

Channel Archiver Manual

EPICS



March 3, 2006,
for R3.14.4 and higher

Involvements

Bob Dalesio designed the original index file, data file layout, and implemented the first prototype.

From then on, the following people have been involved at one time or another:

Thomas Birke,
Sergei Chevtsov,
Kay-Uwe Kasemir,
Chris Larrieu,
Craig McChesney,
Peregrine McGehee,
Nick Pattengale.

No Warranty

Although the programs and procedures described in this manual are meant to be helpful instruments for archiving, maintaining and retrieving control system data, there is no warranty, either expressed or implied, including, but not limited to, fitness for a particular purpose. The entire risk as to the quality and performance of the programs and procedures is with you. Should the programs or procedures prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event will anybody, including the persons listed above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the programs (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the programs to operate with any other programs).

Contents

1 Overview	1
2 Background	2
2.1 What is a Channel?	2
2.2 Data Sources	3
2.3 Sampling Options	4
2.4 Time Stamps	5
2.5 Sensible Sampling	5
2.6 Times: EPICS, Local, Greenwich, Daylight Saving	7
2.7 Time Stamp Correlation	9
2.7.1 “Raw” Data	9
2.7.2 “Before or at” Interpretation of Start Times	9
2.7.3 Spreadsheet Generation	10
2.7.4 Averaging, Linear Interpolation	11
2.7.5 Plot-Binning	13
3 ArchiveEngine	15
3.1 Configuration	16
3.1.1 write_period	18
3.1.2 get_threshold	18
3.1.3 file_size	18
3.1.4 ignored_future	18
3.1.5 buffer_reserve	18
3.1.6 max_repeat_count	19
3.1.7 disconnect	19
3.1.8 group	19
3.1.9 channel	20
3.2 Starting and Stopping	22
3.2.1 Starting	22
3.2.2 “-log” Option	22
3.2.3 “-description” Option	22
3.2.4 “-port” Option	22
3.2.5 “-nocfg” Option	23
3.2.6 The “archive_active.lck” File	23

3.2.7	More than one ArchiveEngine	23
3.2.8	Stopping	23
3.3	Web Interface	24
3.4	Threads	25
4	Data Retrieval	27
4.1	ArchiveExport	28
4.2	Java Archive Client	29
4.3	Data Server	31
4.3.1	Installation	32
4.3.2	Configuration	34
4.3.3	Testing, Debugging	34
4.3.4	Standalone Data Server	35
4.3.5	Running ArchiveDataServerStandalone	36
4.4	XML-RPC Protocol	37
4.4.1	archiver.info	37
4.4.2	archiver.archives	39
4.4.3	archiver.names	39
4.4.4	archiver.values	40
4.4.5	Note about Tiny Numbers and Precision	42
4.5	Perl Client	42
4.6	StripTool	44
4.7	Matlab	45
5	Data Storage	50
5.1	Index Files	50
5.1.1	Implementation Details	52
5.2	Data Files	53
5.2.1	Implementation Details	54
5.3	Index Tool	55
5.3.1	make_indexconfig.pl	55
5.3.2	Internals	56
5.4	Data Tool	58
5.5	Statistics	58
5.5.1	Write Performance	59
5.5.2	Index Performance	59
5.5.3	Impact of Data Management on Performance	61
5.5.4	New Index compared to old “multi archive”	62
5.5.5	Retrieval Performance	62
6	Example Setup	64
6.1	archiveconfig.xml	64
6.2	ArchiveDaemon	65
6.2.1	Status Information	65
6.3	Indices	65
6.4	Data Server	65

6.5	Current Archive Status	66
6.6	Directory Layout	66
6.7	Sub-Archives	67
6.7.1	ArchiveDaemon Details	68
6.8	Common Tasks	69
6.8.1	Check Daemon, Engine, Connected Channels,	69
6.8.2	Modify Engine's Request Files	69
6.8.3	Add Engine or Daemon	69
6.8.4	An Engine isn't running	70
6.8.5	I want to stop a Daemon	70
6.8.6	A Daemon isn't running	70
6.8.7	Re-building a Master Index	70
6.9	Data Management	70
7	ArchiveDaemon	73
7.0.1	Configuration	74
7.0.2	Starting and Running	76
7.0.3	Daemon's Web Pages	78
7.0.4	Disabling Engines	79
7.0.5	Stopping and More	79
8	Setup, Installation	80
8.1	Compilation	80
8.1.1	XML-RPC	81
8.1.2	Xerces XML Library	83
8.1.3	Expat	84
8.1.4	XML-Simple	84
8.1.5	Frontier	84
8.2	Installation	85
8.2.1	DTD Files	85
9	Common Errors and Questions	87
9.1	Why is there no data in my archive?	87
9.2	Why do I get #N/A, why are there missing values in my spread-sheet?	87
9.3	Why do I not get what I think I should get from the network data server?	88
9.4	Back in time?	88
9.5	Found an existing lock file 'archive_active.lck'	90
9.6	Crashes of the ArchiveEngine,	90
9.7	'ChannelName': Cannot add event because data type is unknown	91
9.8	Cannot create a new data file within file size limit	91
9.9	Found an existing 'indextool_active.lck' lock file	91

10 Legacy	92
10.1 Directory Files	92
10.2 ArchiveManager	92
10.3 Archive Engine ASCII Configurations	93
10.4 CManager, CAbgManager	94
11 Changes	95

Chapter 1

Overview

The Channel Archiver is an archiving toolset for the Experimental Physics and Industrial Control System, EPICS [1]. It can archive any value that is available via ChannelAccess (CA), the EPICS network protocol [2]. We use the term “archiver” whenever we refer to the collection of programs which allow us to take samples, place them into some storage and retrieve them again. The archiver toolset roughly splits into the following pieces:

Sampling: The ArchiveEngine collects data from a given list of ChannelAccess Channels. The details of when a sample is taken etc. can be configured: One may store every change, store changes that exceed a dead-band (that is configured on the CA server) or use periodic scanning. The configuration and operation of the ArchiveEngines will obviously require some planning, as only data that was sampled and stored will be available for future retrieval and analysis. Some sensible compromise will have to be made between the urge to store all minuscule changes of all the available channels of a site on one hand and data storage constraints on the other.

Storage: The data is stored in binary index and data files. Most end users need not be concerned about the internals of those files, not even where they are located, because additional indices allow several sub-archives to appear like one, bigger, combined archive. Somebody at each site, though, will need to perform maintenance tasks: Decide where the data sets are located, how they are backed up and how users can access them.

Retrieval: The archiver toolset provides generic retrieval tools for browsing the available channels and values, including simple multi-channel comparisons. An API allows users to write more sophisticated data analysis tools.

Chapter 2

Background

2.1 What is a Channel?

The Channel Archiver deals with Channels that are served by EPICS ChannelAccess. It stores all the information available via ChannelAccess:

- Time Stamp
- Status/Severity
- Value
- Meta information:
Units, Limits, ... for numeric channels, enumeration strings for enumerated channels.

The archiver stores the original time stamps as it receives them from ChannelAccess. It cannot check if these time stamps are valid, except that it refuses to go “back in time” because it can only append new values to the end of the data storage. It is therefore imperative to properly configure the data sources, that is: the clocks on the CA servers. For more details on the EPICS time stamps refer to section 2.6.

NOTE: If the CA server provides bad time stamps, for example stamps that are older than values which are already in the archive, or stamps that are unbelievably far ahead in the future, the ArchiveEngine will log a warning message and refuse to store the affected samples. This is a common reason for “Why is there no data in my archive?”. (There is one more, hard to resolve reason for back-in-time warnings, see page 88).

As for the values themselves, the native data type of the channel as reported by ChannelAccess is stored. For those familiar with the ChannelAccess API, this means: Channels that report a native data type of DBR_XXX_ are stored as DBR_TIME_XXX after once requesting the full DBR_CTRL_XXX information. The Archiver can therefore handle scalar and array numerics (double, int, ...), strings and enumerated types.

2.2 Data Sources

Before even considering the available sampling options, it is important to understand the data sources, the ChannelAccess servers whose channels we intend to archive. In most cases we will archive channels served by an EPICS Input/Output Controller (IOC) which is configured via a collection of EPICS records. Alternatively, we can archive channels served by a custom-designed CA server that utilizes the portable CA library PCAS. In those cases, one will have to contact the implementor of the custom CA server for details. In the following, we concentrate on the IOC scenario and use the analog input record from listing 2.1 as an example.

```
record(ai, "aiExample")
{
    field(SCAN, ".1_second")
    field(ADEL, "0.1")
    field(EGU, "Volts")
    field(PREC, "2")
    field(HOPR, "4095")
    field(LOPR, "0")
    field(HIHI, "10")
    field(HIGH, "9")
    field(LOW, "1")
    field(LOLO, "0")
    field(HHSV, "MAJOR")
    field(HSV, "MINOR")
}
```

Listing 2.1: “aiExample” record

What happens when we try to archive the channel “aiExample”? We will receive updates for the record’s value field (VAL). In fact we might as well have configured the archiver to use “aiExample.VAL” with exactly the same result. The record is scanned at 10 Hz, so we can expect 10 values per second. Almost: The archive deadband (ADEL) limits the values that we receive via CA to changes beyond 0.1. When archiving this channel, we could store at most 10 values per second or try to capture every change, utilizing the ADEL configuration to limit the network traffic.

NOTE: The archiver has no knowledge of the scan rate nor the deadband configuration of your data source! You have to consult the IOC database or PCAS-based code to obtain these.

With each value, the archiver stores the time stamp as well as the status and severity. For aiExample, we configured a high limit of 10 with a MAJOR severity. Consequently we will see a status/severity of HIHI/MAJOR whenever the VAL field reaches the HIHI limit. In addition to the value (VAL field), the archiver

also stores certain pieces of meta information. For numeric channels, it will store the engineering units, suggested display precision, as well as limits for display, control, warnings, and alarms. For enumerated channels, it stores the enumeration strings. Applied to the aiExample record, the suggested display precision is read from the PREC field, the limits are derived from HOPR, LOPR, HIHI, ..., LOLO.

NOTE: You will have to consult the record reference manual or even record source code to obtain the relations between record fields and channel properties. The analog input record's EGU field for example provides the engineering units for the VAL field. We could, however, also try to archive aiExample.SCAN, that is the SCAN field of the same record. That channel aiExample.SCAN will be an *enumerated* type with possible values "Passive", ".1 second" and so on. The EGU field of the record no longer applies! Another example worth considering: While HOPR defines the upper control limit for the VAL field, what is the upper control limit if we archive the HOPR field itself?

It is also important to remember that the archiver — just like any other ChannelAccess client — does **not** know anything about the underlying EPICS record type of a channel. In fact the channel might not be based on any record at all if we use a PCAS-based server. Given the name of an analog input record, it will store the record's value, units and limits, that is: most of the essential record information. Given the name of a stepper motor record, the archiver will also store the record's value (motor position) with the units and limits of the motor position. It will not store the acceleration, maximum speed or other details that you might consider essential parts of the record. To archive those, one would have to archive them as individual channels.

2.3 Sampling Options

The ArchiveEngine supports these sampling mechanisms:

Monitor: In this mode, the ArchiveEngine requests a CA monitor, i.e. it subscribes to changes and we store all the values that the server sends out. The CA server configuration determines when values are sent.

Sampled: In this mode, the ArchiveEngine periodically requests a value from the CA server, e.g. every 30 seconds.

Sampled using monitors: This mode is very similar to the previous one: The ArchiveEngine is again configured to store periodic samples, e.g. one sample every 5 seconds. But instead of actively requesting a value from the CA server at this rate, it establishes a monitor and only saves a value every 5 seconds.

The configuration of the engine in section 3.1 describes how one selects the sampling mechanism for each channel. When selecting monitored operation, you will need to provide an estimate of how many monitors the channel emits,

so that the engine can allocate appropriate buffer space (more on this in sections 3.1.5 and 3.1.9).

The difference between the two sampled modes is subtle but important for performance reasons. Assume our data source changes at 1 Hz. If we want to store a value every 30 seconds, it is most efficient to send a 'read'-request every 30 seconds. If, on the other hand, we want to store a value every 5 seconds, it is usually more effective to establish a monitor, so we automatically receive updates about every second, and simply *ignore* 4 of the 5 values.

When configuring a channel, the user only selects either "Monitor" or "Scan" with a sampling rate. The ArchiveEngine will automatically determine which mechanism to use for sampled operation, periodic reads or monitors (see the *get.threshold* configuration parameter, section 3.1.2, for details).

NOTE: The values dumped into the data storage will not offer much indication of the sampling method. In the end, we only see values with time stamps. If for example the time stamps of the stored values change every 20 seconds, this could be the result of a monitored channel that happened to change every 20 seconds. We could also face a channel that changed at 10 Hz but was only sampled every 20 seconds.

2.4 Time Stamps

Each ChannelAccess Server provides time-stamped data. An IOC for example stamps each value when the corresponding record is processed. These time-stamps offer nano-second granularity. Most applications will not require the full accuracy, but some hardware-triggered acquisition, utilizing interrupts on a fast CPU, might in fact put the full time stamp resolution to good use.

The ChannelArchiver as a generic tool does not know about the origin of the time stamps, but it tries to conserve them. Fig. 2.1 shows the same channel, archived with different methods. When using the "Monitor" method for archiving, we capture all the changes of the channel, resulting in the data points marked by black diamonds. When we use scanned operation, e.g. every 30 seconds, the following happens: About every 30 seconds, the ArchiveEngine stores the current value of the channel *with its original time stamp!*. So while the ArchiveEngine might take a sample at

14:53:30, 14:54:00, 14:54:30, 14:55:00, ...,

it stores the time stamps that come with the values, and in the example from Fig. 2.1 those happened to be

14:53:29.091, 14:53:59.092, 14:54:29.094, 14:54:59.095, ...

2.5 Sensible Sampling

The data source configuration and sampling need to be coordinated. In fact the whole system needs to be understood. When we deal with water tank temper-

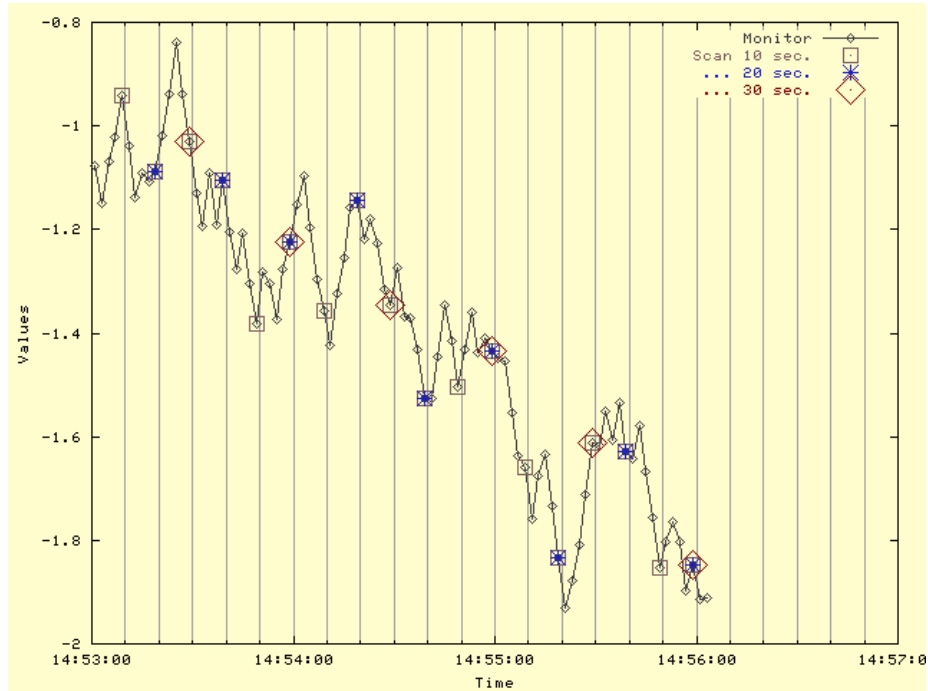


Figure 2.1: Time Stamps and Sampling

atures as one example, we have to understand that the temperature is unlikely to change rapidly. Let us assume that it only varies within 30...60 seconds. The analog input record that reads the temperature could be configured to scan every 2 seconds. Not because we expect the temperature to change that quickly but mostly to provide the operator with a warm and fuzzy feeling that we are still reading the temperature: The operator display will show minuscule variations in temperature every 2 seconds. An ArchiveEngine that is meant to capture the long-term trend of the tank temperature could then sample the value every 60 seconds.

On the other extreme could be channels for vacuum readings along linac cavities. The records that read them might be configured to scan as fast as the sensing devices permit, maybe beyond 10 Hz, so that interlocks on the IOC run as fast as possible. Their deadbands (ADEL and MDEL) on the other hand are configured to limit the data rate that is sent to monitoring CA clients: Only meaningful vacuum changes are sent out, significantly reducing the amount of data sent onto the network. The ArchiveEngine can then be configured to monitor the channel: During normal operation, when the vacuum is fairly stable, it will only receive a few values, but whenever the vacuum changes because of a leak, it will receive a detailed picture of the event.

Another example is a short-term archive that is meant to store beam po-

sition monitor (BPM) readings for every beam pulse. The records on the IOC can then be configured with `ADEL=-1` and the ArchiveEngine to use monitors, resulting in a value being sent onto the network and stored in the archive even if the values did not change. The point here is to store the time stamps and beam positions for each beam pulse for later correlation. Needless to say that this can result in a lot of data if the engine is kept running unattended. The preferred mode of operation would be to run the engine only for the duration of a short experiment.

NOTE: The scanning of the data source and the ArchiveEngine run in parallel, they are not synchronized. Example: If you have a record scanned every second and want to capture every change in value, configuring the ArchiveEngine to scan every second is **not** advisable: Though both the record and the ArchiveEngine would scan every second, the two scans are not synchronized and rather unpredictable things can happen. Instead, the "Monitor" option for the ArchiveEngine should be used for this case.

2.6 Times: EPICS, Local, Greenwich, Daylight Saving

The EPICS base software that is used by the IOCs and also the archiver deals with time as seconds and nanoseconds since January 1, 1990. This "EPICS Time" is using Greenwich Mean Time (GMT), also known as Universal Time Coordinates (UTC). So the EPICS Time stamp 0 stands for 01/01/1990, 00:00:00 UTC.

