!


!*----- Correlator configuration -------------*!
!table 'correl'!
  name = 'all'   date = 80Sep22   time = 00h00m00.0s
  fftsize  = 512   interleav = 0   overlap  = 1   interpol  = 1
  filter   = 0     window    = 'uniform'
  time_avg = 2.0   spect_avg = 2                                  !row!
  time_avg  = 2.0                                                 !row!
!endtable!


!*----- Sampler/Formatter/Recorder -----------*!
!table 'formatter'!
  name = 'all'  date = 80Sep22  time = 00h00m00.0s
  sample_rate = 32.0e+6     sample_mode = '4-level'  format = 'VLBA'
  formatter_mode = 2.0      barrel_swx  = 'on'
  track_swx = 'off'         write_speed = 90
!row!
!endtable!

!*----- Baseband Channels and Frequencies ----*!
!table 'channels'!

  name = 'all'  date = 80Sep22  time = 12h00m00.0s
  bbconv = 1 fe = 6cm ifchan = A  net_side = +1 sky_freq = 4.994990e+9
           bbfilter = 4M  fchanu = 1 fchanl = 0              !row!
  bbconv = 2 fe = 6cm ifchan = A  net_side = +1 sky_freq = 4.998990e+9
           bbfilter = 4M  fchanu = 2 fchanl = 0              !row!
  bbconv = 3 fe = 6cm ifchan = C  net_side = +1 sky_freq = 4.994990e+9
           bbfilter = 4M  fchanu = 3 fchanl = 0              !row!
  bbconv = 4 fe = 6cm ifchan = C  net_side = +1 sky_freq = 4.998990e+9
           bbfilter = 4M  fchanu = 4 fchanl = 0              !row!
  bbconv = 5 fe = 6cm ifchan = A  net_side = +1 sky_freq = 5.002990e+9
           bbfilter = 4M  fchanu = 5 fchanl = 0             !row!
  bbconv = 6 fe = 6cm ifchan = A  net_side = +1 sky_freq = 5.006990e+9
```

```
          bbfilter = 4M  fchanu = 6 fchanl = 0            !row!
  bbconv = 7 fe = 6cm ifchan = C  net_side = +1 sky_freq = 5.002990e+9
          bbfilter = 4M  fchanu = 7 fchanl = 0            !row!
  bbconv = 8 fe = 6cm ifchan = C  net_side = +1 sky_freq = 5.006990e+9
          bbfilter = 4M  fchanu = 8 fchanl = 0            !row!

  name = 'all'  date = 80Sep22  time = 12h02m00.0s
  bbconv = 1 fe = 6cm ifchan = A  net_side = +1 sky_freq = 4.994990e+9
          bbfilter = 16M  fchanu = 1 fchanl = 0             !row!
  bbconv = 2 fe = 6cm ifchan = C  net_side = +1 sky_freq = 5.010990e+9
          bbfilter = 16M  fchanu = 2 fchanl = 0             !row!
  bbconv = 3              fchanu = 0 fchanl = 0             !row!
  bbconv = 4              fchanu = 0 fchanl = 0             !row!
  bbconv = 5              fchanu = 0 fchanl = 0             !row!
  bbconv = 6              fchanu = 0 fchanl = 0             !row!
  bbconv = 7              fchanu = 0 fchanl = 0             !row!
  bbconv = 8              fchanu = 0 fchanl = 0             !row!

  name = 'all'  date = 80Sep22  time = 12h04m00.0s
  bbconv = 1 fe = 6cm ifchan = A  net_side = +1 sky_freq = 4.994990e+9
          bbfilter = 4M  fchanu = 1 fchanl = 0              !row!
  bbconv = 2 fe = 6cm ifchan = A  net_side = +1 sky_freq = 4.998990e+9
          bbfilter = 4M  fchanu = 2 fchanl = 0              !row!
  bbconv = 3 fe = 6cm ifchan = C  net_side = +1 sky_freq = 4.994990e+9
          bbfilter = 4M  fchanu = 3 fchanl = 0              !row!
  bbconv = 4 fe = 6cm ifchan = C  net_side = +1 sky_freq = 4.998990e+9
          bbfilter = 4M  fchanu = 4 fchanl = 0              !row!
  bbconv = 5 fe = 6cm ifchan = A  net_side = +1 sky_freq = 5.002990e+9
          bbfilter = 4M  fchanu = 5 fchanl = 0             !row!
  bbconv = 6 fe = 6cm ifchan = A  net_side = +1 sky_freq = 5.006990e+9
          bbfilter = 4M  fchanu = 6 fchanl = 0            !row!
  bbconv = 7 fe = 6cm ifchan = C  net_side = +1 sky_freq = 5.002990e+9
          bbfilter = 4M  fchanu = 7 fchanl = 0            !row!
  bbconv = 8 fe = 6cm ifchan = C  net_side = +1 sky_freq = 5.006990e+9
          bbfilter = 4M  fchanu = 8 fchanl = 0            !row!
!endtable!

!*----- Clocks Table --------------------*!
!table 'clocks'!
```

```
  name = 'HY'
  date = 80Sep22  time = 00h00m00.0s  offset = -1.00e-6  !row!
                  time = 06h00m00.0s  offset = -6.0e-6  !row!
                  time = 18h00m00.0s  offset = -18.0e-6 !row!
  date = 80Sep23  time = 01h00m00.0s  offset = -25.0e-6  !row!
  name = 'MPI'
  date = 80Sep22  time = 00h00m00.0s  offset = -1.00e-6  !row!
                  time = 06h00m00.0s  offset = -6.0e-6  !row!
                  time = 18h00m00.0s  offset = -18.0e-6 !row!
  date = 80Sep23  time = 01h00m00.0s  offset = -25.0e-6  !row!
  name = 'GB'
  date = 80Sep22  time = 00h00m00.0s  offset = -1.00e-6  !row!
                  time = 06h00m00.0s  offset = -6.0e-6  !row!
                  time = 18h00m00.0s  offset = -18.0e-6 !row!
  date = 80Sep23  time = 01h00m00.0s  offset = -25.0e-6  !row!
!endtable!

!*----- Stations Table -------------------*!
!table 'stations'!
  name = 'MPI'
  x =  4.03394212e+6   y =  4.86993120e+05 z =  4.90043183e+06
  axistype = 'altaz'   axisoff = 0.00
!row!
  name = 'HY'
  x =  1.49240669e+06  y = -4.45726733e+06  z =  4.29688210e+06
  axistype = 'altaz'   axisoff  = 0.00
!row!
  name = 'GB'
  x = 8.882882548e+5 y = -4.92448405e+6 z = 3.94413087e+6
  axistype = 'polar'  axisoff  = 0.0
!row!
!endtable!

!*----- Source Table ---------------------*!
!table 'sources'!
  name ='BLLAC'
  ra   = 22h00m39.363s   dec = 42d02'08.57"
  parallax = 0.0
  date = 80Sep23  time = 12h00m00.0s
```

```
  d1ra  = 0.0     d1dec  = 0.0    d2ra  = 0.0   d2dec  = 0.0
!row!
  name ='VIRGO'  epoch = 2000.0
  ra    = 12h30m48.450s   dec  = 12d23'28.49"
!row!
  name = '3C779'
  ra    = 12h12m45.612s   dec  = 31d08'54.55"
!row!
!endtable!