People living in Germany are typically in a time zone one hour east of UTC. For them, the EPICS Time stamp 0 translates into January 1, 1990, at 01:00:00 in the morning. This is one example of "Local Time". Anybody living in the United States is of course familiar with time zone conversions ever since you tried to match what's in the TV Guide with what's actually on TV.

The EPICS base software includes routines for converting EPICS Time into Local Time and vice versa. Before EPICS R3.14, these routines used an environment variable `EPICS_TS_MIN_WEST` which needed to be set to the minutes west of UTC. "-60" for Germany in the above example. Since R3.14, this environment variable is no longer used. The time stamp conversion code in EPICS base now relies on the operating system and the C/C++ runtime library to handle any time zone issues.

It is important to remember that the data served by CA Servers is in EPICS time, that is based on UTC and *not* your local time. The ArchiveEngine stores that data as received, which is again in EPICS time based on UTC. When the network data server is asked for samples, those are also based UTC, albeit with a slight shift from a 1990 epoch to 1970, simply because this is more convenient to use in most programming languages. C, C++, Java and perl all include routines for converting 1970-epoch seconds to local time and back.

Time stamps are only converted to local time when they are displayed or

entered. The ArchiveExport program will provide you with e.g. a spreadsheet that has a “Time” column in local time. The Java Archive Client will plot the data with a time axis in local time. Whenever you specify start and end times for a data request, this is done in local time.

An example of possible consequences: Assume you live in San Francisco, California (UTC+8), and you receive a CD-ROM with archived data from the SNS in Oak Ridge, Tennessee (UTC+5). If you want to investigate what happened at noon on 01/01/2004 at the SNS, you will have to query for 09:00:00 to adjust for the different time zones.

The EPICS base code also relies on the operating system services for daylight saving time (DST). At least under RedHat Linux 9 this seems to work fine when you are inside the United States. Example: In the US, daylight saving time went into effect on 04/04/2004, 02:00:00, and I archived a “stringin” record which had device support that converted the time stamp of the record into a string, including daylight savings information. The result looks like this when the data is exported again on the next day, that is at a time where DST is in effect:

EPICS Seconds	Time	Value
449917196	01:59:56	04/04/2004 01:59:56.805984000
449917197	01:59:57	04/04/2004 01:59:57.816036000
449917198	01:59:58	04/04/2004 01:59:58.826102000
449917199	01:59:59	04/04/2004 01:59:59.836110000
449917200	03:00:00	04/04/2004 03:00:00.846121000 (DST)
449917201	03:00:01	04/04/2004 03:00:01.856196000 (DST)
449917202	03:00:02	04/04/2004 03:00:02.867379000 (DST)
449917203	03:00:03	04/04/2004 03:00:03.876315000 (DST)
449917204	03:00:04	04/04/2004 03:00:04.886356000 (DST)

The seconds of the raw EPICS time stamp simply continue to count up though the transition because UTC is not affected by DST. (This test was run in the US Mountain time zone, UTC+7, and the nanoseconds of the EPICS time stamp are omitted.) The Value of the string record, which contains the local time as the IOC saw it, jumps from what would have been 02:00:00 to 03:00:00 DST. When the time stamp is printed (at a later time when DST applies), the “Time” column matches the times in the value string.

After changing the computer’s clock to times in January 2004 and 2005, that is to times outside of DST before and after the data in the archive, the result is the exact same: The EPICS time stamp routines use the DST settings that apply for a given time stamp, not for the current time.

Countries differ in their algorithms for switching to DST, and if your operating system does not follow the rules in your location, you might see sudden offsets in time between the wall clock and the archived data.

2.7 Time Stamp Correlation

We have stressed more than once that the Channel Archiver preserves the original time stamps as sent by the CA servers. This commonly leads to difficulties when comparing values from different channels. The following subsections investigate the issue in more detail and show several ways of manipulating the data in order to allow data reduction and cross-channel comparisons. In short, the options described in the following subsections are:

Raw Data: Provides every archived sample “as is”.

Spreadsheet: Staircase interpolation/filling to form a spreadsheet.

Averaging, Linear Interpolation: Maps the raw data onto specific time stamps.

Plot Binning: Reduces the number of samples for plotting.

2.7.1 “Raw” Data

Even when two channels were served by the same IOC, and originating from records on the same scan rate, their time stamps will slightly differ because a single CPU cannot scan several channels at exactly the same time. Tab. 2.1 shows one example.

Time	A	Time	B
17:02:28.700986000	0.0718241	17:02:28.701046000	-0.086006
17:02:37.400964000	0.0543581	17:02:37.510961000	-0.111776
...		...	

Table 2.1: Example Time Stamps for two Channels A and B.

When we try to export this data in what we call raw spreadsheet format, a problem arises: Even though the two channels' time stamps are close, they do not match, resulting in a spreadsheet as shown in Tab. 2.2. Whenever one channel has a value, the other channel has none and vice versa. This spreadsheet does not yield itself to further analysis; calculations like $A - B$ will always yield '#N/A' since either A or B is undefined.

2.7.2 “Before or at” Interpretation of Start Times

When you invoke a retrieval tool with a certain start time, the archive will rarely contain a sample for that exact start time. As an example, you might ask for a start time of “07:00:00” on some date. Not because you expect to find a sample with that exact time stamp, but because you want to look at data from the beginning of that day's operations shift, which nominally began at 7am.

Time	A	B
3/22/2000 17:02:28.700986000	0.0718241	—
3/22/2000 17:02:28.701046000	—	-0.086006
3/22/2000 17:02:37.400964000	0.0543581	—
3/22/2000 17:02:37.510961000	—	-0.111776
...		

Table 2.2: Spreadsheet for raw Channels A and B.

The software underlying all retrieval tools anticipates this scenario by interpreting all start times as “before or at”. Given a start time of “07:00:00”, it returns the last sample before that start time, unless an exact match is found. So in case a sample for the exact start time exists, it will of course be returned. But if the archive contains no such sample, the previous sample is returned.

Applied to Tab. 2.2, channel A, you would get the samples shown in there not only if you asked for “17:02:28.700986000”, the exact start time, but also if you asked for “17:02:30”. It is left to the end user to decide whether that previous sample is still useful at the requested start time, if it’s “close enough”, or if it needs to be ignored.

2.7.3 Spreadsheet Generation

There are several ways of mapping channels onto matching time stamps. One is what we call Staircase Interpolation or Filling: Whenever there is no current value for a channel, we re-use the previous value. This is often perfectly acceptable because the CA server will only send updates whenever a channel changes beyond the configured deadband. So if we monitored a channel and did not receive a new value, this means that the previous value is still valid — at least within the configured deadband. In the case of scanned channels we have no idea how a channel behaved in between scans, but if we e.g. look at water temperatures, it might be safe to assume that the previous value is still “close enough”. Table 2.3 shows the previously discussed data subjected to staircase interpolation. Note that in this example there is no initial value for channel B, resulting in one empty spreadsheet cell. From then on, however, there are always values for both channels, because any missing samples are filled by repeating the previous one. Because of the interpretation of start times explained in section 2.7.2, you would get the result in Tab. 2.3 not only if you asked for values beginning “3/22/2000 17:02:28.700986000”, but also when you asked for e.g. “17:02:30”: Since neither channel A nor B have a sample for that exact time stamp, the retrieval library would select the preceding sample for each channel, resulting in the output shown in Tab. 2.3.

NOTE: While table 2.3 marks the filled values by printing them in italics, spreadsheets generated by archive retrieval tools will not accent the filled values in any way, so care must be taken: Those filled values carry artificial time stamps. If

you depend on the original time stamps in order to synchronize certain events, you must not use any form of interpolation but always retrieve the raw data.

Time	A	B
3/22/2000 17:02:28.700986000	0.0718241	—
3/22/2000 17:02:28.701046000	<i>0.0718241</i>	-0.086006
3/22/2000 17:02:37.400964000	0.0543581	<i>-0.086006</i>
3/22/2000 17:02:37.510961000	<i>0.0543581</i>	-0.111776
...		

Table 2.3: Spreadsheet for Channels A and B with Staircase Interpolation; “filled” values shown in italics.

You did of course notice that the staircase interpolation does not reduce the amount of data. Quite the opposite: In the above examples, channels A and B each had 2 values. With staircase interpolation, we don’t get a spreadsheet with 2 lines of data but 4 lines of data. The main advantage of filling lies in the preservation of original time stamps.

2.7.4 Averaging, Linear Interpolation

Both averaging and linear interpolation generate artificial values from the raw data. This can be used to reduce the amount of data: For a summary of the last day, it might be sufficient to look at one value every 30 minutes, even though the archive could contain much more data. Another aspect is partly cosmetic and partly a matter of convenience: When we look at Tab. 2.3, we find rather odd looking time stamps. While these reflect the real time stamps that the ArchiveEngine received from the ChannelAccess server, it is often preferable to deal with data that has time stamps which are nicely aligned, for example every 10 seconds: 11:20:00, 11:20:10, 11:20:20, 11:20:30 and so on. To accomplish this, the data is binned. For example, the time span of one day, 24 hours, can be divided into 2880 sections, each of which covers 30 seconds. Each of those sections is called a “Bin”. The raw samples for the day are then investigated as follows:

- When we select Averaging, the average over all the samples that fall into a bin is returned. The center of the bin is used as a time stamp.
- When we select Linear Interpolation, the value of the channel at the border of each bin is determined via linear interpolation between the last sample before and the first sample after the border of the bin. The border of each bin determines the time stamp.

Fig. 2.2 compares the result of retrieving the raw data with averaging and linear interpolation over 10-second-bins. Averages are determined for the center of each bin, i.e. 08:48:35, 08:48:45, ..., while linear interpolation generates

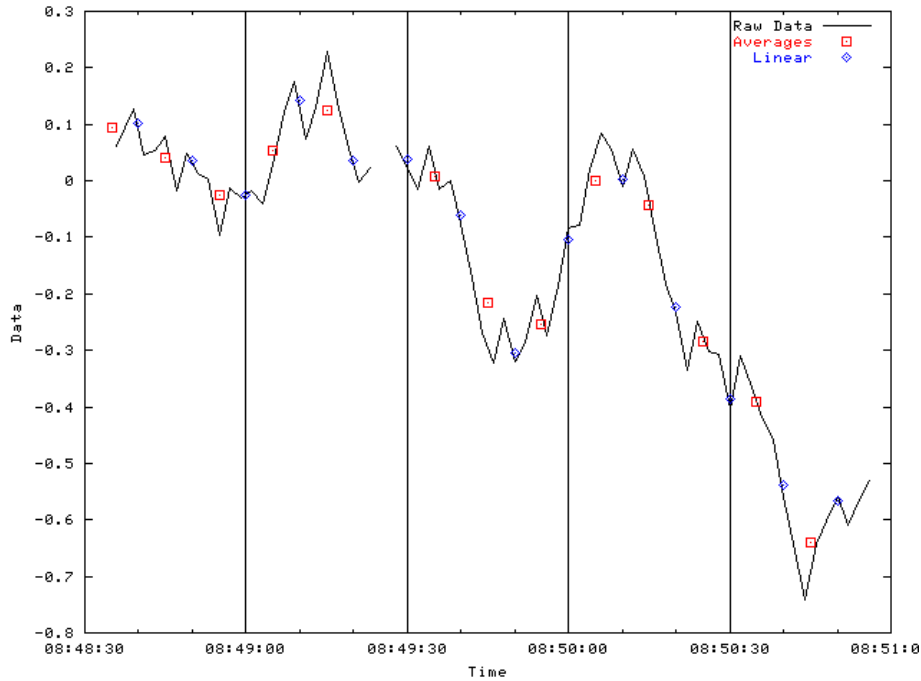


Figure 2.2: Averaging and Linear Interpolation, see text.

values for the bin-borders at 08:48:40, 08:48:50, ... The linearly interpolated values do in fact exactly fall onto the connecting lines between raw samples if you consider the full time stamps, but the plotting program chosen to produce fig. 2.2 rounds down to full seconds.

Note the gap just before 08:49:30: Since the channel was disconnected, no average is returned for the bin from 08:49:20 to 08:49:30. Averaging and linear interpolation are further limited to scalar, numeric samples of type double, float or int. Arrays or strings will not be interpolated.

NOTE: Averaging and linear interpolation must be used with caution. Both methods can hide important details in the raw data, and it is up to the user to determine when to use them and with what bin size. If for example you want to compare several water tank temperatures and you know that the water temperature can only change slowly, linear interpolation for e.g. every 60 seconds might be a reasonable approach.

On the other hand, consider a channel that monitors radiation counts per minute. The raw data will mostly reflect the fairly constant background radiation. Of interest are probably only temporary 'spikes' in the data, since they indicate radiation incidents that need to be correlated with e.g. beam loss. Interpolation of this type of data over 5 minutes will yield useless results. Most temporary increases in radiation within a bin are lost, you will only see val-

ues close to the background radiation as they were measured around the bin borders. Averaging will show a slight increase for those bins that contain a radiation incident, but the magnitude of those 'spikes' will not at all compare to the raw data.

2.7.5 Plot-Binning

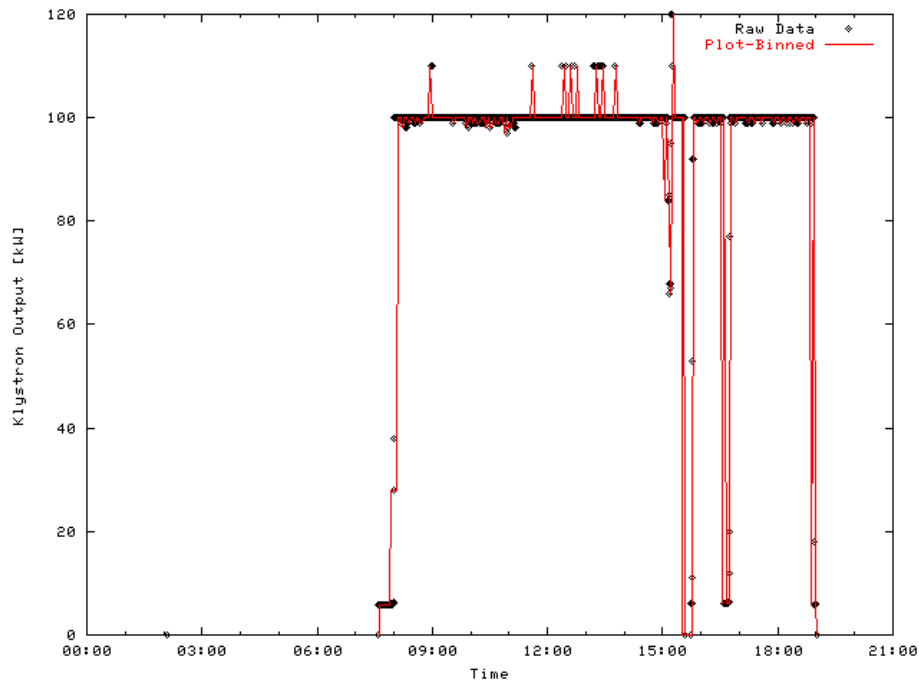


Figure 2.3: Plot-Binning, see text.

This method is meant for plotting, providing data that — when plotted — looks very much if not exactly like the raw data, albeit significantly reducing the number of data points and hence speeding up the plot. To accomplish this, the data is binned as described in the previous section. The following is then applied to each bin:

- If there is no sample for the time span of a bin, the bin remains empty.
- If there is one sample, it is placed in the bin.
- If there are two samples, they are placed in the bin.
- If there are more than two samples, the first and last one are placed in the bin. In addition, two artificial samples are created with a time stamp right

between the first and last sample. One contains the minimum, the other the maximum of all raw samples whose time stamps fall into the bin. They are presented to the user in the sequence initial, minimum, maximum, final.

Fig. 2.2 compares the raw data of a Klystron test run, 2400 samples, with the result of plot-binning, bin size 600 seconds, yielding around 280 samples. While plot binning significantly reduced the sample count, the overall shape of the klystron output as well as the outliers are well preserved.

Note that the “before or at” interpretation of start times does not apply for Plot-Binning: The exact start time of the request is used to determine the beginning of the first bin, and only samples within each bin are considered, there is no interpolation onto bin-boundaries. In general, the use of N bins can result in up to $4N$ data points, since each bin might provide an initial, minimum, maximum and final value. In most cases, this results in a significant data reduction. As long as we plot this such that the width of the plot in pixels is close to the number of bins, there is little visual difference between the raw data plot and the binned plot. Typical numbers for N are around the width of a computer screen in pixels, that is 800...1200. For the special case where 3 raw values happen to fall into every bin, we will get $4N$ instead of $4N$ data points. For typical N , that is a slight but not dramatic increase in retrieval or plotting time. It is neglectable compared to the fact that binning guarantees an upper limit of $4N$ data points, no matter how many raw samples there are.

Chapter 3

ArchiveEngine

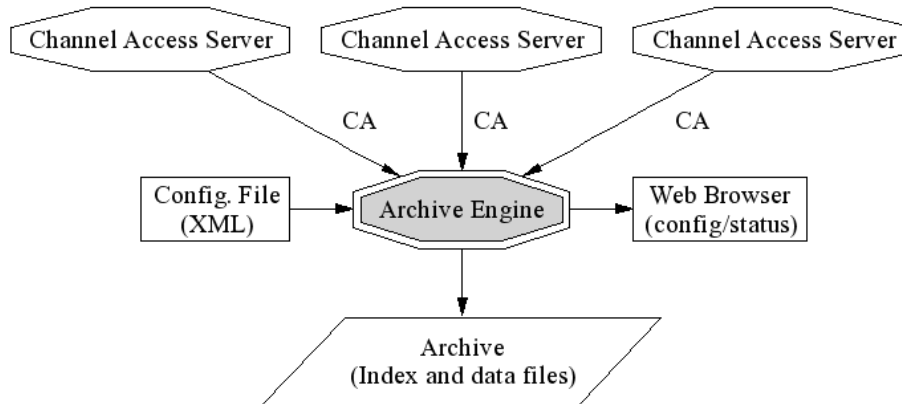


Figure 3.1: Archive Engine, refer to text.