!*----- Tape Table ----------------------*!
!table 'tapes'!
  date = 80Sep22 name = 'HY'
  tapeid = VLBA1012  start = 12h00m28.2s  stop = 24h00m00.0s
!row!
  date = 80Sep23 name = 'HY'
  tapeid = VLBA1013  start = 00h00m31.3s  stop = 12h00m00.0s
!row!
  date = 80Sep22 name = 'MPI'
  tapeid = VLBA1023  start = 12h00m28.2s  stop = 24h00m00.0s
!row!
  date = 80Sep23 name = 'MPI'
  tapeid = VLBA1024  start = 00h00m31.3s  stop = 12h00m00.0s
!row!
  date = 80Sep22 name = 'GB'
  tapeid = VLBA1911  start = 12h00m28.2s  stop = 24h00m00.0s
!row!
  date = 80Sep23 name = 'GB'
  tapeid = VLBA1123  start = 00h00m31.3s  stop = 12h00m00.0s
!row!
!endtable!

!*  Earth postion, velocities : Section B, Astronomical Almanac   *!
!*  Sun position, velocities  : Section C, Astronomical Almanac   *!
!*  Moon position, velocities : Section D, Astronomical Almanac   *!
!table 'solar'!
  date = 80Sep22
  earth_x  = 1.5e+11      earth_y  = 1.0e+10      earth_z  = 1.0e+09
  earth_vx = 1.0e+02      earth_vy = 30.0e+03     earth_vz = 1.0e+01
```

```
  sun_x  = -1.5e+11      sun_y  = -1.0e+10      sun_z  = -1.0e+09
  sun_vx = -1.0e+02      sun_vy = -30.0e+03     sun_vz = -1.0e+01
  moon_x  = 3.844e+08     moon_y  = 1.0e+07       moon_z  = 1.0e+07
  moon_vx = 1.0e+02       moon_vy = 1.0e+03       moon_vz = 1.0e+01
!row!
  date = 80Sep23
  earth_x  = 1.5e+11      earth_y  = 1.0e+10      earth_z  = 1.0e+09
  earth_vx = 1.0e+02      earth_vy = 30.0e+03     earth_vz = 1.0e+01
  sun_x  = -1.5e+11      sun_y  = -1.0e+10      sun_z  = -1.0e+09
  sun_vx = -1.0e+02      sun_vy = -30.0e+03     sun_vz = -1.0e+01
  moon_x  = 3.844e+08     moon_y  = 1.0e+07       moon_z  = 1.0e+07
  moon_vx = 1.0e+02       moon_vy = 1.0e+03       moon_vz = 1.0e+01
!row!
!endtable!

!*--------- UT1-UTC Table -------------------------*!
!table 'UT1'!
  date = 80Sep20 time = 00h00m00.0s ut1utc = 0.0560 !row!
  date = 80Sep21 time = 00h00m00.0s ut1utc = 0.0560 !row!
  date = 80Sep22 time = 00h00m00.0s ut1utc = 0.0560 !row!
  date = 80Sep23 time = 00h00m00.0s ut1utc = 0.0560 !row!
  date = 80Sep24 time = 00h00m00.0s ut1utc = 0.0560 !row!
!endtable!

!*--------- Polar Motion Table --------------------*!
!table 'polar'!
  date = 80Sep20 time = 00h00m00.0s x = -0.0090 y = 0.3190 !row!
  date = 80Sep21 time = 00h00m00.0s x = -0.0090 y = 0.3190 !row!
  date = 80Sep22 time = 00h00m00.0s x = -0.0090 y = 0.3190 !row!
  date = 80Sep23 time = 00h00m00.0s x = -0.0090 y = 0.3190 !row!
  date = 80Sep24 time = 00h00m00.0s x = -0.0090 y = 0.3190 !row!
!endtable!

!*----- Physical Constants (1976, 1984 IAU)--*!
!table 'constants'!
  vlight = 299792.458e3   gauss = 0.01720209       accelgrv = 9.78031846
  gmsun = 1.32712499e+20  gmmoon = 4.90279750e+12  tsecau = 499.00478
  earthrad = 6378145.0    emsmms = 81.3006592      ugrvcn = 6.668e-11
  eflat = 0.00335289      preconst = 5029.0966     relative = 1.0
```

```
  dipolar = 1.0          ephepoch = 2000.0
  etide_lag = 0.0        love_h = 0.60967        love_l = 0.085
  stn_a_x = 0.0001       stn_a_y = 0.0001        stn_a_z = 0.0001
!row!
!endtable!

!QUIT!
```

C QUEUE_TABLE.H Job Queue Header File

```
------------------------------------------------------------------------
-r--r--r--  1 jbenson  vlb          2108 Sep 29 10:09 s.QUEUE_TABLE.H
   in directory /home/ccc/vlb/fxcorr/src/code/SCCS
------------------------------------------------------------------------

/*+  QUEUE_TABLE.H
%% Include file struct for correlator Queue Table.

    The Queue Table holds locations and descriptions of each active
and pending correlator job in the job queue in the real_time system.
---
LANGUAGE: C
ENVIRONMENT: vxWorks
++$ AUDIT TRAIL
1.0  89May10   jbenson   Initial submission
1.1  89Aug07   jbenson  replace *model_ptr with *active_ptr
--$
-*/
#define N_JOBS 100

     struct queue_table {
        int     job_index;      /* job index number                      */
        char    *job_ptr;       /* ptr to job table                      */
        char    *active_ptr;    /* ptr to active_base_table              */
        long    date;           /* date of job entry                     */
        float   time;           /* time of job entry                     */
        int     n_tables;       /* number of input data tables this job  */
        char    filename[80];   /* script file name                      */
        int     priority;       /* job priority level                    */
        int     flag_CALC;      /* CALC completed, 0 = no, 1 = yes       */
        int     flag_job_active;   /* status = -1, error  occurred;
                                      status =  0, job on hold;
                                      status =  1, job ready in queue;
                                      status =  5, job is correlating,
                                      status = 10, job finished normally */
        int     flag_job_archived;
     };
```

D TABLES.H Job Tables Header File

```
-------------------------------------------------------------------
-r--r--r--  1 jbenson  vlb         21347 Sep 19 10:42 s.TABLES.H
   in directory /home/ccc/vlb/fxcorr/src/code/SCCS
-------------------------------------------------------------------

/*+ TABLES.H
%%  Structure declarations for the correlator real-time data tables.

The TABLES.H include file contains the structure declarations for all of the
tables created and filled by the script file reader, and subsequentially read
by tasks in the real-time system.
---
LANGUAGE: C (Include file)
ENVIRONMENT: vxWorks
++$ AUDIT TRAIL
 1.0 89/04/26 jbenson Initial submission
 1.1 89/05/15 fxcorr date and time created 89/05/15 15:27:48 by fxcorr
 1.2 89/05/18 jbenson change time/angles to double, long's to int's.
 1.3 89/05/24 jbenson put structs in alpha order, more documentation
 1.4 89/05/25 jbenson re-load alphabetized file
 1.5 89/07/13 jbenson station coords in constants table
 1.6 89/07/18 jbenson added net_side, sky_freq to channels struct
 1.7 89/07/19 jbenson station-based correl, channels, obs, formatter tables
 1.8 89/07/19 jbenson put bbconv back into channels table
 1.11 89/09/13 jbenson added array_id to observations table
 1.12 89/09/14 jbenson changed formatter_mode from char[16] to float
--$
-*/

/* TABLE KEYWORD DESCRIPTION STRUCTURE */
/* The script keyword table describes the script keyword by name, format,
   and location in the relavent data table. */

struct table_desc {
    char            keyword[16];       /* keyword string */
    char            format[16];        /* keyword value format */
    char           *cp;                /* keyword pointer */
```

```
};

/* JOB TABLE STRUCTURE */

struct job_table {
    char            table_name[16];    /* data table name */
    int             num_rows;          /* number of rows in current table */
    int             row_size;          /* row size */
    int             num_keys;          /* number of keywords in table */
    char           *base_ptr;          /* pointer to first row in table */
    char           *current_row;       /* pointer to the current row */
    struct tables_desc *desc;          /* pointer to current table's keyword
                                        * descriptions */
};

/* DATA TABLE DECLARATIONS  */

#define N_TABLES   13       /* number of data tables declared  */

/* Baseband Channels Table */

struct channels {
    char            name[16];          /* station name */
    int             date;              /* channels config epoch (MJD) */
    double          time;              /* channels config epoch (rad) */
    int             net_side;          /* net sideband counter */
    char            bbfilter[16];      /* baseband converter bandwidth,
                                        * {0-255} */
    char            ifchan[16];        /* baseband converter IF input,
                                        * {'A','B','C','D'} */
    int             bbconv;            /* baseband converter unit number */
    char            fe[16];            /* front-end id (20cm,6cm, etc..) */
    double          sky_freq;          /* sky frequency at zero Hz baseband
                                        * (GHz) */
    int             fchanu;            /* formatter channel which receives
                                        * USB signal */
    int             fchanl;            /* formatter channel which receives
                                        * LSB signal */
};
```

```
/* Clocks Table  */

struct clocks {
    /* Station Clock Errors  */
    char            name[16];          /* station name */
    int             date;              /* clock epoch date (MJD) */
    double          time;              /* clock epoch time (rad) */
    float           offset;            /* station clock offset (sec) */
};

/* Correlator Configuration */

struct correl {
    char            name[16];          /* station name */
    int             date;              /* correl config epoch (MJD) */
    double          time;              /* correl config epoch (rad) */
    int             fftsize;           /* number of spectral points */
    int             interleav;         /* interleaving factor       */
    int             overlap;           /* overlapping factor        */
    int             interpol;          /* interpolation factor      */
    char            filter[16];        /* digital filter option     */
    char            window[16];        /* FFT window selection      */
    float           time_avg;          /* time average interval (secs)  */
    float           spect_avg;         /* spectral channel averaging */
};

/* Physical Constants  */
/* The physical constants will be shifted into #define statements
   after the software integration and debugging is complete  */

struct constants {
    double          vlight;            /* velocity of light (m/s) */
    double          gauss;             /* grav constant (kg-m**3/sec**2) */
    double          accelgrv;          /* accel grav at equator (m/sec**2) */
    double          gmsun;             /* solar mass * newt grav
                                        * (m**3/sec**2)   */
    double          gmmoon;            /* lunar mass * newt grav
                                        * (m**3/sec**2)   */
```

```
    double          tsecau;             /* au in light secs (sec/au)  */
    double          earthrad;           /* equatorial radius (m) */
    double          emsmms;             /* earth mass/moon mass   */
    double          ugrvcn;             /* newt grav const (m**3/kg-sec**2) */
    double          eflat;              /* sqr eccentricity of earth shape  */
    double          dipolar;            /* diurnal polar motion scale ellips */
    double          etide_lag;          /* earth tides lag angle (rads) */
    double          love_l;             /* earth tide love number l */
    double          love_h;             /* earth tide love number h */
    double          ephepoch;           /* ephmeris ref epoch (years)  */
    double          preconst;           /* precession const (arcsec/century) */
    double          relative;           /* post newt expansion parm          */
    double          stn_a_x;            /* geocentric coord station a (m)  */
    double          stn_a_y;            /* geocentric coord station a (m)  */
    double          stn_a_z;            /* geocentric coord station a (m)  */
};

/* Sampler/Formatter/Recorder Table  */

struct formatter {
    char            name[16];           /* station name */
    int             date;               /* formatter config epoch (MJD) */
    double          time;               /* formatter config epoch (rad) */
    float           sample_rate;        /* sampler rate (MHz) */
    char            sample_mode[16];    /* sampler mode, 2 or 4 levels */
    char            format[16];         /* format = 'VLBA' or 'MKIII' */
    float           formatter_mode;     /* formatter mode */
    char            barrel_swx[16];     /* barrel switch, on or off */
    char            track_swx[16];      /* track switch, on or off */
    float           write_speed;        /* recorder write speed (inches/sec) */
};

/* Job Card Table */

struct job {
    char            jobid[16];          /* job id for current script file */
    char            program[16];        /* observing program id */
    int             date_start;         /* job start date */
    int             date_stop;          /* job stop date */
```

```
    double          start;          /* job start time UTC */
    double          stop;           /* job stop time UTC */
    int             n_drives;       /* max number similtaneous PBD's */
    int             n_chans;        /* max number similtaneous channels */
    int             n_fft_pts;      /* largest fft size  */
    int             n_stns;         /* number stations */
    int             fft_factor;     /* fft multiplicity factor */
};

/* Observations Table       */

struct observations {
    char            source[16];     /* source name */
    char            name[16];       /* station name */
    int             array_id;       /* this obs belongs to array_id */
    int             date;           /* observe date (MJD) */
    double          start;          /* observe start time (rads) */
    double          stop;           /* observe stop time (rads) */
};

/* Polar Motion Table  */

struct polar {
    /* Earth's Spin Axis Position Offsets */
    int             date;           /* epoch date (MJD)  */
    double          time;           /* epoch time (rad)  */
    float           x;              /* x offset (milliarcsec)  */
    float           y;              /* y offset (milliarcsec)  */
};

/* Sun, Earth, Moon positions (J2000.0)  */

struct solar {
    int             date;           /* solar system epoch date (MJD)  */
    float           earth_x;        /* earth position (meters) */
    float           earth_y;        /* in solar system baricentric */
    float           earth_z;        /* coordinate system */
    float           earth_vx;       /* earth velocity in x */
    float           earth_vy;       /* earth velocity in y */
```

```
    float           earth_vz;           /* earth velocity in z */
    float           moon_x;             /* moon position */
    float           moon_y;             /* in earth-moon baricentric */
    float           moon_z;             /* coordinate system */
    float           moon_vx;            /* moon velocity in x */
    float           moon_vy;            /* moon velocity in y */
    float           moon_vz;            /* moon velocity in z */
    float           sun_x;              /* sun position */
    float           sun_y;
    float           sun_z;
    float           sun_vx;             /* sun velocity in x */
    float           sun_vy;             /* sun velocity in y */
    float           sun_vz;             /* sun velocity in z */
};

/* Sources Table            */

struct sources {
    char            name[16];           /* source name */
    double          ra;                 /* source ra (J2000.0) */
    double          dec;                /* source dec (J2000.0) */
    float           epoch;              /* position epoch (J2000.0) */
    float           parallax;           /* source parallax (arcsec) */
    int             date;               /* position derivatives epoch, date */
    double          time;               /* position derivatives epoch, time */
    double          d1ra;               /* first deriv of ra */
    double          d1dec;              /* first deriv of dec */
    double          d2ra;               /* second deriv of ra */
    double          d2dec;              /* second deriv of dec */
};

/* Stations Table */

struct stations {
    char            name[16];           /* station name */
    double          x;                  /* geocentric x coord (meters) */
    double          y;                  /* geocentric y coord (meters) */
    double          z;                  /* geocentric z coord (meters) */
    double          x_phs;              /* geocentric coords for separate
```

```
                                          phase delay model (if req.) */
    double          y_phs;              /* geocentric y coord (meters) */
    double          z_phs;              /* geocentric z coord (meters) */
    char            axistype[12];       /* antenna mount type */
    float           axisoff;            /* non-intersecting axis offset
                                         * (meters) */
};

/* Tapes Table              */

struct tapes {
    char            name[16];           /* recording station name */
    char            tapeid[16];         /* tape reel id */
    int             date;               /* begin recording, date (MJD) */
    double          start;              /* begin recording, time (rad) */
    double          stop;               /* end recording, time (rad) */
};

/* UT1 - UTC table    */

struct UT1 {
    int             date;               /* epoch date (MJD) */
    double          time;               /* epoch time (rad) */
    double          ut1utc;             /* ut1 - utc correction (secs) */
};
```

E TABLES_DESC.H Script Keywords Header File

```
------------------------------------------------------------------------
-r--r--r--  1 jbenson  vlb         23569 Sep 19 10:42 s.TABLES_DESC.H
   in directory /home/ccc/vlb/fxcorr/src/code/SCCS
------------------------------------------------------------------------

/*+ TABLES_DESC.H
%% Descriptive Structures for the correlator real-time data tables.