The ArchiveEngine is an EPICS ChannelAccess client. It can save any channel served by any ChannelAccess server. One ArchiveEngine can archive data from more than one CA server. For more details on the CA server data sources, refer to section 2.2 on page 3. The ArchiveEngine supports the sampling options that were described in section 2.3 on page 4. The ArchiveEngine is configured with an XML file that lists what channels to archive and how. Each given channel can have a different periodic scan rate or be archived in monitor mode (on change). One design target was: Archive 10000 values per second, be it 1000 channels that change at 10Hz each or 10000 channels which change at 1Hz.

The ArchiveEngine saves the full information available via ChannelAccess: The value, time stamp and status as well as control information like units, display and alarm limits, ... The data is written to an archive in the form of local

disk files, specifically index and data files. Chapter 5 provides details on the file formats. While running, status and configuration of the ArchiveEngine are accessible via a built-in web server, accessible via any web browser on the network. The chapter on data retrieval, beginning on page 27, introduces the available retrieval tools that allow users to look at the archived data.

3.1 Configuration

The ArchiveEngine expects an XML-type configuration file that follows the document type description format from listing 3.1 (see section 8.2.1 on DTD file installation). Listing 3.2 provides an example. In the following subsections, we describe the various XML elements of the configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for the ArchiveEngine Configuration -->
<!-- Note that we do not allow empty configurations: -->
<!-- Each config. must contain at least one group, -->
<!-- and each group must contain at least 1 channel. -->
<!ELEMENT engineconfig (( write_period|get_threshold|
                           file_size|ignored_future|
                           buffer_reserve|
                           max_repeat_count|disconnect)*,
                           group+)>
<!ELEMENT group (name,channel+)>
<!ELEMENT channel (name,period,(scan|monitor),disable?)>
<!ELEMENT write_period (#PCDATA)><!-- int seconds -->
<!ELEMENT get_threshold (#PCDATA)><!-- int seconds -->
<!ELEMENT file_size (#PCDATA)><!-- MB -->
<!ELEMENT ignored_future (#PCDATA)><!-- double hours -->
<!ELEMENT buffer_reserve (#PCDATA)><!-- int times -->
<!ELEMENT max_repeat_count (#PCDATA)><!-- int times -->
<!ELEMENT disconnect EMPTY>
<!ELEMENT name (#PCDATA)>
<!ELEMENT period (#PCDATA)><!-- double seconds -->
<!ELEMENT scan EMPTY>
<!ELEMENT monitor EMPTY>
<!ELEMENT disable EMPTY>
```

Listing 3.1: XML DTD for the Archive Engine Configuration

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE engineconfig SYSTEM "engineconfig.dtd">
<engineconfig>
  <write_period>30</write_period>
  <get_threshold>20</get_threshold>
  <file_size>30</file_size>
  <ignored_future>1.0</ignored_future>
  <buffer_reserve>3</buffer_reserve>
  <max_repeat_count>120</max_repeat_count>
  <group>
    <name>Vacuum</name>
    <channel><name>vac1 </name>
      <period>0.1</period><monitor/>
    </channel>
    <channel><name>vac2 </name>
      <period>1</period><monitor/><disable/>
    </channel>
    <channel><name>vac3 </name>
      <period>2</period><scan/>
    </channel>
  </group>
  <group>
    <name>RF</name>
    <channel><name>rf1 </name>
      <period>1</period><monitor/>
    </channel>
    <channel><name>rf2 </name>
      <period>1</period><monitor/>
    </channel>
    <channel><name>rf3 </name>
      <period>1</period><scan/>
    </channel>
  </group>
</engineconfig>

```

Listing 3.2: Example Archive Engine Configuration

3.1.1 write_period

This is a Global Option that needs to precede any group and channel definitions. It configures the write period of the Archive Engine in seconds. The default value of 30 seconds means that the engine will write to Storage every 30 seconds.

3.1.2 get_threshold

This global option determines when the archive engine switches from “Sampled” operation to “Sampled using monitors” as described in section 2.3.

3.1.3 file_size

This global option determines when the archive engine will create a new data file. The default of 100 means that the engine will continue to write to a data file until that file reaches a size of approximately 100 MB, at which point a new data file is created.

3.1.4 ignored_future

Defines “too far in the future” as “now + ignored_future”. Samples with time stamps beyond that time are ignored. Details: For strange reasons, the Engine sometimes receives values with invalid time stamps. The most common example is a “Zero” time stamp: After an IOC reboots, all records have a zero time stamp until they are processed. For passive records, as commonly used for operator input, this time stamp will stay zero until someone enters a value on an operator screen or via a safe/restore utility. The Engine cannot archive those values because the retrieval relies on the values being sorted in time. A zero time stamp does not fit in.

Should an IOC (for some unknown reason) produce a value with an outrageous time stamp, e.g. “1/2/2035”, another problem occurs: Since the archiver cannot go back in time, it cannot add further values to this channel until the date “1/2/2035” is reached. Consequently, future time stamps have to be ignored. (default: 6h)

3.1.5 buffer_reserve

To buffer the samples between writes to the disk, the engine keeps a memory buffer for each channel. The size of this buffer is

$$buffer_reserve \times \frac{write_period}{scan_period}$$

Since writes can be delayed by other tasks running on the same computer as well as disk activity etc., the buffer is bigger than the minimum required: buffer_reserve defaults to 3.

3.1.6 max_repeat_count

When sampling in a scanned mode (as opposed to monitored), the engine stores only new values. As long as a value matches the preceding sample, it is not written to storage. Only after the value changes, a special value marked with a severity of ARCH_REPEAT and a status that reflects the number of repeats is written, then the new sample is added.

This procedure conserves disk space. The disadvantage lies in the fact that one does not see any new samples in the archive until the channel changes, which can be disconcerting to some users. Therefore the max_repeat_count configuration parameter was added. It forces the engine to write a sample even if the channel has not change after the given number of repeats. The default is 120, meaning that a channel that is scanned every 30 seconds will be written once an hour even if it had not changed.

3.1.7 disconnect

This global option selects how “disabled” channels (see 3.1.9) are handled. By default, disabled channels will stay connected via ChannelAccess, but no values are archived. When setting the “disconnect” option, disabled channels will instead disconnect from ChannelAccess and then, later, attempt to reconnect once the channel is again enabled.

In general, it is a good idea to stay with the default. That way we leave the connection handling to the ChannelAccess client library, which is optimized to do this. The engine will still receive new data, and as soon as the channel is re-enabled, it can thus store the most recent value.

The disconnect feature was added for the rare case that you have IOCs that are temporarily off-line, and some PV will tell you about the fact. You can then use that PV to disable and disconnect the affected channels, preventing the ChannelAccess client library from continuing to issue connection attempts. Another example would be that you want to reduce the network load of continuing CA monitors for channels that are archived via monitors at a high rate but disabled. Most likely, though, checking your channels’ update rates or using a temporary archive engine might be the better solution.

3.1.8 group

Every channel belongs to a group of channels. The configuration file must define at least one group. For organizational or aesthetic purposes, you might add more groups. One important use of groups is related to the “disable” feature, see section 3.1.9.

name

This mandatory sub-element of a group defines its name.

3.1.9 channel

This element defines a channel by providing its name and the sampling options. A channel can be part of more than one group. To accomplish this, simply list the channel as part of all the groups to which it should belong.

name

This mandatory sub-element of a channel defines its name. Any name acceptable for ChannelAccess is allowed. The archive engine does not perform any name checking, it simply passes the name on to the CA client library, which in turn tries to resolve the name on the network. Ultimately, the configuration of your data servers decides what channel names are available.

period

This mandatory sub-element of a channel defines the sampling period. In case of periodic sampling, this is the period at which the periodic sampling attempts to operate. In case of monitored channels (see next option), this is the estimated rate of change of the channel. The period is specified in units of seconds.

If a channel is listed more than once, for example as part of different groups, the channel will still only be sampled once. The sampling mechanism is determined by maximizing the data rate. If, for example, the channel “X” is once configured for periodic sampling every 30 seconds and once as a monitor with an estimated period of one second, the channel will in fact be monitored with an estimated period of 1 second.

scan

Either “monitor” or “scan” need to be provided as part of a channel configuration to select the sampling method. True to its name, “scan” selects scanned operation, where the preceding “period” tag determines the sampling period, that is the time between taking samples. As an example, scanned operation with a period of 60 means: Every 60 seconds, the engine will write the most recent value of the channel to the archive.

monitor

As an alternative to the “scan” tag, “monitor” can be used, requesting monitored operation, that is: An attempt is made to store each change received via ChannelAccess. The “period” tag is used to determine the in-memory buffer size of the engine. That means: If samples arrive much more frequently than estimated via the “period” tag, the archive engine might drop samples. (See also “buffer_reserve”, 3.1.5). Some examples, assuming that channel “fred” actually emits monitors at 1 Hz:

- “fred 1 Monitor”
Every value sent by fred is archived. Might be a good idea for some channels, but don’t try to store every value of every PV of your control system indefinitely unless you are prepared to deal with that amount of data.
- “fred 1”
The engine will sample once per second, and the channel changes once a second, so you might think that you archive every value just as in the previous example. But think again. The sampling of the engine on the host and the scanning of the channel on the CA server are not synchronized, plus there are additional network delays. So you will sometimes miss values whenever more than one sample arrived between the engine’s sampling, or get duplicate values whenever no new value arrived between the engine’s sampling. Bad idea.
- “fred 60”
The engine will sample every 60 seconds. This is a very reasonable setup: The channel samples at 1 Hz, so you get frequent updates for the operator interface, but for the archive we only care about a sample per minute and save storage space by ignoring finer detail.
- “fred 10 Monitor”
Probably an error. The engine is instructed to save every incoming monitor, but you told it to allocate buffers for only one value every 10 seconds, even though we knew that the channel will emit monitors much more often. So you will get “overflow” errors, the engine will overwrite older samples in its ring buffer with newly arriving samples, and the archive will contain the last few samples that happened to be in the buffer at write-to-disk time.

disable

This optional sub-element of a channel turns the channel into a “disabling” channel for the group. Whenever the value of the channel is above zero, sampling of the whole group will be disabled until the channel returns to zero or below zero (see 3.1.7 for additional disconnection).

This is useful for e.g. a group of channels related to power supplies: Whenever the power supply is off, we might want to disable scanning of the power supplies’ voltage and current because those channels will only yield noise. By disabling the sampling based on a “Power Supply is Off” channel, we can avoid storing those values which are of no interest.

NOTE: There is no “enabling” feature, meaning: The channel marked as “disable” will disable its group whenever it is above zero. There is no “enable” flag that would enable archiving of a group whenever the flagged channel is above zero. If you want it the other way around, you typically add a CALC record to handle the inversion.

3.2 Starting and Stopping

3.2.1 Starting

The ArchiveEngine is a command-line program that displays usage information similar to the following:

```
ArchiveEngine Version 2.1.0 , EPICS 3.14.4
```

```
USAGE: ArchiveEngine [ Options ] <config-file> <index-file>
```

Options :

-port <port>	Web server TCP port
-description <text>	description for HTTP display
-log <filename>	write logfile
-nocfg	disable online configuration

Minimally, the engine is therefore started by simply naming the configuration file and the path to the index file, which can be in the local directory:

```
ArchiveEngine engineconfig.xml ./index
```

After collecting some data, the ArchiveEngine will create the specified index file together with data files in the same directory that contains the index file.

3.2.2 “-log” Option

This option causes the ArchiveEngine to create a log file into which all the messages that otherwise only appear on the standard output are copied.

3.2.3 “-description” Option

This option allows setting the description string that gets displayed on the main page of the engine's built-in HTTP server, see section [3.3](#).

3.2.4 “-port” Option

This option configures the TCP port of the engine's HTTP server, again see section [3.3](#). The default port number is 4812.

If you think this number stinks for a default, you are not too far off base: In Germany, there is a very well known Au-de-Cologne called 4711. Since forty-seven-eleven is therefore easily remembered by anybody from Germany, adding 1 to each 47 and 11 naturally results in an equally easy to remember 4812. And for those who fail to appreciate the German-centered default port number, the “-port” option allows you to pick a number of your personal fancy.

3.2.5 “-nocfg” Option

This option disables the “Config” page of the engine’s HTTP server, in case you want to prohibit online changes.

3.2.6 The “archive_active.lock” File

You can only run one ArchiveEngine per directory because it creates the index and data files in there. When running, this lock file is created. The ArchiveEngine will refuse to run if this file already exists. After shutdown, the ArchiveEngine will remove this lock file. If the ArchiveEngine crashes or is not stopped gracefully by the operating system, this lock file will be left behind. You cannot start the ArchiveEngine again until you remove the lock file. This is a reminder for you to check the cause of the improper shutdown and maybe check the data files for corruption.

NOTE: This is no 100% dependable check. Data corruption occurs when two engines attempt to write to the same index and data files. The lock file, however, is created in the directory where the ArchiveEngine was started, which could be different from the directory where the data gets written. Example:

```
cd /some/dir
ArchiveEngine -p 7654 engineconfig.xml /my/data/index &
```

```
cd /another/dir
ArchiveEngine -p 7655 engineconfig.xml /my/data/index &
```

This is a sure-fire way to corrupt the data in “/my/data/index” and the accompanying data files because two ArchiveEngines are writing to the same archive.

3.2.7 More than one ArchiveEngine

You can run multiple ArchiveEngines on the same computer. But they must

1. be in separate directories. See the preceding discussion of the lock file which is meant to assist in avoiding this problem.
2. use a different TCP port number for the built-in web server

In practice this means that you have to create different directories on the disk, one per ArchiveEngine, and in there run the ArchiveEngines with different “-p <port>” options.

3.2.8 Stopping

While the ArchiveEngine can be stopped by pressing “CTRL-C” or using the equivalent “kill” command in Unix, the preferred method is via the built-in web server. Use any web browser and point it to

`http://<host where engine is running>:<port>/stop`

Per default, the engine uses 4812, so you could use the following URL to stop that engine on the local computer:

`http://localhost:4812`

3.3 Web Interface

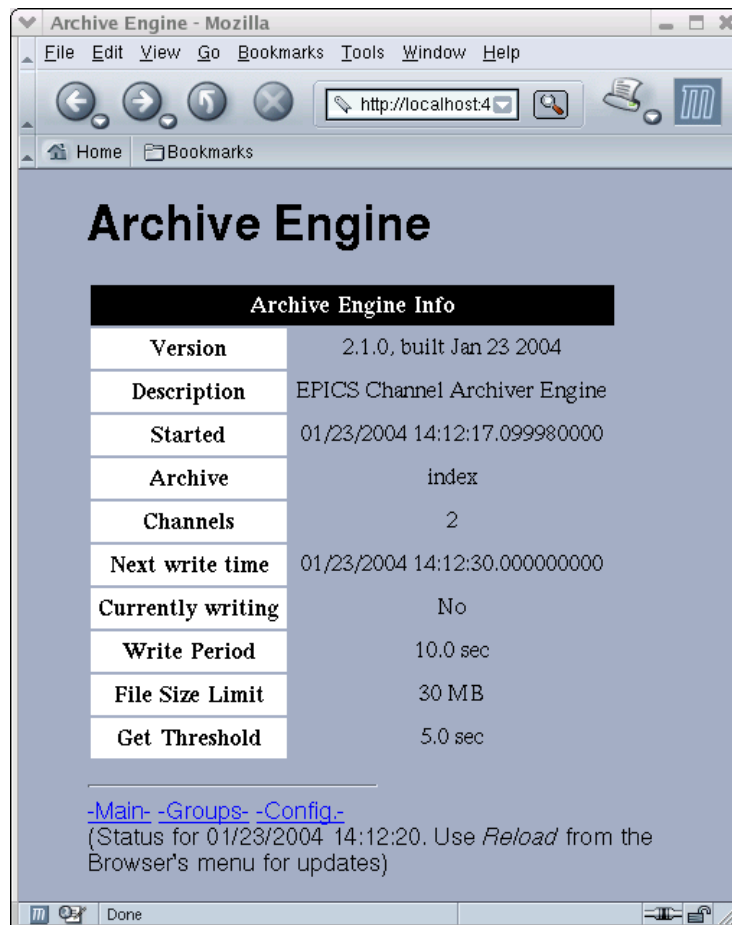


Figure 3.2: Main Page of Archive Engine's HTTPD

The ArchiveEngine has a built-in web server (HTTP Daemon) for status and configuration information. You can use any web browser to access this web server. You can do that on the computer where the ArchiveEngine is running as well as from other computers, be it a PC or Macintosh or other

system as long as that computer can reach the machine that is running the ArchiveEngine via the network. You do *not* need a web server like the Apache web server for Unix or the Internet Information Server for Win32 to use this. The ArchiveEngine *itself* acts as a web server.

You *cannot* view archived data with this mechanism. See the documentation on data retrieval (chapter 4) for that, because the archive engine's HTTPD is meant for access to the status and configuration of the running engine, not for accessing the data samples.

To access the ArchiveEngine's web server, you need to know the Internet name of the machine that is running the ArchiveEngine as well as the TCP port. If you are on the same machine, use "localhost". The port is configured when you start the ArchiveEngine, it defaults to 4812. Then use any web browser and point it to

```
http://<host where engine is running>:<port>
```

Example for an ArchiveEngine running on the local machine with the default port number:

```
http://localhost:4812
```

The start page of the ArchiveEngine web server should look similar to the one shown in Fig. 3.2. By following the links, one can investigate the status of the groups and channels that the ArchiveEngine is currently handling. The "Config" page allows limited online-reconfiguration. Whenever a new group or channel is added, the engine attempt to write a new config file called onlineconfig.xml in the directory where it was started. It is left to the user to decide what to do with this file: Should it replace the original configuration file, so that online changes are preserved? Or should it be ignored, because online changes are only meant to be temporary and with the next run of the engine, the original configuration file will be used?

Note also that the ArchiveEngine does not allow online removal of channels and groups. The scan mechanism of a channel can only be changed towards a higher scan rate or lower period, similar to the handling of multiply defined channels in a configuration file. Refer to the section discussing the "period" tag on page 20.

3.4 Threads

The ArchiveEngine uses several threads:

- A main thread that reads the initial configuration and then enters a main loop for the periodic scan lists and writes to the disk.
- The ChannelAccess client library is used in its multi-threaded version. The internals of this are beyond the control of the ArchiveEngine, the total number of CA client threads is unknown.

- The ArchiveEngine's HTTP (web) server runs in a separate thread, with each HTTP client connection again being handled by its own thread. The total number of threads therefore depends on the number of current web clients.