   The table descriptions below list the keywords for each data
table as they are represented in the script files. Along with each
keyword is its data type and a pointer offset to its location with
respect to the beginning of the appropriate data table row.
   The description structures are declared once, as part of the
job queue table initialization.
---
LANGUAGE: C (Include file)
ENVIRONMENT: VxWorks
++$ AUDIT TRAIL
 1.0 89/04/26 jbenson Initial submission
 1.1 89/05/15 fxcorr date and time created 89/05/15 15:28:16 by fxcorr
 1.2 89/05/24 jbenson put structs in alpha order, lined up columns by hand.
 1.3 89/05/25 jbenson re-load alphabetized file.
 1.4 89/07/13 jbenson add stn_a_x, y, z to constants table.
 1.5 89/07/18 jbenson added net_side, sky_freq to channels description
 1.7 89/07/19 jbenson put bbconv back into channels table
 1.6 89/07/19 jbenson station-based correl, channels, obs, formatter tables
 1.8 89/09/13 jbenson add array_id to observations table
 1.9 89/09/14 jbenson change formatter_mode from char to float
--$
-*/

/* Channels Table Description */

struct channels channels_onerow;
#define channels_P(xx) (char *)(&channels_onerow.xx)
struct table_desc chan_desc[] = {
    "name",     "char",  channels_P (name[0]) - channels_P (name[0]),
```

```
    "date",     "date",  channels_P (date) - channels_P (name[0]),
    "time",     "time",  channels_P (time) - channels_P (name[0]),
    "net_side", "int",   channels_P (net_side) - channels_P (name[0]),
    "bbfilter", "char",  channels_P (bbfilter) - channels_P (name[0]),
    "ifchan",   "char",  channels_P (ifchan) - channels_P (name[0]),
    "bbconv",   "int",   channels_P (bbconv) - channels_P (name[0]),
    "fe",       "char",  channels_P (fe) - channels_P (name[0]),
    "sky_freq", "double",channels_P (sky_freq) - channels_P (name[0]),
    "fchanu",   "int",   channels_P (fchanu) - channels_P (name[0]),
    "fchanl",   "int",   channels_P (fchanl) - channels_P (name[0])
};

/* Clocks Table Description */

struct clocks   clocks_onerow;
#define clocks_P(xx) (char *)(&clocks_onerow.xx)
struct table_desc clocks_desc[] = {
    "name",   "char",  clocks_P (name[0]) - clocks_P (name[0]),
    "date",   "date",  clocks_P (date) - clocks_P (name[0]),
    "time",   "time",  clocks_P (time) - clocks_P (name[0]),
    "offset", "float", clocks_P (offset) - clocks_P (name[0])
};

/* Physical Constants Table Description */

struct constants constants_onerow;
#define constants_P(xx) (char *)(&constants_onerow.xx)
struct table_desc constants_desc[] = {
    "vlight",   "double", constants_P (vlight) - constants_P (vlight),
    "gauss",    "double", constants_P (gauss) - constants_P (vlight),
    "accelgrv", "double", constants_P (accelgrv) - constants_P (vlight),
    "gmsun",    "double", constants_P (gmsun) - constants_P (vlight),
    "gmmoon",   "double", constants_P (gmmoon) - constants_P (vlight),
    "tsecau",   "double", constants_P (tsecau) - constants_P (vlight),
    "earthrad", "double", constants_P (earthrad) - constants_P (vlight),
    "emsmms",   "double", constants_P (emsmms) - constants_P (vlight),
    "ugrvcn",   "double", constants_P (ugrvcn) - constants_P (vlight),
    "eflat",    "double", constants_P (eflat) - constants_P (vlight),
    "dipolar",  "double", constants_P (dipolar) - constants_P (vlight),
```

```
    "etide_lag","double", constants_P (etide_lag) - constants_P (vlight),
    "love_l",   "double", constants_P (love_l) - constants_P (vlight),
    "love_h",   "double", constants_P (love_h) - constants_P (vlight),
    "ephepoch", "double", constants_P (ephepoch) - constants_P (vlight),
    "preconst", "double", constants_P (preconst) - constants_P (vlight),
    "relative", "double", constants_P (relative) - constants_P (vlight),
    "stn_a_x",  "double", constants_P (stn_a_x) - constants_P (vlight),
    "stn_a_y",  "double", constants_P (stn_a_z) - constants_P (vlight),
    "stn_a_z",  "double", constants_P (stn_a_z) - constants_P (vlight)
};

/* Correl Table Description */

struct correl   correl_onerow;
#define correl_P(xx) (char *)(&correl_onerow.xx)
struct table_desc correl_desc[] = {
    "name",      "char",  correl_P (name[0]) - correl_P (name[0]),
    "date",      "date",  correl_P (date) - correl_P (name[0]),
    "time",      "time",  correl_P (time) - correl_P (name[0]),
    "fftsize",   "int",   correl_P (fftsize) - correl_P (name[0]),
    "interleav", "int",   correl_P (interleav) - correl_P (name[0]),
    "overlap",   "int",   correl_P (overlap) - correl_P (name[0]),
    "interpol",  "int",   correl_P (interpol) - correl_P (name[0]),
    "filter",    "char",  correl_P (filter) - correl_P (name[0]),
    "window",    "char",  correl_P (window) - correl_P (name[0]),
    "time_avg",  "float", correl_P (time_avg) - correl_P (name[0]),
    "spect_avg", "float", correl_P (spect_avg) - correl_P (name[0])
};

/* Formatter Table Description */

struct formatter formatter_onerow;
#define formatter_P(xx) (char *)(&formatter_onerow.xx)
struct table_desc formatter_desc[] = {
    "name",        "char",  formatter_P(name[0]) - formatter_P(name[0]),
    "date",        "date",  formatter_P(date) - formatter_P(name[0]),
    "time",        "time",  formatter_P(time) - formatter_P(name[0]),
    "sample_rate", "float", formatter_P(sample_rate) - formatter_P (name[0]),
    "sample_mode", "char",  formatter_P(sample_mode) - formatter_P (name[0]),
```

```
    "format",      "char",  formatter_P(format) - formatter_P (name[0]),
    "formatter_mode","float",formatter_P(formatter_mode) - formatter_P (name[0]),
    "barrel_swx",  "char",  formatter_P(barrel_swx) - formatter_P (name[0]),
    "track_swx",   "char",  formatter_P(track_swx) - formatter_P (name[0]),
    "write_speed", "float", formatter_P(write_speed) - formatter_P (name[0])
};

/* Job Table Description */

struct job      job_onerow;
#define job_P(xx) (char *)(&job_onerow.xx)
struct table_desc job_desc[] = {
    "jobid",      "char", job_P (jobid) - job_P (jobid),
    "program",    "char", job_P (program) - job_P (jobid),
    "date_start", "date", job_P (date_start) - job_P (jobid),
    "date_stop",  "date", job_P (date_stop) - job_P (jobid),
    "start",      "time", job_P (start) - job_P (jobid),
    "stop",       "time", job_P (stop) - job_P (jobid),
    "n_drives",   "int",  job_P (n_drives) - job_P (jobid),
    "n_chans",    "int",  job_P (n_chans) - job_P (jobid),
    "n_fft_pts",  "int",  job_P (n_fft_pts) - job_P (jobid),
    "n_stns",     "int",  job_P (n_stns) - job_P (jobid),
    "fft_factor", "int",  job_P (fft_factor) - job_P (jobid)
};