As a result, the total number of threads changes at runtime. Though these internals should not be of interest to end users, this can be confusing especially on older releases of Linux where each thread shows up as a process in the process list. On Linux version 2.2.17-8 for example we get process table entries as shown in Tab. 3.3 for a single ArchiveEngine, connected to four channels served by excas, no current web client. The only hint we get that this is in fact one and the same ArchiveEngine lies in the consecutive process IDs.

PID	TTY	TIME	CMD
29721	pts /5	00:00:00	ArchiveEngine
29722	pts /5	00:00:00	ArchiveEngine
29723	pts /5	00:00:00	ArchiveEngine
29724	pts /5	00:00:00	ArchiveEngine
29725	pts /5	00:00:00	ArchiveEngine
29726	pts /5	00:00:00	ArchiveEngine
29727	pts /5	00:00:00	ArchiveEngine
29728	pts /5	00:00:00	ArchiveEngine

Listing 3.3: Output of Linux 'ps' process list command, see text.

The first conclusion is that one should not be surprised to see multiple ArchiveEngine entries in the process table. The other issue arises when one tries to 'kill' a running ArchiveEngine. Though the preferred method is via the engine's web interface, one can try to send a signal to the first process, the one with the lowest PID.

Chapter 4

Data Retrieval

Data retrieval requirements can cover a wide range. One person might be interested in the temperature of a water tank during the last night. For this, it is probably sufficient to retrieve the raw data for the respective channel and plot it. If, on the other hand, we want to look at the same temperature for the last 3 month, the raw data will amount to too many samples and some sort of data reduction or interpolation is helpful. We already mentioned the problems of time stamp correlation that arise when comparing different channels in section [2.7](#).

The Channel Archiver toolset includes some generic tools that can be used “as is”. While those try to cover many data retrieval requirements, certain requests can only be handled in customized data mining programs (which might be e.g. perl scripts). For this, the archiver offers a network data server. In short, these are your options:

- Java Archive Client
This is meant to be *the* data client. Use it to browse the available data, generate plots, export data to spreadsheets, from any computer on the network by accessing the network data server.
- ArchiveExport
A command-line tool. Less convenient to use, requires direct access to the data files. Use this when the Java Archive Client or network data server are not available.
- Archive Data Server
You can access the Archive Data Server via XML-RPC from most programming languages. Use this method for customized data mining programs.

4.1 ArchiveExport

ArchiveExport is a command-line tool for local tests, i.e. it does *not* connect to the archive data server but requires that you have direct read access to the index and data files. It is mostly meant for testing.

When invoked without valid arguments, it will display a command description similar to this:

USAGE: ArchiveExport [Options] <index file> {channel}

Options:

-verbose	Verbose mode
-list	List all channels
-info	Time-range info on channels
-start <time>	Format: "mm/dd/yyyy[_hh:mm:ss[.nano-secs]]"
-end <time>	(exclusive)
-text	Include text column for status/severity
-match <reg. exp.>	Channel name pattern
-interpolate <seconds>	interpolate values
-output <file>	output to file
-gnuplot	generate GNUPlot command file
-Gnuplot	generate GNUPlot output for Image

ArchiveExport produces spreadsheet-type output in TAB-separated ASCII text, suitable for import into most spreadsheet programs for further analysis. Per default, ArchiveExport uses Staircase Interpolation to map the data for the requested channels onto matching time stamps, but one can select Linear Interpolation via the "-interpolate" option. When GNUPlot output is selected, the Plot-Binning method is used, where the bin size is determined by the "-interpolate" argument. See section 2.7 on page 9 for details.

Assuming that your current working directory contains an archive index file that is aptly named "index", the following invocation will generate a spreadsheet file "data.txt" with the data of all channels that match the pattern "IOC" for the date of January 27, 2003:

```
ArchiveExport index -m IOC \
    -s "01/27/2003" \
    -e "01/28/2003" >data.txt
```

To plot this in OpenOffice, you could create a new spreadsheet, then use the menu item Insert/Sheet/FromFile, select the file "data.txt" and configure the text import dialog to use "Separated by Tab". You will notice that even though the original text file contains time stamps with nano-second resolution, for example "01/27/2003 23:57:25.579346999", the spreadsheet program might use a default representation of e.g. "01/27/03 23:57 pm". In order to see the full

time stamp detail, one needs to reformat those spreadsheet columns with a user-defined format like “MM/DD/YYYY HH:MM:SS.000000000”. If you use Microsoft Excel, you might be limited to a format with millisecond resolution: “MM/DD/YYYY HH:MM:SS.000”. For graphing the data, the most suitable option is often an “X-Y-Graph”, using the first row for labels, with the data series taken from the columns.

The following call sequence will generate a GNUPlot data file “data” together with a GNUPlot command file “data.plt” and execute it within GNUPlot:

```
ArchiveExport index -m IOC \
                    -s "01/27/2003" \
                    -e "01/28/2003" -o data -gnu
gnuplot
      G N U P L O T
      Version 3.7 patchlevel 3
      ....
gnuplot> load 'data.plt'
```

4.2 Java Archive Client

This tool is meant to be the main data retrieval tool. It provides a graphical user interface to allow data browsing. It is based on Java and hence usable on many operating systems. It accesses the data via the DataServer described in section 4.3, which means that it can access the data via the network. You invoke the java archive client with the URL of the web server that hosts the archive data server. The interactive GUI shown in Fig. 4.1 then allows you to select one of the archives served by the network data server, investigate the available channels, perform basic calculations on single or multiple channels, export data in spreadsheet format and much more. A separate manual describes the viewer.

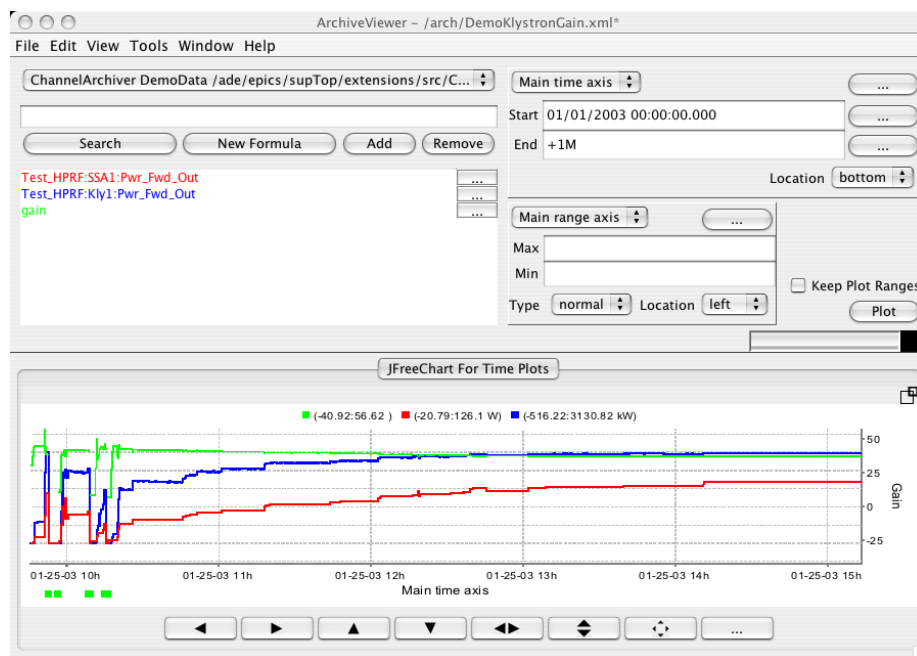


Figure 4.1: Java Archive Client.

4.3 Data Server

The archiver toolset includes a network data server. By running this data server on a computer that has physical access to your archived data, be it because the data resides on a local disk or an NFS-mapped volume, other machines on the network can get read-access to your data.

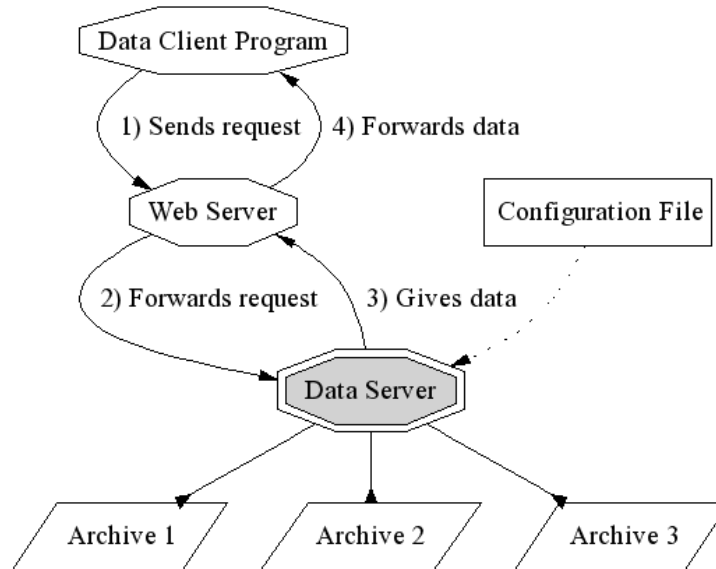


Figure 4.2: Data Server, refer to text.

The data server is hosted by a web server, using the XML-RPC protocol to serve the data. This means that software running on disparate operating systems, running in different environments can access your data over the Internet via a URL. As an example, your data server might be a Linux machine on a subnet behind a firewall. After you configure the firewall to pass HTTP requests, any Linux, Win32, Macintosh computer both inside or outside of the firewall can access the data from within perl, python or tcl scripts, programs written in C, C++ or Java, actually pretty much any programming language. As illustrated in fig. 4.2, the client program sends its requests to a web server, which forwards it to the data server that is running as a CGI tool under the web server. The dataserver accesses the relevant archives — you determine which ones are available via a configuration file — and returns the data though the web server to the client program. You can configure access security via e.g. the Apache web server configuration.

NOTE: The fact that the data server is hosted by a web server, accessible via a URL, does *not* imply that you can use any web browser to retrieve data. You have to use the XML remote procedure call protocol described in section 4.4 on

page 37. XML-RPC handles the network connections, data type conversions, and XML-RPC libraries are available for most programming languages. We provide the Java Archive Client (see section 4.2) as a generic, interactive data client.

4.3.1 Installation

After successful compilation in ChannelArchiver/XMLRPCServer, you should have a program “XMLRPCServer/O.\$EPICS_HOST_ARCH/ArchiveDataServer”. You have to install it as a CGI tool under a web server, passing an environment variable “SERVERCONFIG” to it which points to a configuration file. This means

- You need a web server like Apache running on a computer with access to the archive data.
- You need to know how to start, stop and configure that web server.
- You need to know how to install, configure and possibly troubleshoot CGI tools under that web server.
- You need to create a configuration file for the ArchiveDataServer.

With all but the last point you are very much on your own, because this manual cannot even try to describe all the possible configurations and errors. What follows is only an example setup for the Apache Web Server under Linux and Mac OS X. Your web server, even if it is also a version of Apache, is likely to be quite different.

1. Locate your web server configuration file. This is often a variant of “/etc/httpd/conf/httpd.conf”. For Mandrake 10, check “/etc/httpd/conf/commonhttpd.conf”, for Mac OS X it’s “/etc/httpd/httpd.conf”.
2. After each change to the configuration, you will have to restart the web server. This is often done via “/etc/rc.d/initd/httpd restart” or “/usr/sbin/apachectl restart”.
3. Create a new web directory for the archiver with a CGI sub-directory. This is typically done under /var/www/html, except for Mac OS X, where you would use /Library/WebServer/Documents. Check the “DocumentRoot” variable in your web server configuration file for the correct location and adjust the following accordingly.

```
mkdir /var/www/html/archive
mkdir /var/www/html/archive/cgi
```

Change the permissions of those directories to your liking. Usually, “everybody” needs read and execution access, because the web server will

run CGI programs as a low-privileged user. Our main interest here is the CGI“cgi” subdirectory. You can use the “archive” directory to store e.g. the dtd files or web pages with user information that relate to the archive setup at your site.

4. Copy the ArchiveDataServer binary into the cgi directory as “ArchiveDataServer.cgi”. In this example, both the “cgi” directory and the “.cgi” extension of “ArchiveDataServer.cgi” are important, because they identify the binary as a CGI program.
5. Assert that the web server can execute the ArchiveDataServer, that it recognizes it as a CGI tool, by adding the following to the Apache config file:

```
# Check that environment variables are available ,
# assert that this directive is not commented-out.
# Your web server might use a different module name
# or location , in my case it happened to be this:
LoadModule env_module libexec/httpd/mod_env.so
AddModule mod_env.c

# Check that CGI is enabled , i.e. assert that
# these are not commented-out:
LoadModule cgi_module libexec/httpd/mod_cgi.so
AddModule mod_cgi.c

# This tells Apache that ArchiveDataServer.cgi
# is a CGI program because of the .cgi extension:
AddHandler cgi-script .cgi

<Directory /var/www/html/archive>
    Order Allow,Deny
    Allow from All
</Directory>

# Allow cgi-scripts in the ....cgi directory:
<Directory /var/www/html/archive/cgi>
    SetEnv EPICS_TS_MIN_WEST 300
    SetEnv LD_LIBRARY_PATH /usr/local/lib:...
    SetEnv SERVERCONFIG \
        /var/www/html/archive/cgi/serverconfig.xml

    # I also like to enable perl-CGI, but that is
    # unrelated to the archiver
    PerlHandler Apache::Registry
    PerlSendHeader On
```

```
# This directive enables CGI for this dir.:
Options +ExecCGI
</Directory>
```

The LD_LIBRARY_PATH needs to list all the directories that contain shared libraries which your ArchiveDataServer.cgi uses. In most cases, this includes the

- install location of the expat and XML-RPC libraries, often /usr/local/lib,
- “lib” subdirectories of EPICS base and EPICS extensions, something like /ade/epics/supTop/base/R3.14.6/lib/linux-x86:...

The SERVERCONFIG variable needs to point to your server configuration file, more about which next. The ExecCGI option is essential to allow CGI functionality. You can skip the Perl configuration for the data server.

4.3.2 Configuration

You need to prepare an XML-formatted configuration file for the ArchiveDataServer that follows the DTD from listing 4.1 (see section 8.2.1 on DTD file installation). Note that the ArchiveDataServer might not verify your configuration file, so you are strongly encouraged to use a tool like ‘xmllint’ on Linux to check your configuration against the DTD. Listing 4.2 shows one example configuration which lists two archives to be served. Client programs will internally use the respective ‘key’ to access them.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for the XML-RPC Data Server Configuration -->
<!ELEMENT serverconfig (archive+)>
<!ELEMENT archive (key,name,path)>
<!ELEMENT key (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT path (#PCDATA)>
```

Listing 4.1: XML DTD for the Data Server Configuration

4.3.3 Testing, Debugging

Section 4.5 describes a perl client to the network data server. It can be used to see if all the archives you listed in the configuration are actually accessible and so on.

In case that client tool gets nothing but errors, debugging of the CGI ArchiveDataServer can be difficult, since one cannot easily peek into the running (or failing to run) program when it is launched by the web server.


```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE serverconfig SYSTEM "serverconfig.dtd">
<serverconfig>
  <archive>
    <key>1</key>
    <name>Vacuum</name>
    <path>/home/data/vac/index</path>
  </archive>
  <archive>
    <key>2</key>
    <name>RF-LLRF</name>
    <path>/home/data/llrf/index</path>
  </archive>
</serverconfig>

```

Listing 4.2: Example Data Server Configuration

The XMLRPCServer directory contains some self-tests in test.sh, where the ArchiveDataServer is run without an actual web server. If test.sh works, your ArchiveDataServer program is fundamentally functional.

If it doesn't work from within your web server, try changing your user ID to 'guest' or 'nobody', somebody similar to the user ID used by the web server when it runs your CGI tool. Set the LD_LIBRARY_PATH as you set it for the web server cgi directory, and try test.sh again.

You can also try this to see more of the web server response:

```

telnet my_web_server_host 80
GET /archive/cgi/ArchiveDataServer.cgi HTTP/1.0<RETURN>
<RETURN>

```

You should see an error message about "XML-RPC Fault: Expected HTTP method POST", indicating that the data server was launched correctly and was looking for a proper XML-RPC request. If you see only pages of binary-looking garbage, your web server doesn't recognize ArchiveDataServer.cgi as a CGI program, and instead of running it, you get a copy of it.

4.3.4 Standalone Data Server

The preferred deployment of the network data server is within a standard web server as described in the previous sections. For experiments, however, there is a way to use a version of the data server which combines the simple "Abyss" web server and the network data server into one standalone program.

Advantages:

- Easier to configure than a standard web server. No serverconfig.xml, no serverconfig.dtd, no CGI setup trouble. When running "ArchiveDataServer-

Standalone”, you will see right away if e.g. a shared library is missing, while the “ArchiveDataServer” CGI-plugin to a web server would simply not run for reasons harder to diagnose.

- Ordinary users beyond “root” or “Administrator” can run the standalone data server.

Disadvantages:

- The Abyss HTTPD still requires some configuration.
- Compared to e.g. the Apache web server, there is much less control over who can access the data.
- Currently limited to serving a single archive per running instance of the standalone data server. The ‘key’ of that archive is fixed to ‘1’ and the ‘name’ to ‘Archive’. Under a standard web server, one can run more than one ArchiveDataServer and configure each one to serve multiple archives with selected key and name.

4.3.5 Running ArchiveDataServerStandalone

After successful compilation in ChannelArchiver/XMLRPCServer, you will have a program “ArchiveDataServerStandalone”.

1. For starters, you can test in the “ChannelArchiver/DemoData” directory, but typically you would copy all “abys*” files and directories from there into the directory where you intend to run the standalone data server.
2. Edit “abyss.conf” to suit your needs. Most users might only have to adjust the “Port” option, which defaults to 8080, and the “ServerRoot” variable.
3. Run the data server with the abyss config file and the path of the archive’s index. When inside DemoData, this would be an example:

```
> ArchiveDataServerStandalone abyss.conf index
ArchiveDataServerStandalone Running
Unless your 'abyss.conf' selects
a different port number,
the data should now be available via the XML-RPC URL
http://localhost:8080/RPC2
```

The data is now accessible via XML-RPC by pointing the ArchiveDataClient.pl test tool or the Java ArchiveViewer to the URL

```
http://<hostname>:<port>/RPC2
```

Example:

```
http://localhost:8080/RPC2
```

4.4 XML-RPC Protocol

The following is a description of the calls implemented by the archive data server based on the XML-RPC protocol. For details on XML-RPC, including the specifications and examples of how to use it from within C, C++, Java, perl, please refer to <http://www.xmlrpc.com>.

Users of Java should probably utilize the Java archive data client library provided with the ChannelArchiver. Users of other programming environments need to refer to the following.

4.4.1 archiver.info

This call returns version information. It will allow future compatibility if clients check for the correct version numbers. In addition, it provides hints on how to decode the values served by this server.

```
{ int32      ver ,
  string     desc ,
  string     how[] ,
  string     stat[] ,
  { int32 num,
    string sevr ,
    bool has_value ,
    bool txt_stat
  }         sevr[]
} = archiver.info()
```

ver: Version number. The first released software uses '1'.

desc: Cute description that one can print.

how: Array of strings with a description of the request methods supported for 'how' in the archiver.values() call described further below in section 4.4.4.

stat: Array of strings with a description of the "status" part of the values returned by the archiver.values() call.

sevr: Array of structures with a description of the "severity" part of the values returned by the archiver.values() call.

The result is a structure with a numeric "ver" member, a string "desc" member and so on as listed above. Implementations like perl will return a hash with members "ver", "desc", etc. The strings in "how" describe the request method for how=0, how=1, and so on. The strings in "stat" describe the enumerated status values, the typical result is shown in table 4.1.

The more important information is in the "sevr" array. It also lists severity numbers ("num") and their associated string representation ("sevr"). In addition

to the alarm severities defined by the EPICS base software, the archiver uses some special severity values which have the “has_value” property set to false. They identify situations that have no value because the channel was disconnected or the archiver was turned off. Other special severities identify repeat counts which are used in the periodic scanning modes of the archive engine: If the channel did not change for N sample times, a repeat count of N is logged instead of logging the same value N times. In that case, the “txt_stat” property is set to false because the status (stat) field no longer corresponds to a status string from table 4.1. Instead, it provides the repeat count N. Table 4.2 lists the typical content of the “sevr” array, table 4.3 presents examples for decoding values based on their status and severity information.

Array Element	String
0	NO_ALARM
1	READ_ALARM
2	WRITE_ALARM
3	HIHI_ALARM
4	HIGH_ALARM
5	LOLO_ALARM
6	LOW_ALARM
7	STATE_ALARM
...	...
17	UDF_ALARM
...	...

Table 4.1: Alarm Status Values returned in the “stat” member of archiver.info()

num	sevr	has_value	txt_stat
0	NO_ALARM	true	true
1	MINOR	true	true
2	MAJOR	true	true
3	INVALID	true	true
3968	Est_Repeat	true	false
3856	Repeat	true	false
3904	Disconnected	false	true
3872	Archive_Off	false	true
3848	Archive_Disabled	false	true

Table 4.2: Alarm Severity Values returned in the “sevr” member of archiver.info()

Severity (sevr)	Status (stat)	Value	Example Text
0	0	3.14	"3.14"
1	6	3.14	"3.14 MINOR LOW"
3856	6	3.14	"3.14 Repeat 6"
3904	0	0	"Disconnected"

Table 4.3: Examples for decoding samples returned from the `archiver.values()` call based on their Status and Severity

4.4.2 archiver.archives

Returns the archives that this data server can access.

```
{ int32 key,
  string name,
  string path }[] = archiver.archives()
```

key: A numeric key that is used by the following routines to select the archive.

name: A description of the archive that one could e.g. use in a drop-down selector in a GUI application for allowing the user to select an archive.

path: The path to the index file, valid on the file system where the data server runs. It might be meaningful to a few users who want to know exactly where the data resides, but it is seldom essential for XML-RPC clients to look at this.

The result is an array of structures with a numeric "key" member and strings "name" and "path". An example result could be:

```
{ key=1, name="Vacuum", path="/home/data/vac/index" },
{ key=2, name="RF", path="/home/data/RF/index" }
```

So in the following one would then use `key=1` to access vacuum data etc. One can expect the keys to be small, positive numbers, but they are not guaranteed to be consecutive as 1, 2, 3, ... Since the keys could be something like 10, 20, 30 or 1, 17, 42, they are not useful as array indices.

4.4.3 archiver.names

Returns channel names and start/end times. The key must be a valid key obtained from `archiver.keys`. Pattern is a regular expression; if left empty, all names are returned.

NOTE: The Time Stamps are *not* the raw EPICS time stamps with 1990 epoch, but use the `time.t` data type based on a 1970 epoch.

```
{string name,
  int32 start_sec ,  int32 start_nano ,
  int32 end_sec ,    int32 end_nano }[]
  = archiver.names(int32 key,  string pattern)
```

The result is an array of structures, one structure per channel that matches the pattern. Start/end gives an idea of the time range that can be found in the archive for that channel. The archive might actually contain entries *after* the reported end time because the index might not be up too date on the end times.

4.4.4 archiver.values

This call returns values from the archive identified by the key for a given list of channel names and a common time range.

```
result = archiver.values(
  int key,
  string name[],
  int32 start_sec ,  int32 start_nano ,
  int32 end_sec ,    int32 end_nano , int32 count ,
  int32 how)
```

The parameter "how" determines how the raw values of the various channels get arranged to meet the requested time range and count. For details on the methods mentioned in here refer to section 2.7 and following, beginning on page 9.

how = 0 (raw): Get raw data from archive (see 2.7.1), starting w/ 'start', up to either 'end' time or max. 'count' samples.

how = 1 (spreadsheet): Get data that is filled or staircase-interpolated, starting w/ 'start', up to either 'end' time or max. 'count' samples (see 2.7.3). For each channel, the same number of values is returned. The time stamps of the samples match accross channels, so that one can print the samples for each channel as columns in a spreadsheet. If a spreadsheet cell is empty because the channel does not have any useful value for that point in time, a status/severity of UDF/INVALID is returned (Tables 4.2 and 4.1).

how = 2 (averaged): Get averaged data from the archive, starting w/ 'start', up to either 'end' time or max. 'count' samples (see 2.7.4). The data is averaged within bins whose size is determined by of (end-start)/count, so you should expect to get close to 'count' values which cover 'start' to 'end'. Again refer to section 2.7.

how = 3 (plot binning): Uses the plot-binning method based on 'count' bins (see 2.7.5).

how = 4 (linear): Get linearly interpolated data from the archive, starting w/ 'start', up to either 'end' time or max. 'count' samples (see 2.7.4). The data is interpolated onto time slots which are multiples of (end-start)/count, so you should expect to get close to 'count' values which cover 'start' to 'end'. Again refer to section 2.7.

The result is an array of structures, one structure per requested channel:

```
result := { string name, meta, int32 type,
            int32 count, values }[]
```

name: The channel name. Result[i].name should match name[i] of the request, so this is a waste of electrons, but it's sure convenient to have the name in the result, and we're talking XML-RPC, so forget about the electrons.

meta: The meta information for the channel. This is itself a structure with the following entries:

```
meta := { int32 type;
           type==0: string states[],
           type==1: double disp_high,
                   double disp_low,
                   double alarm_high,
                   double alarm_low,
                   double warn_high,
                   double warn_low,
                   int prec, string units
         }
```

type: Describes the data type of this channel's values:

```
string    0
enum      1 (XML int32)
int       2
double    3
```

count: Describes the array size of this channel's values, using 1 for scalar values. Note that even scalar values are returned as an array with one element!

values: This is an array where each entry is a structure of the following layout:

```
values := { int32 stat, int32 sevr,
            int32 secs, int32 nano,
            <type> value[] } []
```

The values for status and severity match in part those that the EPICS IOC databases use. The ArchiveEngine simply receives and stores them, they are passed on to the retrieval tools without change. In addition, the archiver toolset uses special severity values to indicate a disconnected channel or the fact that the ArchiveEngine was shut down. For details refer to section 4.4.1 and the tables 4.1, 4.2 and 4.3.

4.4.5 Note about Tiny Numbers and Precision

Some systems deal with small numbers. Vacuum readings often use numbers like $5 \cdot 10^{-8}$. The XML-RPC specification is unfortunately unclear how numbers should get serialized and parsed other than specifically prohibiting the exponential notation. The best one could serialize the example number would therefore be "0.00000005".

When the ArchiveDataServer is build with the XML-RPC library for C/C++ as described in the installation section, 8, it will attempt to properly serialize small numbers. When using another XML-RPC library, and this includes the XML-RPC library that your client program uses, small numbers might end up being serialized as zero. The Java Archive Client appears to handle small numbers, as does the "Frontier" XML-RPC library for perl.

A similar issue applies to the precision of floating point numbers: The ArchiveDataServer serializes numbers with a fixed precision that is determined by the XML-RPC library for C/C++. You can patch the library to increase the precision.

4.5 Perl Client

The ArchiveDataClient.pl perl script is provided as a starting point for users who want to write perl scripts that access the ArchiveDataServer via XML-RPC. It requires the installation of the "Frontier" XML-RPC library for Perl. The ArchiveDataClient script might also help you test your ArchiveDataServer setup because it offers a command-line interface that is very close to the underlying XML-RPC calls. What follows is an example session:

USAGE: ArchiveDataClient.pl [options] { channel names }

Options:

```

-u URL      : Set the URL of the DataServer
-i          : Show server info
-a          : List archives (name, key, path)
-k key      : Specify archive key.
-l          : List channels
-m pattern  : ... that match a patten
-h how      : 'how' number; retrieval method
-s time     : Start time MM/DD/YYYY HH:MM:SS.NNNNNNNNN
-e time     : End time MM/DD/YYYY HH:MM:SS.NNNNNNNNN
```