/* Observations table Description */

struct observations obs_onerow;
#define obs_P(xx)  (char *)(&obs_onerow.xx)
struct table_desc obs_desc[] = {
    "source",    "char", obs_P (source[0]) - obs_P (source[0]),
    "name",      "char", obs_P (name[0]) - obs_P (source[0]),
    "array_id",  "int",  obs_P (array_id - obs_P (source[0]),
    "date",      "date", obs_P (date) - obs_P (source[0]),
    "start",     "time", obs_P (start) - obs_P (source[0]),
    "stop",      "time", obs_P (stop) - obs_P (source[0]),
};

/* Polar Table Description */
```

```
struct polar    polar_onerow;
#define polar_P(xx) (char *)(&polar_onerow.xx)
struct table_desc polar_desc[] = {
    "date", "date",  polar_P (date) - polar_P (date),
    "time", "time",  polar_P (time) - polar_P (date),
    "x",    "float", polar_P (x) - polar_P (date),
    "y",    "float", polar_P (y) - polar_P (date)
};

/* Solar System Table Description */

struct solar    solar_onerow;
#define solar_P(xx) (char *)(&solar_onerow.xx)
struct table_desc solar_desc[] = {
    "date",     "date",  solar_P (date) - solar_P (date),
    "earth_x",  "float", solar_P (earth_x) - solar_P (date),
    "earth_y",  "float", solar_P (earth_y) - solar_P (date),
    "earth_z",  "float", solar_P (earth_z) - solar_P (date),
    "earth_vx", "float", solar_P (earth_vx) - solar_P (date),
    "earth_vy", "float", solar_P (earth_vy) - solar_P (date),
    "earth_vz", "float", solar_P (earth_vz) - solar_P (date),
    "moon_x",   "float", solar_P (moon_x) - solar_P (date),
    "moon_y",   "float", solar_P (moon_y) - solar_P (date),
    "moon_z",   "float", solar_P (moon_z) - solar_P (date),
    "moon_vx",  "float", solar_P (moon_vx) - solar_P (date),
    "moon_vy",  "float", solar_P (moon_vy) - solar_P (date),
    "moon_vz",  "float", solar_P (moon_vz) - solar_P (date),
    "sun_x",    "float", solar_P (sun_x) - solar_P (date),
    "sun_y",    "float", solar_P (sun_y) - solar_P (date),
    "sun_z",    "float", solar_P (sun_z) - solar_P (date),
    "sun_vx",   "float", solar_P (sun_vx) - solar_P (date),
    "sun_vy",   "float", solar_P (sun_vy) - solar_P (date),
    "sun_vz",   "float", solar_P (sun_vz) - solar_P (date)
};

/* Sources Table Description */

struct sources  src_onerow;
```

```
#define src_P(xx) (char *)(&src_onerow.xx)
struct table_desc src_desc[] = {
    "name",     "char",   src_P (name[0]) - src_P (name[0]),
    "ra",       "angle",  src_P (ra) - src_P (name[0]),
    "dec",      "angle",  src_P (dec) - src_P (name[0]),
    "epoch",    "float",  src_P (epoch) - src_P (name[0]),
    "parallax", "float",  src_P (parallax) - src_P (name[0]),
    "date",     "date",   src_P (date) - src_P (name[0]),
    "time",     "time",   src_P (time) - src_P (name[0]),
    "d1ra",     "double", src_P (d1ra) - src_P (name[0]),
    "d1dec",    "double", src_P (d1dec) - src_P (name[0]),
    "d2ra",     "double", src_P (d2ra) - src_P (name[0]),
    "d2dec",    "double", src_P (d2dec) - src_P (name[0])
};

/* Stations Table Description */

struct stations stations_onerow;
#define stations_P(xx) (char *)(&stations_onerow.xx)
struct table_desc stations_desc[] = {
    "name",     "char",   stations_P (name[0]) - stations_P (name[0]),
    "x",        "double", stations_P (x) - stations_P (name[0]),
    "y",        "double", stations_P (y) - stations_P (name[0]),
    "z",        "double", stations_P (z) - stations_P (name[0]),
    "x_phs",    "double", stations_P (x_phs) - stations_P (name[0]),
    "y_phs",    "double", stations_P (y_phs) - stations_P (name[0]),
    "z_phs",    "double", stations_P (z_phs) - stations_P (name[0]),
    "axistype", "char",   stations_P (axistype[0]) - stations_P (name[0]),
    "axisoff",  "float",  stations_P (axisoff) - stations_P (name[0])
};


/* Tapes Table Description */

struct tapes    tapes_onerow;
#define tapes_P(xx) (char *)(&tapes_onerow.xx)
struct table_desc tapes_desc[] = {
    "name",   "char", tapes_P (name[0]) - tapes_P (name[0]),
    "tapeid", "char", tapes_P (tapeid[0]) - tapes_P (name[0]),
```

```
    "date",   "date", tapes_P (date) - tapes_P (name[0]),
    "start",  "time", tapes_P (start) - tapes_P (name[0]),
    "stop",   "time", tapes_P (stop) - tapes_P (name[0])
};

/* UT1 Table Description */

struct UT1      UT1_onerow;
#define UT1_P(xx) (char *)(&UT1_onerow.xx)
struct table_desc UT1_desc[] = {
    "date",   "date",   UT1_P (date) - UT1_P (date),
    "time",   "time",   UT1_P (time) - UT1_P (date),
    "ut1utc", "double", UT1_P (ut1utc) - UT1_P (date)
};
```

F JOB_DESC.H Job-Structure Header File

```
-----------------------------------------------------------------------
-r--r--r--  1 jbenson  vlb          4168 Aug  7 11:48 s.JOB_DESC.H
   in directory /home/ccc/vlb/fxcorr/src/code/SCCS
-----------------------------------------------------------------------

/*+ JOB_DESC.H
%% Initialization values for Job Table.

  A Job Table is created by the the Job_Loader task. The contents of
the table_com[] array are copied into the new Job Table. Each row
contains : the data table name, the number of rows in that table
(filled in by Job_Loader), the data table row size, the number of keys,
the data table base pointer and current row pointer, and the pointer
to the description table for the current data table.
---

LANGUAGE: C
ENVIRONMENT: vxWorks

++$ AUDIT TRAIL
 1.0 89/05/11 jbenson Initial submission.
 1.1 89/05/15 fxcorr date and time created 89/05/15 15:27:28 by fxcorr
 1.2 89/05/24 jbenson lined up columns by hand.
 1.3 89/07/13 jbenson modify constants table enrty for station a coords
 1.5 89/07/19 jbenson put bbconv back into channels table
 1.4 89/07/19 jbenson station-based correl, channels, obs, formatter tables
--$
-*/


/*  Job Table Description */

#define rowsize(yy) sizeof(struct yy)
struct job_table table_com[] ={
  "stations",     0, rowsize (stations),     9, 0, 0, &stations_desc[0],
  "UT1",          0, rowsize (UT1),          3, 0, 0, &UT1_desc[0],
  "polar",        0, rowsize (polar),        4, 0, 0, &polar_desc[0],
```

```
  "sources",      0, rowsize (sources),      11, 0, 0, &src_desc[0],
  "channels",     0, rowsize (channels),     11, 0, 0, &chan_desc[0],
  "formatter",    0, rowsize (formatter),    10, 0, 0, &formatter_desc[0],
  "observations", 0, rowsize (observations), 5, 0, 0, &obs_desc[0],
  "correl",       0, rowsize (correl),       11, 0, 0, &correl_desc[0],
  "tapes",        0, rowsize (tapes),         5, 0, 0, &tapes_desc[0],
  "clocks",       0, rowsize (clocks),        4, 0, 0, &clocks_desc[0],
  "constants",    0, rowsize (constants),    20, 0, 0, &constants_desc[0],
  "job",          0, rowsize (job),          11, 0, 0, &job_desc[0],
  "solar",        0, rowsize (solar),        19, 0, 0, &solar_desc[0]
};
```

G ACTIVE.H Active Base Table Header File

```
-----------------------------------------------------------------
-rw-rw-r--  1 jbenson  vlb          878 Aug 14 10:40 ACTIVE.H
-r--r--r--  1 jbenson  vlb         1332 Sep 19 10:46 s.ACTIVE.H
   in directory /home/ccc/vlb/fxcorr/src/code/SCCS
-----------------------------------------------------------------

/*+ ACTIVE.H
%%  Structure declaration for active_table, base table of active tables.

The active tables associated with a particular correlator job each have
a one row description in the active_base_table. As new active tables are
allocated and filled, their respective table base pointers and row counters
are updated in the active_base_table.
---
LANGUAGE: C (Include file)
ENVIRONMENT: vxWorks
++$ AUDIT TRAIL
 1.0  89/08/07 jbenson  Initial submission
--$
-*/

#define N_ACTIVE 100

struct active_base_table {
        char    table_name[16];     /* active table name */
        int     table_num;          /* index or version number that
                                       distinguishes multiple versions
                                       of the same table_name; first
                                       index number = 0  */
        int     num_rows;           /* number of row in current table */
        int     row_size;           /* row size (bytes) */
        int     num_elements;       /* number of elements per row */
        char    *base_ptr;          /* pointer to first row in table */
        char    *current_row;       /* pointer to current row */
};

#include "MODEL.H"
```

H MODEL.H Model Table Header File

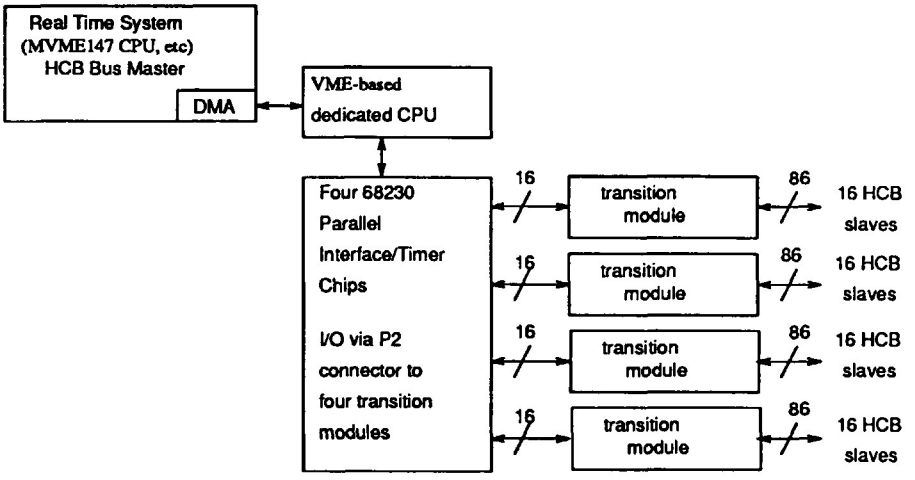
```
------------------------------------------------------------------
-r--r--r--  1 jbenson  vlb          5540 Sep 19 10:40 s.MODEL.H
   in directory /home/ccc/vlb/fxcorr/src/code/SCCS
------------------------------------------------------------------

/*+ MODEL.H
%% Structure declaration for CALC/polynomial model table.

The MODEL.H include file is the structure declaration of a table
created by the Model Task. The Model Table address is stored in the
Job Table. The CALC solutions and the polynomial coefficients from
the quintic spline fitter are held in the Model Table.
---
LANGUAGE: C (Include file)
ENVIRONMENT: vxWorks
++$ AUDIT TRAIL
 1.0 89/06/14 jbenson Initial submission
 1.1 89/06/15 jbenson date and time created 89/06/15 12:17:28 by jbenson
 1.2 89/06/15 jbenson shortened record, int to short etc.
 1.3 89/06/16 jbenson change struct name to model_table
 1.4 89/06/16 jbenson add semicolon after last close bracket
 1.5 89/08/28 jbenson add delay rate polynomials, 5 terms each.
 1.6 89/09/15 jbenson add tape_delay word, PBD offset (samples).
--$
-*/


/* MODEL TABLE STRUCTURE    */

struct model_table {
    int             date;             /* model epoch date (MJD) */
    double          time;             /* model epoch time (rad) */
    short           stn_id;           /* station id number */
    short           src_id;           /* source id number */
    short           crl_channel;      /* correlator channel number this
                                       * model */
    short           on_src_flag;      /* on = 1, off = 0 source flag */
```

```
    short           CALC_flag;      /* CALC flag : -1; CALC solution not
                                     * required, 0; CALC not yet
                                     * attempted, 1; CALC successful, >1;
                                     * CALC invalid solution, error no. */
    short           poly_flag;      /* spline fitter flag : -1; polynomial
                                     * solution not required, 0;
                                     * polynomial solution not yet
                                     * attempted, 1; polynomial solution
                                     * successful, >1; invalid solution,
                                     * error no.  */
    int             tape_delay;     /* delay to offset PBD (samples) */
    double          gdelay;         /* group delay (samples) */
    double          grate;          /* group delay rate (samples/second)  */
    double          pdelay;         /* phase delay (turns) */
    double          prate;          /* phase delay rate (turns/second) */
    double          gpoly[4];       /* group delay polynomial (samples)
                                     * coefficients 2 - 5 */
    double          ppoly[4];       /* phase delay polynomial (turns)
                                     * coefficients 2 - 5 */
    double          grpoly[5];      /* group delay rate polynomial */
    double          prpoly[5];      /* phase delay rate polynomial */
};
```

I Hardware Control Bus Interface

I.1 Overview

The VLBA Correlator Hardware Control Bus is the lower level portion of the communications path from the real time control VME system to the correlator hardware (see the "HDWR CTRL BUS INTF (HCB)" module in Figure 2). The complete path is not yet fully defined, but should be something of the following nature:



Each of the 68230 parallel interface chips provides 8 bi-directional data lines and 8 control lines to the corresponding transition module. The transition module contains the handshake hardware to interface to up to 16 slave targets in a correlator rack.

Each transition module provides the following signals to the corresponding rack:

Global to all slave interfaces:		
8	bi-directional data lines	HCB[0..7]
2	mode lines	MODE1, MODE0
1	strobe line	STROBE
Dedicated, one to each slave:		
16	select lines	SELECT
16	acknowledge/attention lines	ACK
43	total signals	

All signals between the transition module and the slave targets are differential RS-485 signals, resulting in 86 wires to the rack. The data lines have 100 ohm terminations at each end of the run (single resistor between the two wires).

The STROBE and MODE lines have terminations at the far end of the run which float low if there is no driver. The SELECT lines have a termination at the slave that floats low if there is no driver. The ACK lines have a termination at the transition module that floats high if there is no driver. (These terminations consist of a series of three resistors, 330 ohm + 150 ohm + 330 ohm, where one end goes to +5V, the other end goes to ground and the differential signal connects to each side of the 150 ohm.)

The HCB bus master may write to a single slave, or may "broadcast" to more than one slave at a time. The hardware provides the capability of allowing slaves to originate an attention request to the master.

Collision protection is provided in part by the hardware and in part by the slave software. A first byte state is provided to indicate that the next byte is the first byte of a transmission from the master to a slave. When a slave detects the first byte state, it must abort any pending communication process. In the hardware, the process of writing to a slave clears any pending hardware attention request from the slave.

When the master detects an attention request from a slave, the master responds by requesting a transmission from the slave.

I.2 Mode Definitions

The two bit mode word is decoded at each slave, along with the select line, to define the following modes:

SELECT	MODE	FUNCTION
not selected	0	allow slave to assert attention to master
not selected	1-3	idle
selected	0	master interrupts slave to flag first byte
selected	1	master writes bytes to slave
selected	2	master reads from slave
selected	3	master resets slave

The RESET function is a hard reset, applied to the slave microprocessor.

I.3 Slave Targets in a Rack

There are presently 8 slaves defined in each of the four correlator racks (see Figure 7). Using the notation HCBn to indicate a specific slave HCB interface port:

- HCB0 thru HCB5 are the six Playback Interfaces in the DPC chassis. The actual HCB interface goes to a 68000 processor on the Deformatter Card, which in turn is interfaced to the four 8751 processors on the two Track Recovery Cards.
- HCB6 goes to the FFT Control Card.
- HCB7 goes to the System Control Card which includes the Multiplier Control functions.
- The Long Term Accumulator may be located in one of the correlator racks. If so, it may also have a HCB interface.

If expansion is required beyond 16 slaves per rack, a second HCB will be provided for the rack.

I.4 Sub-Targets at Each Slave

Bytes per rack indicates the number of bytes that must actually be transferred to the slave, which may be less than the number of physical bytes in the target. For example, there are four sets of Xilinx chips in the PBI that receive identical personalities, so the data for only one set is sent. Update rates are shown if updates are more often than once per observation. Total bytes per rack for PBI, FFT Control, MULT Control and LTA (some byte counts rounded to nearest 1K; *these figures are very preliminary*):

PBI	FFT	MULT	LTA	Rack totals (bytes)	
477K	683K	6K	?	1160K	at start observation
0	31K	0	?	31K	at "minutes" update rate
6K	0	0	?	6K	at tape reversals
111K	82K	2K	?	195K	at tape change

The following sections provide details on the sub-targets in each rack.

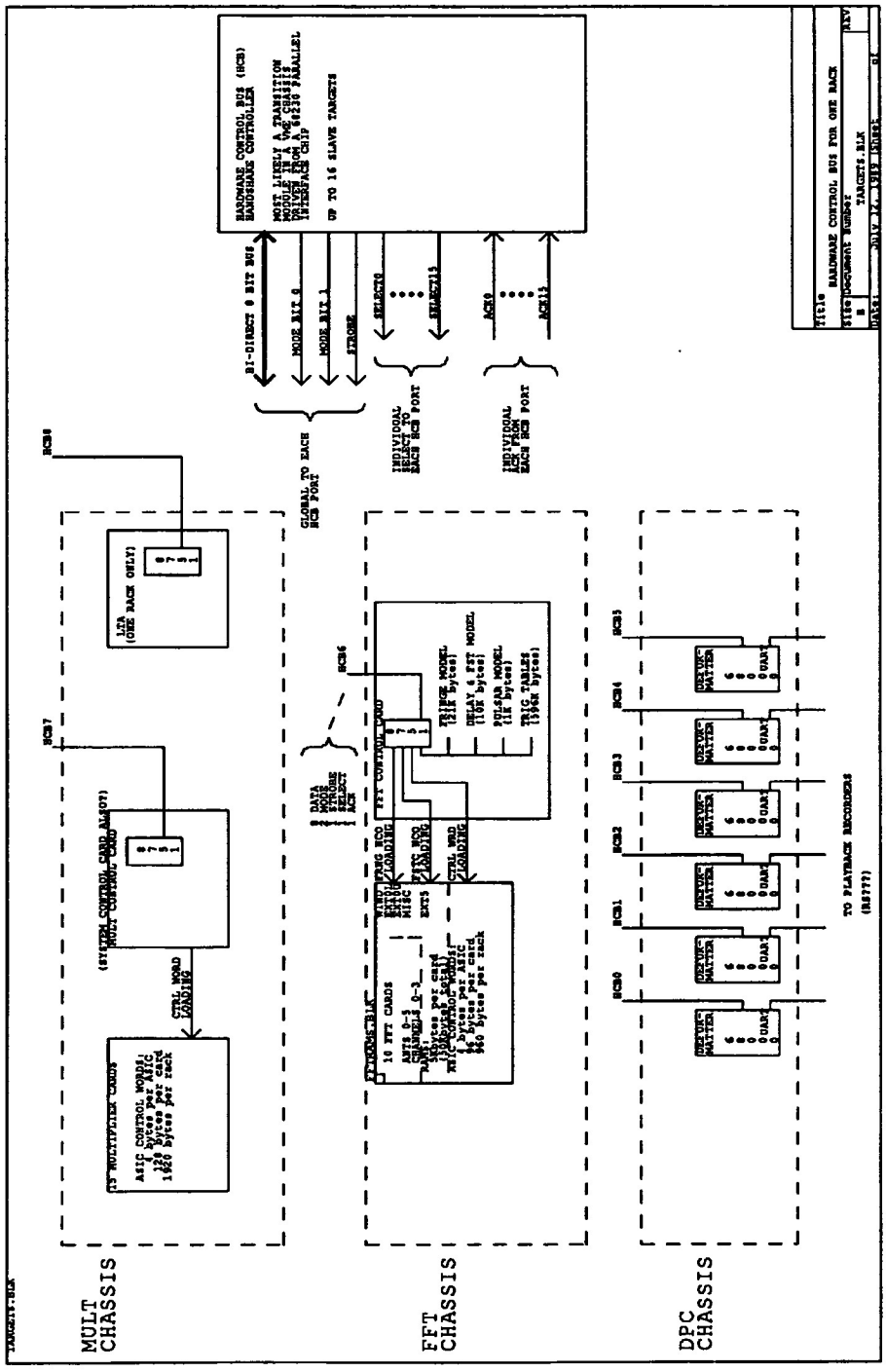


Figure 7: Hardware Control Bus for One Rack

I.4.1 Playback Interface (PBI)

There are 6 PBIs per rack, each with 2 Track Recovery Cards, 1 Deformatter Card. The HCB interface is connected to a 68000 μP on the Deformatter.

sub-target	bytes/ PBI	at obs start	bytes/update/rack	
			rack	period
Deformatter, 68K code	10K	60K		
Deformatter, barrel-roll & data invalid; set of 2 Xilinx XC3020 (1.9K per chip)	4K	24K		
Deformatter: Bit Ordering; set of 4 Xilinx XC3030(?) (2.8K per chip)	11K	66K	66K	tape changes
Deformatter: FFT sequencer RAM	4K	24K	24K	tape changes
Deformatter: R/W sequencer RAM	2K	12K	12K	tape changes
Deformatter: Address sequencer RAM	512	3K	3K	tape changes
Track Recovery: 8751 code (4 total per ant, 4K each)	16K	96K		
Track Recovery: set of 11 Xilinx XC3030 for 9 tracks (2.8K per chip)	31K	186K		
PBI misc: active tracks, tape modes, ?????	1K	6K	6K	tape changes & reversals
Rack sub total:		477K	6K 111k	reversals tape changes

I.4.2 FFT Control Card

The Parameter storage RAM is presently 8K×32; it may be increased to 32K×32.

Sub-Target	Bytes/rack at obs start	Periodic bytes/rack	Update period
8751 code	4K		
Fringe model	21K	21K	minutes
Delay model	5K	5K	minutes
FSEC model	5K	5K	minutes
Pulsar model	1K		
Trig tables	596K		
RAMs on FFT cards: window, EXT0UPR, EXT0LWR, EXT5, misc	50K	50K	tape change
FFT ASIC control words	1K	1K	tape change
Rack sub total:	683K	31K 82K	minute level tape change

Fringe: 10 stations × 4 channels × 8 coef = 320 coef
(320 phase coef + 320 rate coef) × 8 bytes = 5120 bytes
5120 bytes per model × 4 models = 20,480 bytes total

Delay: 10 stations × 2 phase centers × 8 coef = 160 coef
160 coef × 8 bytes = 1280 bytes
1280 bytes per model × 4 models = 5120 bytes total

FST: 10 stations × 2 phase centers × 8 coef = 160 coef
160 coef × 8 bytes = 1280 bytes
1280 bytes per model × 4 models = 5120 bytes total

Pulsar: 1 pulsar × 4 channels × 4 coef = 16 coef
16 coef × 8 bytes = 128 bytes
128 bytes per model × 4 models = 512 bytes total

I.4.3 Mult Control Card (system control card?)

Sub-Target	Bytes/rack at obs start	Periodic bytes/rack	Update Period
8751 code	4K		
Mult ASIC control words	2K	2K	tape change
Rack sub total:	6K	2K	tape change

I.4.4 Long Term Accumulator (LTA)

To be defined.

I.5 HCB Slave Interface Hardware Description

To be inserted later. Until then, consult the schematic in Figure 8.

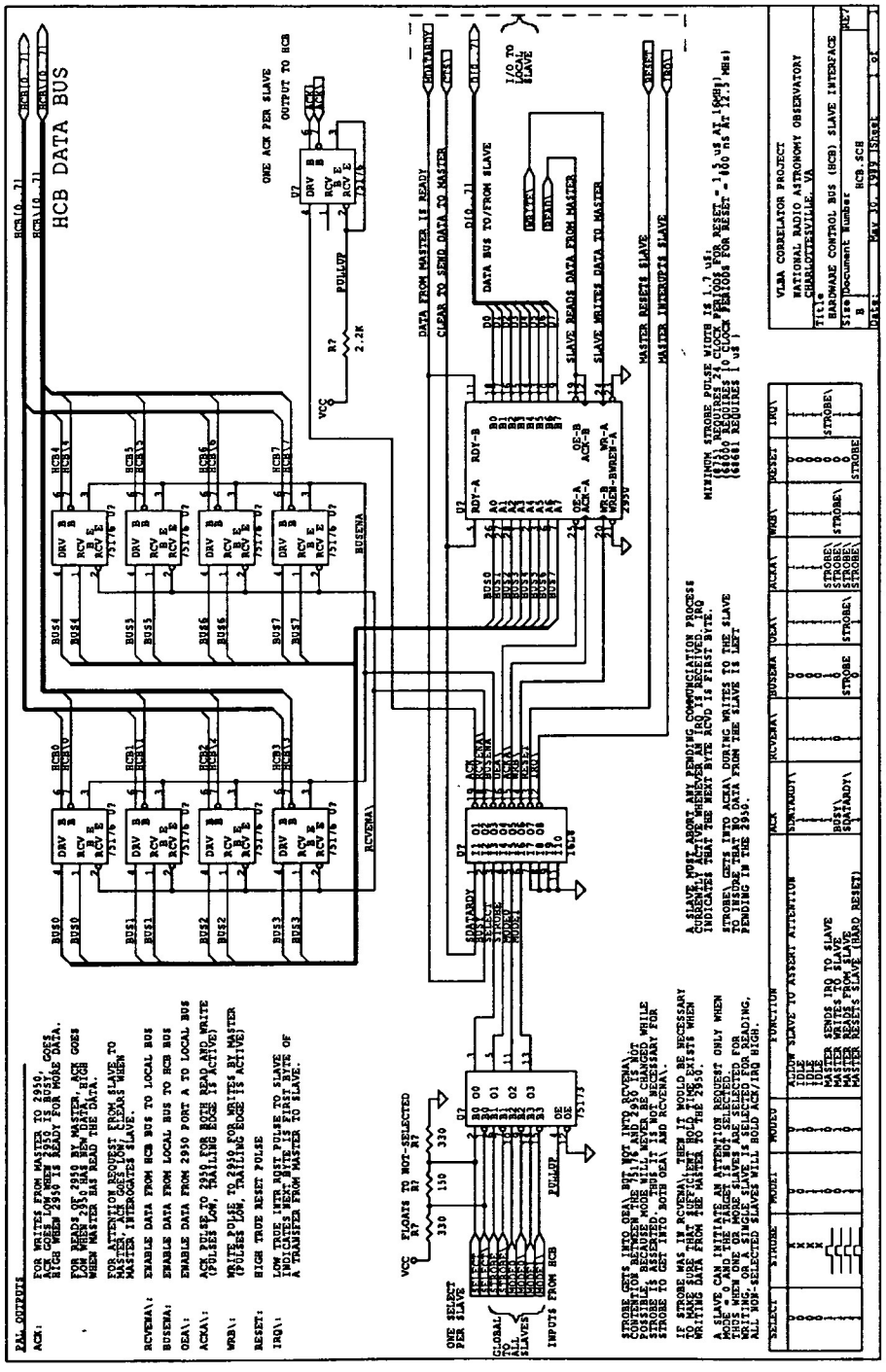


Figure 8: Hardware Control Bus Slave Interface

J Unresolved Issues & TBDs

1. "Single-dish" (auto-correlation) jobs (one-station arrays!). Can AIPS accept auto-correlation spectra?
2. How will we specify things like track-substitution in track recovery logic? (Equivalent question is: how will we be told to do it by ACC?) How will we know about barrel-roll active/inactive?
3. Handling of bad-data flags.
4. Transfer of Schedules/Logs from non-VLBA stations.
5. Worldwide tape bar-code system.
6. Software support for pulsar modes.
7. LTA-output filter.

K Technology Used to Prepare This Document

The correlator project uses Sun workstations under SunOS 4.0.x, with a Sun-3/280 file server and five Sun-3/50 and 3/60 clients. The 3/280 has 16 MB of RAM and one 892 MB disk. The 3/50 clients each have 8 MB of RAM (4 MB original plus 4 MB expansion kits from Solflower Computer).

This document has been typeset using LaTeX version 2.09 executing under TeX version 2.9. The document is in LaTeX's "article" style with 11 pt type. The various sections, figures and appendices were integrated into this typeset document by use of a "Makefile" processed by the basic Unix utility `make`. It is hard to exaggerate the usefulness of `make`. The text was edited by several different people, using several different editors. Two of the authors use Richard Stallman's "GNU" Emacs (distributed by the Free Software Foundation) with `emacstool` under the Sunview-1 window system; another uses Sun's `textedit` window- and mouse-based text editor. The document was examined in the workstation windows while editing by using the `dvitool` previewer from the "Vortex" project at UC-Berkeley. Output was on a QMS PS-810 300 dpi laser printer operating in its (default) Postscript mode using the `dvi2ps` utility.

Figures 1, 3, 4, 6 and the drawing on p. 82 were "edited" in the Sun windows using the "Fig" graphics editor, version Fig 1.4.FS. The "Transfig" package was used to incorporate these figures into the LaTeX document automatically. Figures 2, 5, 7 and 8 were prepared on a PC using ORCAD with output in Postscript format; the Postscript files were inserted (scaled down) into the document with the `\special{}` command of `dvi2ps`.

Appendices B, C, D, E, F, G and H are maintained using the SCCS (Source Code Control System) utility of Berkeley Unix. The extraction of the current SCCS versions and the reformatting of the text for typesetting were performed by a shell script which utilized the AWK language[15]. AWK has proven to be quite useful for a number of utility operations, especially in its `gawk` version from the Free Software Foundation.

These are *public domain* tools; modest shipping and handling charges were paid for the original Unix TeX tape from the Univ. of Washington and the Vortex tape from UC-Berkeley. The other tools were fetched from various anonymous-guest FTP (File Transfer Protocol) archives across the Internet *free of charge.* In fact Unix-TeX is now freely available via FTP.

[15] see "The AWK Programming Language", A. V. Aho, B. W. Kernighan, P. J. Weinberger, Addison-Wesley, 1988.