```

-c count      : Count

$ URL=http://localhost/cgi-bin/xmlrpc/ArchiveDataServer.cgi
$ ArchiveDataClient.pl -u $URL -i
Archive Data Server V 0
Description:
Channel Archiver Data Server V0
Config '/var/www/cgi-bin/xmlrpc/serverconfig.xml'
Supports how=0 (raw), 1 (spreadsheet),
           2 (interpol/average), 3 (plot-binning)
$ ArchiveDataClient.pl -u $URL -a
Archives:
Key 1: 'Xmtr_2002' in '/home/....2002/index'
Key 2: 'Xmtr_2003' in '/home/....2003/index'
$ ArchiveDataClient.pl -u $URL -k 1 -m IOC1:Load
Channels:
Channel Test_HPRF:IOC1:Load,
      11/01/2002 17:09:37.616190999
      - 12/31/2002 23:59:45.579346999
$ ArchiveDataClient.pl -u $URL -k 1 \
-s "12/01/2002" -e "12/31/2002" \
-h 2 -c 31 Test_HPRF:IOC1:Load
Result for channel 'Test_HPRF:IOC1:Load':
Display : 0.000000 ... 100.000000
Alarms   : 0.000000 ... 80.000000
Warnings: 0.000000 ... 50.000000
Units    : '%', Precision: 0
Type: 3, element count 1.
11/30/2002 23:11:36.774193571 11.820585
12/01/2002 22:25:09.677419377 11.846808
12/02/2002 21:38:42.580645183 12.310933
12/03/2002 20:52:15.483870989 0.000000 ARCH_DISCONNECT
12/04/2002 20:05:48.387096795 12.225621
...
12/29/2002 23:58:03.870967751 11.448704

```

4.6 StripTool

StripTool is primarily a Channel Access client for taking “live” samples from CA servers and plotting them over time. For information on the current version, refer to the section on Host Software: Clients under <http://www.aps.anl.gov/epics>. A “History” module allows StripTool to access data from a Channel Archiver network data server.

NOTE: This history module is incomplete. At the time, the StripTool API did not provide means for a history data plug-in to create a configuration GUI. There was also no support for asynchronous data retrieval, meaning StripTool freezes while archive data is requested. Use of the history module is discouraged except for testing.

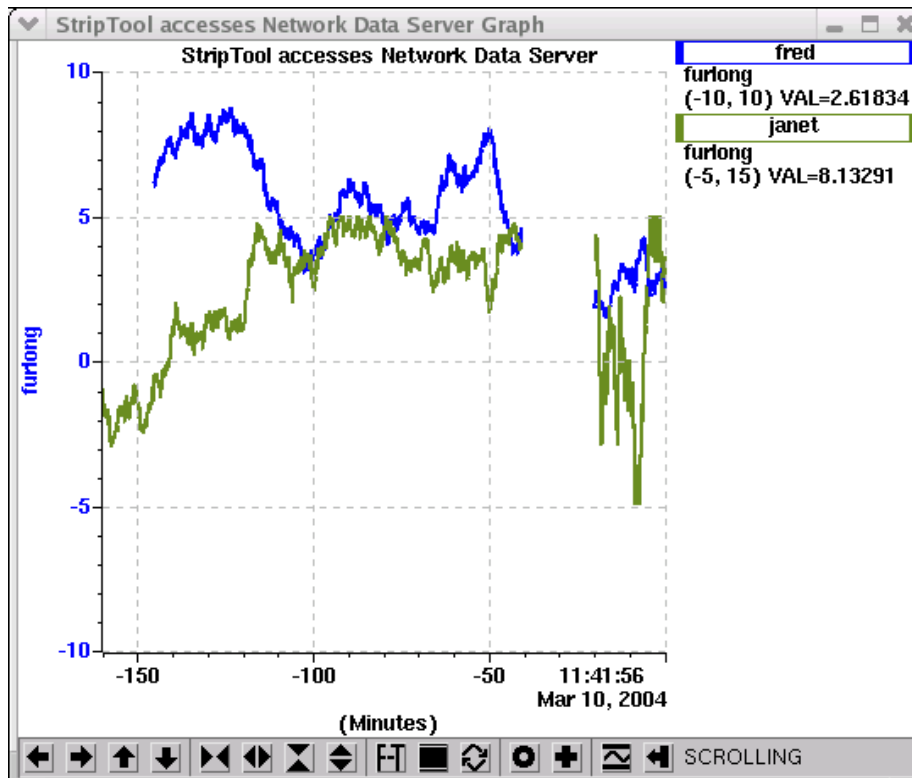


Figure 4.3: StripTool accessing live and archived data.

Fig. 4.3 shows an example of Striptool after running for about 20 minutes. It displays those 20 minutes of live data as well as data retrieved from an archive data server, both types of data clearly separated by a gap of about 20 minutes where the archive was no longer and StripTool not yet running. There will always be a small gap as a result of the ArchiveEngine’s buffering between

writes to the archive.

For configuration details, refer to the file README_XMLRPC which is part of the XML-RPC history module for StripTool.

4.7 Matlab

Programs like Matlab or Octave are ideally suited for the more sophisticated analysis of archived data. The ChannelArchiver includes interface code for Matlab and Octave, allowing those two programs to access data from the ChannelArchiver's Network Data Server. Refer to the file ChannelArchiver/-Matlab/README for details on building, installing and using those extensions. **NOTE:** The Matlab/Octave support is experimental. Its use is discouraged except for testing.

Figures 4.4, 4.5, 4.6, 4.7 and 4.8 showcase some examples and provide you with an excuse to print at least this section of the manual on a color printer.

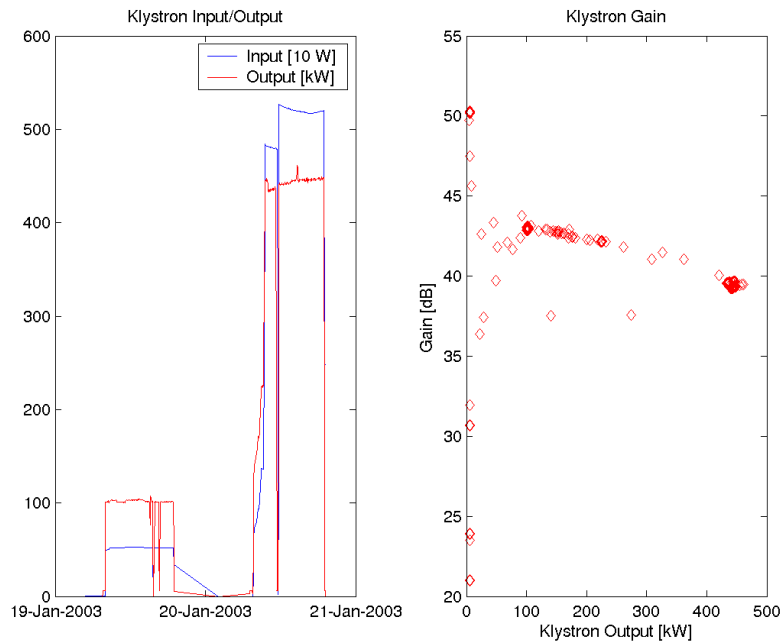


Figure 4.4: Matlab Example: Input and Output power of a Klystron for a two-day test run, combined with a scatter plot of the computed Klystron Gain.

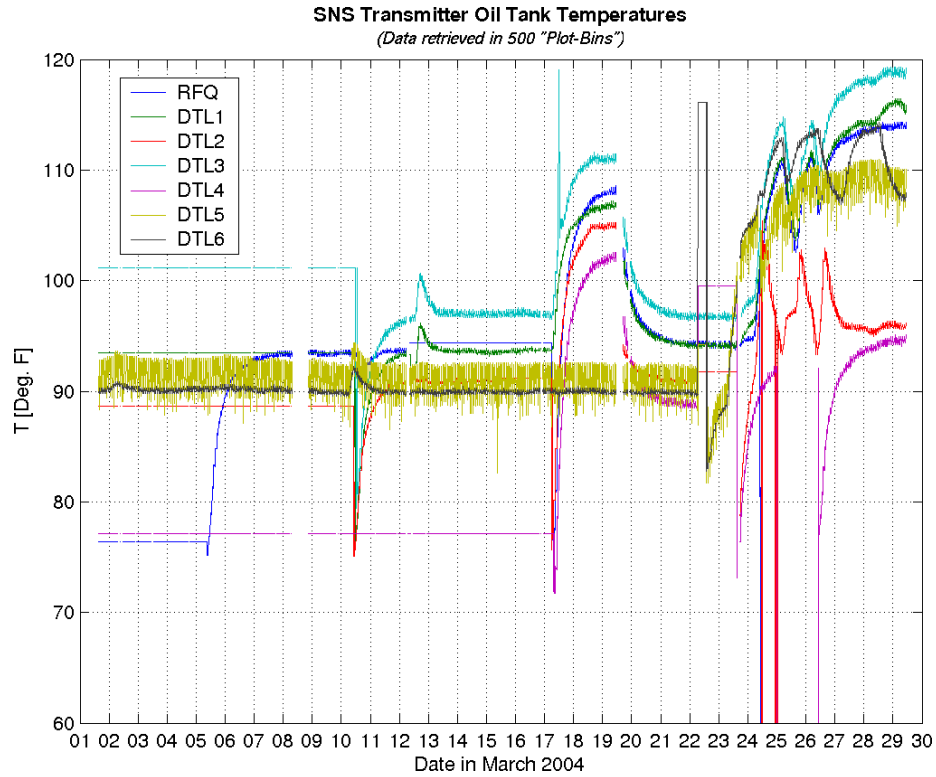


Figure 4.5: Matlab Example: One-month overview of Klystron oil tank temperatures. The Plot-Binning request method as described in section 2.7.5 was used to reduce the amount of data. Interesting features like the noise on the DTL5 signal as well as occasional spikes (which probably result from maintenance work on the oil tank) are preserved.

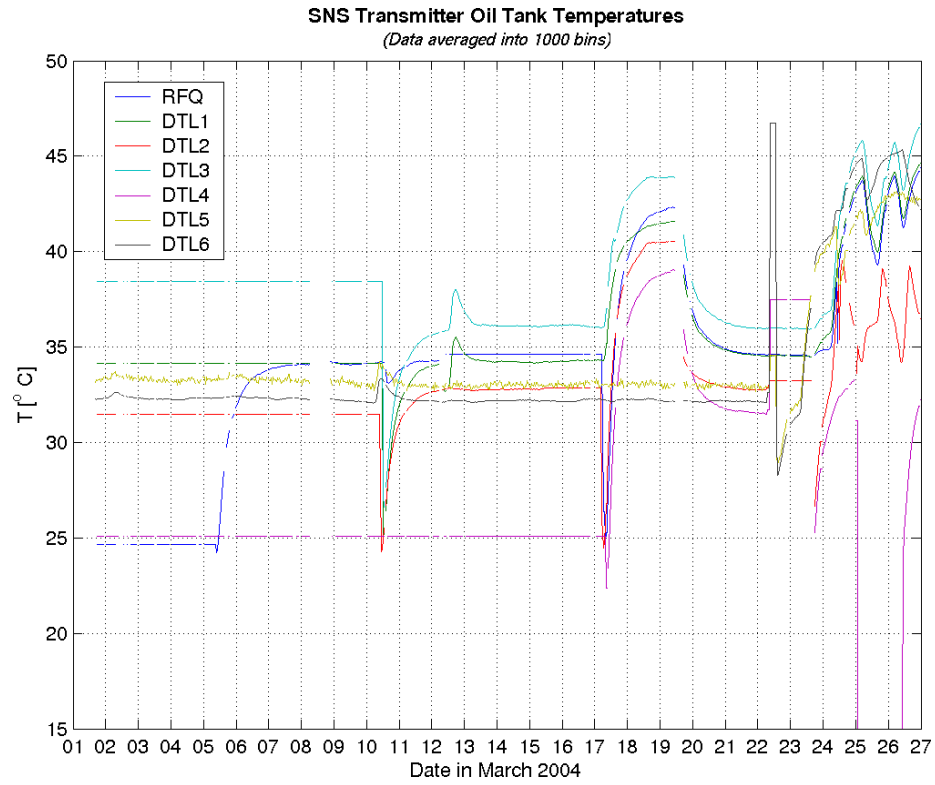


Figure 4.6: Matlab Example: One-month overview of Klystron oil tank temperatures. The raw data was reduced by averaging into 1000 bins as described in section 2.7.4, allowing for easier post-processing, and temperatures were converted to Celsius. Comparison with fig. 4.5 shows how many details can be lost via averaging, so it must be applied with caution.

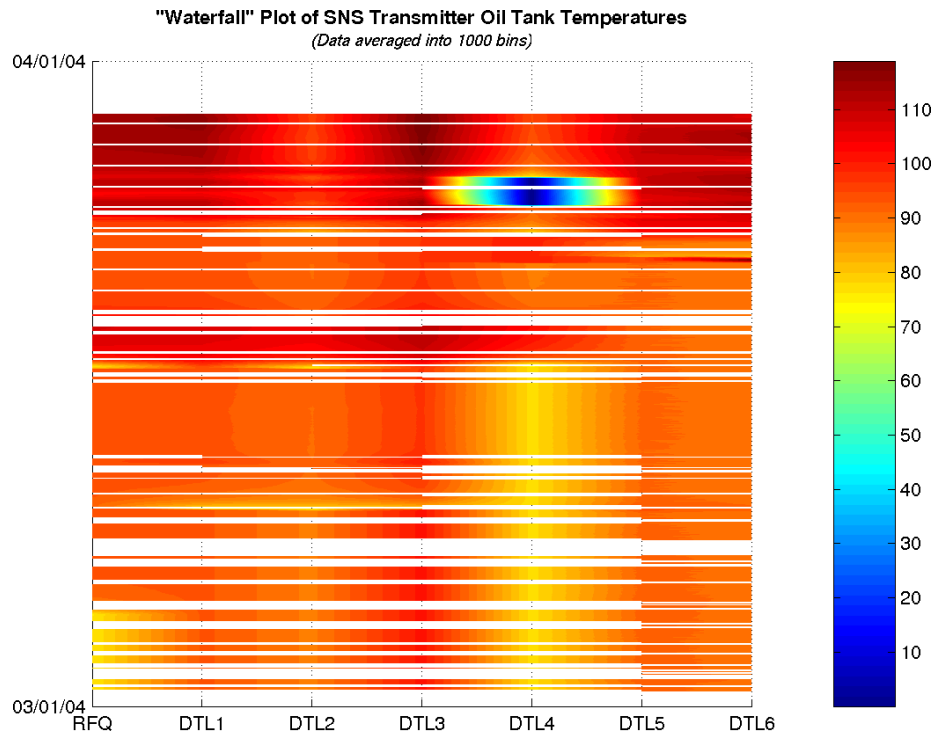


Figure 4.7: Matlab Example: The same data as in 4.6 displayed as a “Waterfall” plot. This type of display hides almost all the detail of the individual channels. Outliers, however, stand out like the possible sensor problem on DTL4, which is why this display method is well suited for an initial investigation of many channels. The example also shows gaps in the data caused by the many times when the archive engine was stopped during the ongoing tests of the new archive engine.

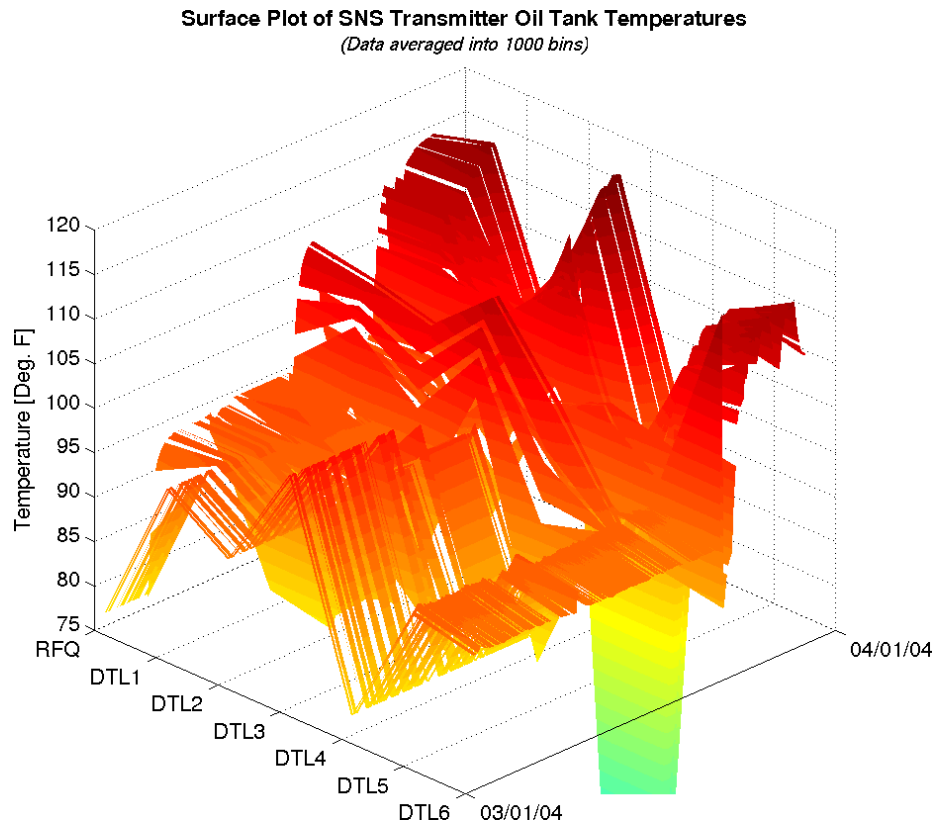


Figure 4.8: Matlab Example: Again the same data as in 4.6 displayed as a surface plot. Tools like Matlab allow the user to rotate this plot in real-time, which might be useful for the inspection of certain channels.

Chapter 5

Data Storage

5.1 Index Files

An index file contains a list of all the channels in an archive, and for each channel it contains information about the data blocks which are available in the Data Files of an archive. The archiver toolset uses index files for two slightly different purposes:

1. Each ArchiveEngine creates an index for the data files that it writes. We refer to this combination of index and data files as a Sub-Archive. If a sub archive contains data for a certain channel and time range, it will contain that data only once.
2. We can create a Master Index that points to data in several sub archives. Several sub-archives might contain data for the same channel and time range. When we combine sub-archives into a master index, we can assign Sub-Archive Priorities to determine what data is considered more important.

Another important difference between sub-archive index files and master index files lies in the fact that the sub-archive index files only the names to their data files: The sub-archive index resides in the same directory as its data files, so a path name is not required to get from the index to the data files. A sub-archive index and its data file can be moved to a new location. As long as the index file and its data files remain together in one directory, the location of that directory does not matter.

The master index file on the other hand contains the path names to its data files, because different sub-archives can use the same data file names within their sub-archive directory. We can only distinguish these data files by their full path. Once a master archive index has been created, the sub-archives must therefore not be moved. After relocating any of the sub-archives, the master index needs to be recreated.

Inside the index file, the channel names are maintained in a hash table and the data block information is kept in a modified RTree structure. An RTree [4] is a balanced tree tailored for holding multidimensional data like rectangles, allowing lookup of rectangles via points that fall inside rectangles. Sergei Chevtsov extended this concept to handle time ranges by requiring that the leaf node entries are non-overlapping and sorted in time.

Each RTree node consists of several records. How many records there are per node is determined by a tunable parameter M : The archiver tools use M as the upper limit of records per node, i.e. a node will simply contain up to M records. (The literature often applies M in a slightly different way, where nodes contain up to $2M - 1$ records.) The records in the leaf nodes of the tree point to data block information (i.e. path to a data file and offset inside that data file) and the time range that is covered by the data block. The records do not overlap, i.e. no two records will cover the same time range, and the records are sorted in time. Since the actual data blocks might overlap (at least for a master index), more than one non-overlapping record might refer to the same data block. Records in parent nodes reflect the time range covered by all their child node records, up to the root node records which hold the total time range covered by all the data blocks.

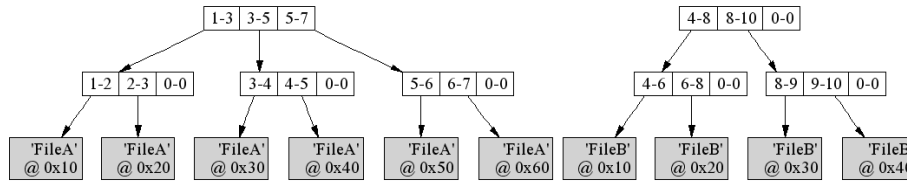


Figure 5.1: RTree Demo, refer to text.

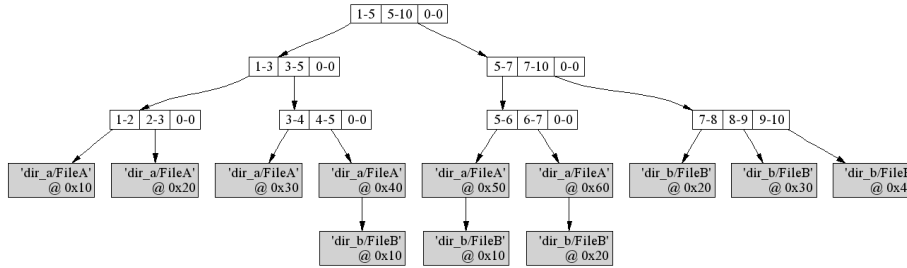


Figure 5.2: RTree Demo, refer to text.

Fig. 5.1 demonstrates two trees with $M = 3$. The one to the left covers the total time range from “1” to “7” (in the real world, these numbers would be much bigger since they represent seconds since some “Epoch”). The data for the time range from e.g. “3” to “4” can be found in data file “FileA” at offset 0x30.

To handle a request for a time range [3;6], we first determine if that range is covered by the tree at all by checking the root's time range. Since that is the case, we go down one level, check the sub-nodes, go down again etc. until we end up at the data blocks.

Fig. 5.2 demonstrates how the two trees from Fig. 5.1 would be combined into a Master Index, assuming that the tree for FileA resides in directory `dir_a` and the data for the second sub-archive resides in `dir_b` relative to the master index. Note how all the data blocks now include a path together with a file name. Since the sub-archive for FileA was listed first in the configuration of the master index, its data blocks take precedence over those from FileB whenever there is an overlap.

The examples used a small number for M so that one can see the tree structure even though we only have a few data blocks. Bigger values of M will reduce the number of read and write operations because the tree is accessed node by node, reading respectively writing all M records of a node in one system call. A big M value will also reduce the height of the tree, as well as the number of nodes and read/write calls. On the other hand, the size of a node obviously grows with M , and the time for reading respectively writing a single node can slightly increase. In general, the number of read/write system calls has a bigger impact on the performance than the size of the individual reads/writes. The records within a node are accessed via a linear search over all the records in a node. A bigger M will require slightly more CPU time for this linear search, which again is negligible compared to the time required for disk access. Overall, a bigger M is likely to increase performance because it reduces the number of disk accesses. The mayor drawback to a big M results from possible fragmentation: If you create many small sub-archives, each tree of the sub-archive will only contain few entries. The minimum size of the tree as well as the size increment whenever a new node needs to be added is determined by M . Big values of M can result in a lot of unused space in the index file, creating unnecessarily big index files. Section 5.5 will present some quantitative details on the performance of the RTree index.

5.1.1 Implementation Details

Table 5.1 shows the basic layout of an index file. The header of the index file contains a 4-ASCII character magic id like 'CAI2' for "Channel Archiver Index Type 2", and the hash table anchor. Those 12 bytes constitute "reserved space" for the FileAllocator class. What follows is start- and end pointers for the FileAllocator's list of allocated and available items, because the remaining file space is handled by the FileAllocator class. The first allocated region is the NameHash, so it's start location would be known. Each hash table entry points to the start of channel entries that hashed to the respective value, and each channel entry contains the anchor for its RTree. The "RTree pointer" in the Hash Entry is actually a file name and an offset. Initially, that file name is empty, because all RTrees are in the same index file that contains the hash

Offset	Content
0x0000	'CAI2'
0x0004	NameHash anchor: start (0x30), size N
0x000C	FileAllocator used list: size, start (0x24), end
0x0018	FileAllocator free list: size, start, end
0x0024	FileAllocator header: size, prev(0), next(??)
0x0030	Hash Table Entry 0: Pointer to first entry for this hash value
0x0034	Hash Table Entry 1:
...	...
??????	Hash Table Entry N-1:
...	...
??????	FileAllocator header: size, prev(0), next(??)
??????	Hash Entry: next, RTree pointer, channel name
...	...
??????	FileAllocator header: size, prev(0), next(??)
??????	RTree: pointer to root node, M value
...	...

Table 5.1: Index file: Example layout.

table. Eventually, that index file might get too big, so the file format already allows for RTree entries to point to other index files. Like every file block after the “reserved space”, the hash table and each channel entry are preceded by a FileAllocator header.

An RTree entry consists of the pointer to the root node and a number of records per node M . The RTree nodes are interlinked as shown in the example in Fig. 5.1 and 5.2, where each node and data block is allocated from the FileAllocator class. For details of how the nodes and data blocks are written to the disk, please refer to the source code.

5.2 Data Files

The data files store the actual data, that is the time stamps, values and the meta information like display limits, alarm limits and engineering units. The archiver stores data for many channels in the same data file. There aren't separate data files per channel because that would produce too many files and slow the archiver down. The names of the data files look like time stamps. They are somewhat related to the time stamps of the samples in there: The name reflects when the data file was created. We then continue to add samples until the engine decides to create a new data file. This means that a data file with a name similar to yesterday's date can still be filled today.

Conclusion 1: Ignore the names of the data files, they don't tell you anything of use about the time range of samples inside.

5.2.1 Implementation Details

Offset	Content
0x0000	'ADF1' (†)
...	...
	'INFO' (†)
0x1000	Numeric CtrlInfo display limits, units, ...
...	...
	'DATA', channel name (†)
0x2000	Data Header prev buffer: "", 0 next buffer: "X", 0x4000 CtrlInfo: 0x1000 dbr_type: dbr_time_double buffer size, amount used, ... Buffer: dbr_time_double, dbr_time_double, ...
...	...
	'DATA', channel name (†)
0x4000	Data Header prev buffer: "X", 0x2000 next buffer: "Y", 0x4000 CtrlInfo: 0x1000 ... Buffer: dbr_time_double, dbr_time_double, ...
...	...

Table 5.2: Data file: Example layout for a data file "X".

Table 5.2 shows the basic layout of a data file "X". Most important, the data file only stores data. It does not need to know about the channel names to which the data belongs. The index would for example tell us that the data of interest for channel "fred" can be found in data file "X" at offset 0x2000. In there, the Data Header points to the preceding buffer (none in this case) and the following buffer (in this case: same file, offset 0x4000). It also provides the data type, size and number of samples to be found in the Data Buffer which immediately follows the Data Header.

Conclusion 2: A data file is nearly useless without the accompanying index file, so you should not separate them.

Beginning with version 2-1-1 of the ChannelArchiver toolset, the items marked with "†" in Table 5.2 were added. A 4-ASCII character magic id at offset 0x0000

identifies the new file type, where each CtrlInfo is preceded by 'INFO' and each Data Header is preceded by 'DATA' followed by the '0'-terminated channel name. One could write a rescue program that e.g. creates an index for data files by searching for the 'INFO' and 'DATA' tags. The Data Buffer contains the raw dbr_time_xxx-type values as received from ChannelAccess. The meta information, that is: limits, engineering units or for enumerated channels the enumeration strings, are stored in a CtrlInfo block. Each Data Header contains a link to a CtrlInfo block, in this case one at offset 0x1000 which happens to contain numeric control information. Each buffer contains a certain number of samples. Whenever a buffer is full, a new one is added. The new buffer might be created at the end of the same data file, but the engine might also create a new data file after a certain time or whenever a data file gets too big. In the example from Table 5.2, the first buffer at offset 0x2000 links to a next buffer at offset 0x4000 in the same file "X", and that buffer in turn points to another buffer in a different file "Y". Note that both the buffer at offset 0x2000 and the one at offset 0x4000 share the same meta information at offset 0x1000, probably because the meta information has not changed.

Conclusion 3: Do not delete individual data files, because this will break the links between data files and result in lost samples. Do not remove the index file. All the data files that were created in one directory together with an index file need to stay together. You can move the index and all data files into a different directory, but you must not remove or rename any single data file.

5.3 Index Tool

The ArchiveIndexTool is used to create Master Indices by combining multiple indices into a new one. When invoked without valid arguments, it will display a command description similar to this:

```
USAGE: ArchiveIndexTool [Options] <archive list file> \
                                     <output index>
```

Options :

```
-help           Show Help
-M <3-100>      RTree M value
-verbose <level> Show more info
```

The archive list file lists all the sub archives, that is the paths to each sub-archive's index file. It needs to be an XML file conforming to the DTD in listing 5.1 (see section 8.2.1 on DTD file installation). Listing 5.2 provides an example.

We refer to chapter 6 for an example of how to use the ArchiveIndexTool in collaboration with the other Channel Archiver tools.

5.3.1 make_indexconfig.pl

As an aid to creating configuration files for the ArchiveIndexTool, you can use the perl script "make_indexconfig.pl" that converts a list of index files into the

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for the ArchiveIndexTool Configuration -->
<!-- Lists all the indices of archives that -->
<!-- should be combined into the master index. -->
<!ELEMENT indexconfig (archive*)>
<!ELEMENT archive (index)>
<!ELEMENT index (#PCDATA)><!-- path -->

```

Listing 5.1: XML DTD for the Archive Index Tool Configuration

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE indexconfig SYSTEM "indexconfig.dtd">
<indexconfig>
  <archive>
    <index>/home/kasemir/xmtrdata/2002/index</index>
  </archive>
  <archive>
    <index>/home/kasemir/xmtrdata/2003/index</index>
  </archive>
</indexconfig>

```

Listing 5.2: Example Archive Index Tool Configuration

appropriately formatted XML:

USAGE: make_indexconfig [-d DTD] index { index }

This tool generates a configuration for the ArchiveIndexTool based on a DTD and a list of index files provided via the command line.

5.3.2 Internals

The Index Tool allows creation of a master index that covers more than one sub archive. For example, we can create configuration files for the index tool to combine several vacuum sub-archives into one, then do the same for cooling data:

```

cd /data/vacuum
make_indexconfig.pl 2004/02_19/index 2004/02_20/index \
  >indexconfig.xml
cd /data/cooling
make_indexconfig.pl 2004/02_19/index 2004/02_20/index \
  > indexconfig.xml

```

After running `ArchiveIndexTool` in `/data/vacuum` and `/data/cooling`, we will have two new indices. One refers to all the vacuum data, the other to all the cooling data:

```
/data/vacuum/index
/data/cooling/index
```

Note that these are only index files. There are no new data files because the new “master” index files will point to data blocks in the existing data files, e.g. the one under `/data/vacuum/2004/02_19`. It is also important to remember that the master index files include the paths to the data files as instructed in the `indexconfig.xml` files. According to the previous example, `/data/vacuum/index` was created from `/data/vacuum/indexconfig.xml` which included the relative path “`2004/02_19/index`”. The vacuum master index will therefore point to data files with a relative path like “`2004/02_19/20040219`”. Whenever we use “`/data/vacuum/index`”, the retrieval tools will prepend the path to the index, “`/data/vacuum`”, to the relative data file path found in the index, for example “`2004/02_19/20040219`”, and thus find the data under its full path of e.g. “`/data/vacuum/2004/02_19/20040219`”. We cannot move “`/data/vacuum/index`” to another location like “`/tmp/index`”. The retrieval tools would then try to access “`/tmp/2004/02_19/20040219`” and fail.

Having said that, it is possible to generate master indices that use the full, absolute paths to their data files by simply listing the full paths to the sub-archives in `indexconfig.xml`. This is, however, not recommended because it will increase the size of the index files simply because the full path names are longer than the relative paths. For the same reason it is advisable to use short path names: When an index file points to many data blocks in many data files, it makes quite some difference if you used a short-named directory tree with paths like “`/data/vac/...`” as opposed to “`/user/data/channel-archiver/data/subsystems/vacuum-system/...`”.

As a second step, we can further combine the master indices for vacuum and cooling data into one index that covers all our data. By creating “`/data/indexconfig.xml`” in which we list “`vacuum/index`” and “`cooling/index`”, and running the `ArchiveIndexTool` in “`/data`”, we create “`/data/index`” which points to all our data. Alternatively, we could have skipped the intermediate indices for vacuum and cooling and created “`/data/indexconfig.xml`” from the beginning like this:

```
cd data
make_indexconfig.pl */2004/* /index \
>indexconfig.xml
ArchiveIndexTool -v1 indexconfig index
```

In any case we end up with “`/data/index`” as an index for all our vacuum and cooling data.

5.4 Data Tool

The ArchiveDataTool allows investigation of data files as well as conversion from old directory-file based archives into ones that utilize an index file.

USAGE: ArchiveDataTool [Options] <index-file>

Options:

-help	Show help
-verbose <level>	Show more info
-list	List channel name info
-dir2index <dir . file>	Convert old directory file to index
-index2dir <dir . file>	Convert index to old directory file
-M <1-100>	RTree M value
-blocks	List data blocks of a channel
-Blocks	List all data blocks
-dotindex <dot filename>	Dump contents of RTree index into dot file
-channel <name>	Channel name
-hashinfo	Show Hash table info
-seek <time>	Perform seek test
-test	Perform consistency tests

5.5 Statistics

It is impossible to provide universal performance numbers for the components of the ChannelArchiver toolset. Tests of a realistic setup are always influenced by network delays: IOCs communicate with ArchiveEngines, data client tools query data from the network data server. And while the archiver tools of course share the CPU with all the other applications that happen to run on the same CPU, the CPU speed is less important. Most crucial is the hard disk performance. Access to data on NFS-mounted disks is by orders of magnitude slower than access to data on local disks. Hard disk access is also hard to reproduce: At least under Linux, the second run of a test is always faster because the operating system caches the disk access. In general, the fewer files and the smaller the involved files are, the better as far as speed is concerned, because the operating system will cache access to files as long as memory allows.

The RTree is a balanced tree. Mathematically, this means that the number of read requests required to locate a node in an RTree depends on the height of the tree, which is again logarithmically related to M and the number of nodes in the tree. An RTree with $M=50$ and height 5 for example has one root node with 50 pointers to sub-nodes, then up to 50^2 nodes on the second level and so on, resulting in access to more than 10^{80} records on the fifth level, that is:

with only 5 reads requests to the disk. In practice, however, there can be a big time difference between 5 read requests to a file of 10 MB total size compared to 5 read requests to a file of 500 MB total size, because the former could be completely buffered by the operating system, while access to the latter will result in individual disk access operations.

The following are performance values obtained on a computer with a 1 GHz CPU, an ordinary IDE disk, that was mostly idle while the archiver tools ran. The corresponding values on a machine with an 800 MHz CPU, concurrently used by other people, but faster hard disks (Mylex DAC960PTL1 PCI RAID Controller with 5 Quantum Atlas 10K drives) were slightly better.

We also provide some comparison to the previous architecture that used the same data file format but instead of the RTree-based index there were “Directory Files”. Instead of being able to combine several sub-archives into one index, one utilized an ASCII “Master File” that simply listed the sub-archives.

5.5.1 Write Performance

As a baseline for raw data writing speed, the ‘bench’ program that can be found in the ChannelArchiver/Engine directory consistently writes at least 80000 values per second on the test computer.

5.5.2 Index Performance

Performance and size of the index depend on the M value configuration of the RTree. Fig. 5.3 displays the file size and the time needed to create an index for a small archive with 5100 channels. The samples occupy 8400 data blocks in a 12 MB data file. The ArchiveIndexTool was used to convert the existing index file of the archive into new indices with different M values.

From the number of channels and data blocks it follows that the samples for most channels occupy only one or two data blocks. Consequently almost all channels can be handled by degenerated RTrees, each with a single node that is both root and leaf of the tree, using only 1 or 2 records in that node. Any records beyond the first few remain unused. Fig. 5.3 clearly indicates how the file size grows linearly with M due to those unused records. The changes in the time needed to create an index can probably be explained as follows: After creating the first new index with $M = 3$, the time dropped observably because the operating system would from now on cache most read requests to the original index. With growing M , the time again increases caused by the growing file sizes of the new indices.

Fig. 5.4 compares the file sizes and creation time over M of a master index that covers 27 sub-archives, a total of 635 MB of data files containing 248261 data blocks for 6164 channels. The smaller archive from the preceding section is actually one sub-archive of this master index. Because some channels have only very few samples, while other channels might have changed every 30 seconds, there cannot be one M value that is ideal for every channel handled

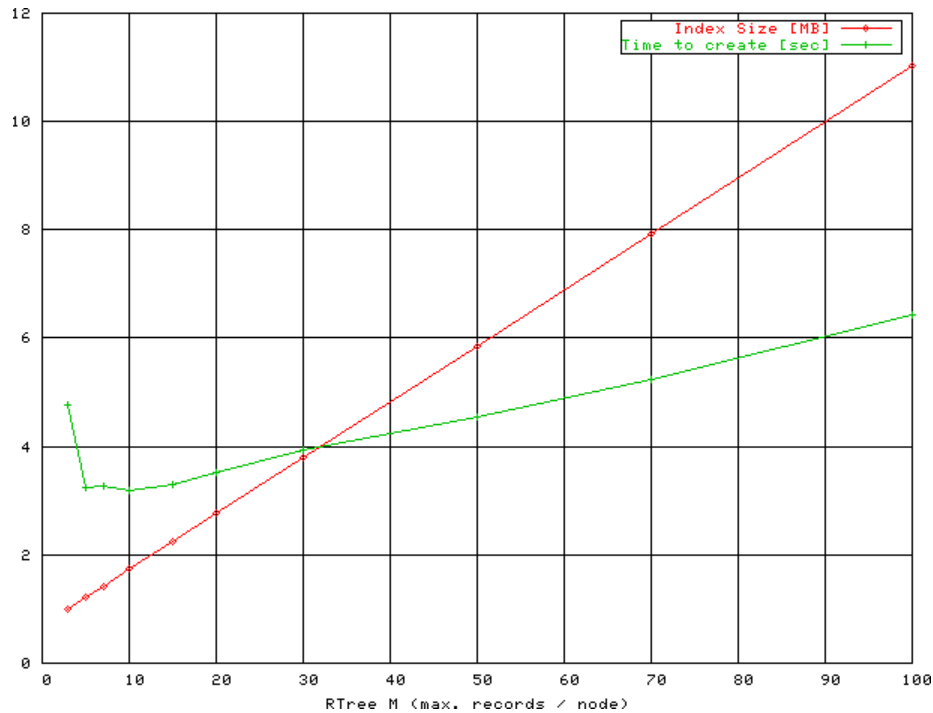


Figure 5.3: RTree M value tuning for small index, see text.

by the master index. By creating the master index with different values of M , we are looking for a compromise that gives best index performance across channels.

Fig. 5.4 shows that values between 10 and 50 result in a smaller master index than M values outside of this range. Remember that for a given height, the number of leaf records in an RTree grows exponentially with M , so slight increases of M beyond 50 will vastly increase the number of leaf records. Archives with twice or ten times the number of data blocks will therefore not require M values that are equally 2 or 10 times bigger. Only very small increases of M would be beneficial. If we consider that in general those bigger archives will also contain channels with only a few samples, $M = 50$ will probably be “as good”.

In the following, M was kept at 50, the default for most archive tools.

- 12 sub-archives, 1.2 MB of old directory files, 1.4 GB Data files:
 Converting directory files into index files with $M = 50$: Just under 3 minutes, resulting in 11 MB for the new index files.
 Creating a master index: 37 seconds for a master index of 9 MB. The master index is slightly smaller than the sum of the individual sub-indices because of better RTree utilization: The M was configured to be 50 in all cases and many channels in the sub-archives use only a fraction of

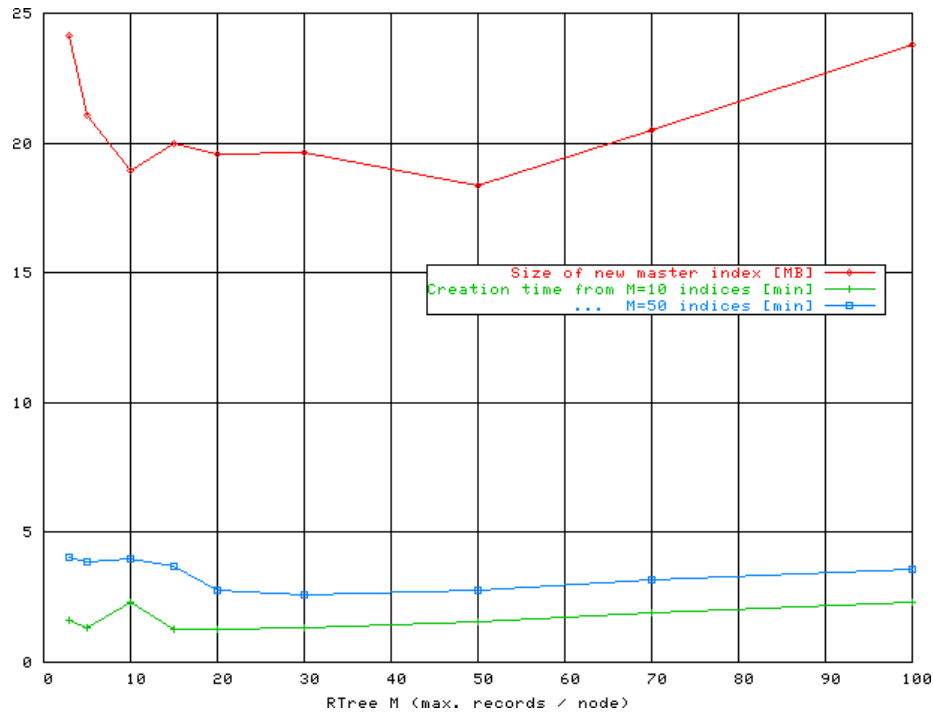


Figure 5.4: RTree M value tuning for master index, see text.

a single RTree node, down to an average record usage of 8%, while the master index uses around 50%. A re-run of the ArchiveIndexTool tool is faster because it detects data block that are already listed in the master index and therefore not added again. In this case, the re-run took 10 seconds.

- 92 sub-archives, 12 MB directory files, 2.3GB of data files:
 Converting into 61 MB of index files: About 12 minutes.
 Creating a master index: Under 2 minutes, the resulting index uses about 18 MB. Re-run: 30 seconds.

5.5.3 Impact of Data Management on Performance

As a less-than-perfect example, we created a collection of mostly hourly sub-archives, resulting in 297 sub-archives, 158 MB index files, 307 MB data files. Creation of a master index took about 25 minutes, resulting in a master index file size of 65 MB. A re-run of the Index Tool took about 3 minutes.

By combining the hourly sub-archives into monthly ones, the count was reduced from 297 sub-archives to only 16. This took about 8 minutes, resulting in 5.5 MB for index files and 148 MB for data files. Creation of a master index

for the 16 sub-archives now took 26 seconds, a re-run was further reduced to 1.5 seconds. Overall this shows that periodic data management, combining individual sub-archives into fewer ones, will reduce not only the number of files but also file sizes, resulting in better performance.

5.5.4 New Index compared to old “multi archive”

We took a test archive consisting of 638 small sub-archives, where the data files totaled only a little over 400 MB. A master index was created as well as a “multi archive” file that lists the 638 sub-archives. Creation of that master index too 15 min.OB

- Time to list all 540 channel names: <1 second.
This takes 40 seconds with the old “multi archive” file. The results for finding names that match a pattern or determining the available time range for a channel are similar.
The new index is clearly superior in this test case simply because the data is contained in one index, while the “multi archive” file required access to all 638 sub-archives.
- The time to retrieve a few samples from the start, middle or end of the archive is fairly constant around 0.1 seconds with the new index. With the previous implementation, the lookup time for the samples depends on where the respective sub-archive is positioned in the multi-archive file. In one test it ranged from 0.1 to 10 seconds. When the data is found in the first few sub-archives listed in the multi-archive file, the times compare to the new index. The further one goes down the list of sub-archives as they appeared in the multi-archive file, the longer it takes. A reproducible test is difficult because the preceding tests (list all channels) causes the operating system to cache many of the sub-archive's directory files.

5.5.5 Retrieval Performance

Tests of the retrieval performance often include not only the code for getting at the data but also for presenting it. In the case of the command-line Archive Export program this would be the process of converting time stamps and values into ASCII text and printing them. In the following tests, the output was redirected to a file.

- Use ArchiveExport to dump all the 143000 values for a channel: 4 seconds, translating into 35700 values per second.
- Use ArchiveExport to dump all the values of one month for 7 channels in a spreadsheet format, which adds the effort for 'staircase' interpolation to the previous test case: Each channel had 20000 to 30000 values. The interpolation generated a 95000 line spreadsheet in 11 secs, that is

around 8600 lines/sec. On a second test run, the time was reduced to 7 seconds, again showing the impact of buffering by the operating system.

- Use Matlab to retrieve the first 500 raw samples of the same 7 channels: The ArchiveServer ran 0.7 seconds, Matlab used a total of 1.7 seconds from sending the request to receiving the data.
- Use Matlab to retrieve data for the same 7 channels, asking the network data server to reduce the raw data (which formed the 95000 line spreadsheet in the previous test) into 500 “Plot Bins”: Around 1500 values per channel were retrieved in 5 seconds. The ArchiveDataServer.cgi ran 3 seconds, so about two seconds were added by the web server, network transfer, Matlab MEX code and Matlab.
- Use the Java Archive Client to retrieve the same 7 channels into 800 “Plot Bins”: Around 2100 values per channel were retrieved and plotted in 12 seconds. (When using the alternate machine with the slower CPU but RAID disks as the data server, the time was reduced to 8 seconds). The Java client was still usable, but slow to respond with this amount of data: Zoom requests took about 2 seconds.

This shows that the initial lookup of a channel and the location of the samples in the data files requires a certain time. Reading the values can then be quite fast and reach more than 30000 values per second when simply fetching the raw samples. Interpolation or binning can internally reach this speed when investigating the raw data,

The network data server typically adds about 1 second of overhead.

Chapter 6

Example Setup

The following describes how the archiver toolset is used at the Spallation Neutron Source (SNS). Other sites might need to modify the tools in the ChannelArchiver/ExampleSetup directory for their need.

For the SNS, every computer involved in archiving has a local directory “/arch”. On some computers, for example ics-srv-archive1, that directory is used to run engines. On another computer, for example ics-srv-web2, that same directory contains data copied from archive1 so that the data server can serve it. The actual data might reside in different partitions, which are accessed via soft-links from /arch. There might be more than one ‘sampling’ computer as well as more than one ‘serving’ computer.

We want to be able to move an engine from one computer to another, and still keep an overview. Therefore a file “/arch/archiveconfig.xml” describes the complete archive setup: Which engines run where, and how the data gets served. The next section describes archiveconfig.xml in a bit more detail. Further sections describe the scripts from ChannelArchiver/ExampleSetup which we typically copy into “/arch/scripts” on all the involved computers.

6.1 archiveconfig.xml

Each computer needs to have the same copy of /arch/archiveconfig.xml. You might generate and distribute that file manually or use a relational database. People who have used previous releases of the archive toolset might remember the archiveconfig.csv file. There is a tool `convert_archiveconfig_to_xml.pl` to convert that file into an archiveconfig.xml skeleton.

In any case, archiveconfig.xml describes the complete archive layout, using the following elements:

- root: Names the root directory, typically ‘/arch’. This has to be the same directory name on all computers, since they all use the same archiveconfig.xml.

- mailbox: Used to communicate from the ArchiveDaemons to the data server. Has to be an existing directory where all involved computers can read and write, e.g. shared via NFS.

update_archive_tree.pl

6.2 ArchiveDaemon

ArchiveDaemon.dtd ArchiveDaemon.pl ArchiveDaemon.xml
start_daemons.pl stop_daemons.pl

6.2.1 Status Information

engine_write_durations.pl make_archive_infofile.pl make_archive_web.pl
show_engines.pl show_sizes.pl

6.3 Indices

update_indices.pl

6.4 Data Server

update_server.pl

ALL THE REST NEEDS TO BE REVIEWED!

A file "archiveconfig.csv" describes the top-level layout:

```
# Archive configuration file
# This is TAB delimited!
# Careful when editing , do not disturb the TABs,
# because this file fails to parse unless
# everything is perfect.
```

#	Type	Name	Port	Desc	Restart	Time
	DAEMON	ICS	4090	ICS Daemon		
	ENGINE	tim	4091	Timing	daily	08:00
	ENGINE	mps	4092	Mach. Prot.	daily	08:30

	DAEMON	RF	4900	RF		
	ENGINE	llrf	4901	LowLvl.	weekly	We 10:20
	ENGINE	hprf	4902	HiPwr.	weekly	We 10:30

This reads as follows: We intend to run one "ArchiveDaemon" for the Integrated Control System (ICS) and one for the channels related to Radio Frequency

(RF). ArchiveDaemon, described in full detail in chapter 7, starts archive engines and monitors their operation. The ICS daemon should maintain one ArchiveEngine for the timing system (tim) and one for the machine protection system (mps), while the RF daemon has one engine for the low-level and one for the high power RF (llrf, hprf).

This separation is somewhat arbitrary. We could have made “llrf” and “hprf” channel groups under one and the same engine. In fact all the above could reside within one engine. It is, however, advisable to spread the channels over different daemons and engines whenever different people deal with the IOCs that host the channels, so that the engineers can independently configure their archiving. In addition, you want to keep the amount of data collected by each engine within certain bounds, for example: not more than one CD ROM per month, one DVD per year, or whatever you plan to do for data maintenance. Another reason is data safety: You can reduce the damage caused by crashes of the engine by limiting the number of channels per engine.

6.5 Current Archive Status

The perl script “make_archive_web.pl” creates a web page displaying the current status of all the archive daemons and engines that are listed in the file “archiveconfig.csv” (you could use other names for the main configuration file, but we will stick with “archiveconfig.csv” in this manual). For the SNS, this script is periodically executed with the result being redirected into the web server’s document tree, so you can usually reach the recent status from this web page:

```
http : //ics-srv-archive1/archive
```

Fig. 6.1 shows one example of that web page: A tabular display of what engines are running, how many channels are connected, and more.

6.6 Directory Layout

The perl script “make_archive_dirs.pl” creates or updates a directory tree based on the file “archiveconfig.csv”. For the preceding example, it would create sub-directories “ICS” and “RF” for the two daemons, plus the following engine directories:

```
ICS/tim
ICS/mps
RF/llrf
RF/hprf
```

Each daemon directory contains a daemon configuration file as well as start/stop scripts. Each engine directory contains a script to stop the engine. Note the absence of a script to start the engine: You should not manually start engines

DAEMON	ENGINE	PORT	DESCRIPTION	STATUS	RESTART	TIME
ICS		4090	ICS Daemon	2 of 2 engines are running 6052 of 6578 channels connected		
	tim	4091	ICS Timing Engine		daily	08:00
	mps	4092	ICS MPS Engine		daily	08:30
RF		4900	RF	2 of 2 engines are running 1401 of 1401 channels connected		
	llrf	4901	LLRF		weekly	We 10:20
	hprf	4902	HPRF			

Figure 6.1: Example of the archive status web page generated by the `make_archive_web.pl` script.

which are under the control of an archive daemon. Each engine directory also contains an “ASCIIConfig” subdirectory with a script “convert_example.sh” that you might use to create the XML configuration file for the ArchiveEngine from ASCII configuration files, though the engineer responsible for the subsystem is free to use any method of his/her choice as long as the result is a configuration file for the engine that follows the naming convention

Daemon-Name / Engine-Name / Engine-Name-group.xml ,

resulting in these for our example:

```
ICS / tim / tim-group . xml
ICS / mps / mps-group . xml
RF / llrf / llrf-group . xml
RF / hprf / hprf-group . xml
```

As long as you end up with engine configuration files of these names, you can employ any text editor, a copy of the example script, or a sophisticated toolset utilizing a relational database. If you want to use the ASCIIConfig directory, check section 10.3 for the format of the ASCII configuration files.

6.7 Sub-Archives

Based on the configuration from the beginning of this chapter, the daemon will

1. Start an ArchiveEngine in “RF/llrf” that writes to a sub-archive named after the current day, e.g. “RF/llrf/2004/02_11/index”. Same for a second engine in “RF/hprf”.
2. Periodically verify if engines that are supposed to run are actually running, attempting to start engines which are found missing.
3. Stop the ArchiveEngines each Wednesday at 10:20 respectively 10:30 and restart them in new subdirectories (using the data of the restart for the path to the index file).
4. Generate or update “RF/indexconfig.xml” whenever a new sub-archive is created for the vacuum or cooling data.
5. Periodically run ArchiveIndexTool on “RF/indexconfig.xml”, generating or updating “RF/master_index”.
6. Provide a web page that lists the status of the two archive engines.

As a result, we create separate sub-archives for the LLRF and HPRF, a new one once per week, which provides some insurance against crashes of an ArchiveEngine. If you are paranoid, you can choose daily sub-archives; compare the ICS setup from the beginning of chapter 6. After running for a while, we will have created sub-archives like these:

```
RF/llrf/2004/02_11/index
RF/llrf/2004/02_18/index
RF/llrf/2004/02_25/index
...
RF/hprf/2004/02_11/index
RF/hprf/2004/02_18/index
RF/hprf/2004/02_25/index
...
```

In addition to each index file, there will of course also be associated data files, but for retrieval purposes we identify an archive solely by its index file: We can invoke e.g. ArchiveExport with the path to any of the index files. This is, however, inconvenient because we will only see data for one week of that one subsystem at a time. The periodic invocation of the ArchiveIndexTool allows us to view all the RF data as a whole via the RF/master_index.

6.7.1 ArchiveDaemon Details

The “Restart” and “Time” columns of the “archiveconfig.csv” file are passed on to the ArchiveDaemon, which is explained in more detail in chapter 7. In many cases it might be sufficient to know these two options:

- Restart=“daily”, Time set to the time of day in 24-hour format HH:MM, e.g. 08:00 for 8’o clock in the morning. The daemon will stop and restart the engine in a new sub-archive each day at the given time.

- Restart="weekly", Time set to a string that combines the day of the week (Mo, Tu, We, Th, Fr, Sa, Su) with the time of day in 24-hour format into "DD HH:MM", e.g. "We 08:00" for Wednesdays, 8'o clock in the morning. Similar to the daily setup, but reduced to once a week.

It is advisable to stagger the restart times of your engines such that they don't all restart at the same day and time in order to reduce the CPU and network load for the ChannelAccess re-connects.

6.8 Common Tasks

6.8.1 Check Daemon, Engine, Connected Channels, ...

See section 6.5. That web page links to all the ArchiveDaemons, which in turn link to all the ArchiveEngines.

6.8.2 Modify Engine's Request Files

Locate your archive engine on either the main archive status page (section 6.5) or in /arch/archiveconfig.csv. According to the example at the beginning of this chapter, the "High Power RF" engine would be run by the "RF" daemon and be called "hprf", so we need to modify /arch/RF/hprf/hprf-group.xml. This is often done via a conversion script in /arch/RF/hprf/ASCIIConfig. If you used another method to create the engine configuration, this is a good time to remember what you did.

Then, to actually use that new config file, the engine needs to restart. We could simply wait for the next scheduled restart, in our example the next Wednesday, 10:30. Alternatively, we can run the script /arch/RF/hprf/stop-engine.sh. Watch the RF daemon via the link on the main archive status page. Within a few minutes, it ought to detect that the engine had stopped and then restart it.

6.8.3 Add Engine or Daemon

Add a line to archiveconfig.csv to define the new engine under an existing daemon. Or add a line for a new daemon, then add the new engine under it. Invoke make_archive_dirs.pl. Per default, it will re-create all daemon and engine directories, so you might want to use the "-s" option to limit its operation to the new or modified subsystem. In case the daemon was already running, it won't learn about the new engine unless you restart it. So run the "stop-daemon.sh" and then invoke the "run-daemon.sh" script in the daemon directory to start the daemon (which will then start any missing engines).

6.8.4 An Engine isn't running

Check the process list to assert that the engine in question is really not running (on UNIX, try “ps -aux”). Look again. If the engine is actually running but not responding via its HTTPD, remove the process. Check the log file of the engine, generated in the engine subdirectory, for any clues. If you are convinced that the engine is not running, but find an “archive.active.lock” lock file in the engine directory, remove it. Now the daemon should be able to start your engine.

6.8.5 I want to stop a Daemon

Run stop-daemon.sh in the daemon directory, or check chapter 7 for more on the daemon's HTTPD.

6.8.6 A Daemon isn't running

Run start-daemon.sh in the daemon directory. If the daemon keeps quitting, check its log file for clues.

6.8.7 Re-building a Master Index

Whenever you add or remove a sub-archive, the master index in the daemon directory could be obsolete: It might still list data in a sub-archive that you removed, or it might not yet include the new sub-archive. Another szenario: you suspect that the master index is broken, because you can retrieve data from the individual sub-archive but not via the master index. The recipe:

- Stop the daemon
- Check, maybe rebuild indexconfig.xml, either manually or by using the helper script make_indexconfig.pl (see section 5.3.1).
- Start the daemon, which causes it to invoke the ArchiveIndexTool.

6.9 Data Management

The generation of daily sub-archives reduces the amount of data that might be lost in case an ArchiveEngine crashes and cannot be restarted by the ArchiveDaemon to one day. In the long run, however, it is advisable to combine the daily sub-archives into bigger ones, for example monthly. The smaller number of sub-archives is easier to handle when it comes to backups. Is also provides slightly better retrieval times. Depending on your situation, monthly archives might either be too big to fit on a CD-ROM or ridiculously small, in which case you should try weekly, bi-weekly, quarterly or other types of sub-archives.

In the following example, we assume that it's March 2004 and we want to combine the two daily vacuum sub-archives from the previous section into one for the month of February 2004:

```
cd vacuum/2004
mkdir 02_xx
ArchiveDataTool -copy 02_xx/index 02_19/index \
    -e "02/20/2004_02:00:00"
ArchiveDataTool -copy 02_xx/index 02_20/index \
    -s "02/20/2004_02:00:00" -e "02/21/2004_02:00:00"
```

Note that we assume a daily restart at 02:00 and thus we force the ArchiveDataTool to only copy values from the time range where we expect the sub-archives to have data. This practice somewhat helps us to remove samples with wrong time stamps that result from Channel Access servers with ill-configured clocks.

There is a perl command `make_compress_script.pl` that aids in the creation of a shell script for the ArchiveDataTool, but you need to review it carefully before invocation. After successfully combining the daily sub-archives for February 2004 into a monthly 2004/02_xx, we need to

1. Stop the ArchiveDaemon because we are about to edit `indexconfig.xml`. The ArchiveEngines controlled by the daemon can run on.
2. Edit `indexconfig.xml` that listed the daily sub-archives for Feb. 2004 and replace them with the single 2004/02_xx/index.
3. Remove or rename the master index file and re-create it with the new `indexconfig.xml`. This step is required because the ArchiveIndexTool will only add new data blocks to the master index, it will not remove existing ones. Since we no longer want to refer to the daily sub-archives, we need to recreate the master index.
4. Start the ArchiveDaemon again, check its online status.
5. One may now move the daily sub-archives that are no longer required to some temporary location. A month later, when we are convinced that nobody is still trying to use them, we can delete them.

Again there is no tool available to automate the `indexconfig.xml` update.

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE archiveconfig SYSTEM "archiveconfig.dtd">
<archiveconfig>
  <root>/arch</root>

  <serverconfig>/var/www/html/archive/cgi/serverconfig.xml</serverconfig>

  <mailbox>/arch/xfer</mailbox>

  <daemon directory='RF'>
    <run>archive1</run>
    <desc>RF</desc>
    <port>4900</port>
    <dataserver>
      <index type='binary' key='10'>RF</index>
      <host>web2</host>
    </dataserver>
    <engine directory='llrf'>
      <run>archive1</run>
      <desc>LLRF</desc>
      <port>4901</port>
      <restart type='weekly'>We 10:20</restart>
      <dataserver>
        <current_index key='4901'>RF llrf (last restart)</current_index>
        <index type='binary'>LLRF data</index>
        <host>web2</host>
      </dataserver>
    </engine>
  </daemon>
</archiveconfig>

```

Listing 6.1: XML DTD for the Data Server Configuration

Chapter 7

ArchiveDaemon

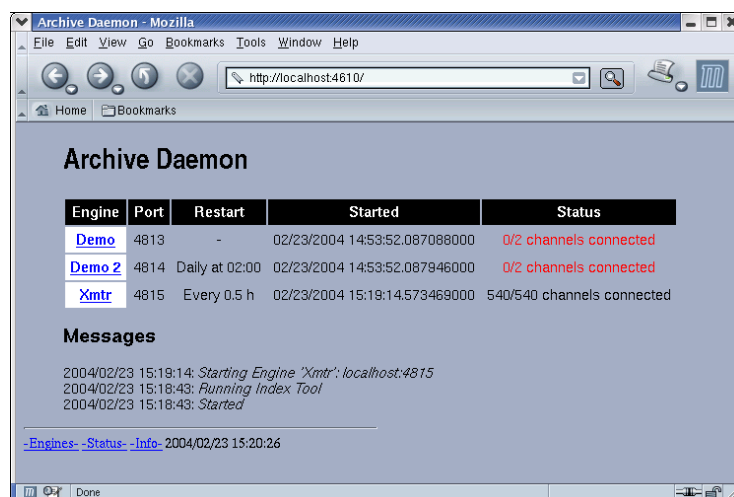


Figure 7.1: Archive Daemon, refer to text.

The ArchiveDaemon is a script that automatically starts, monitors and restarts ArchiveEngines on the local host. It includes a built-in web server, so by listing all the ArchiveEngines that are meant to run on a host in the ArchiveDaemon's configuration file, one can check the status of all these engines on a single web page as shown in Fig. 7.1.

Chapter 6, "Example Setup", on page 64, gives more details on the suggested use of the ArchiveDaemon. The daemon will attempt to start any ArchiveEngine that it does not find running. In addition, the daemon can periodically stop and restart ArchiveEngines in order to create e.g. daily sub-archives. Furthermore, it adds information about each sub-archive of newly created ArchiveEngines to a mailbox directory so that the index mechanism can create the necessary indices and update the data server configuration.

Before using the ArchiveDaemon, one should be familiar with the configuration of the ArchiveEngine (sec. 3), and how to start and stop it. Furthermore, one needs to be familiar with the ArchiveIndexTool (sec. 5.3).

7.0.1 Configuration

The ArchiveDaemon expects to find a configuration file called “ArchiveDaemon.xml” in the directory where it is started. That configuration file, an example of which can be found in listing 7.1, needs to follow the DTD from listing 7.2.

In many cases you might not configure the daemon yourself, but instead use the mechanism described in chapter 6.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE engines SYSTEM "ArchiveDaemon.dtd">
<engines>
  <engine>
    <desc>Demo</desc>
    <port>4813</port>
    <config>/arch/RF/llrf/llrf.xml</config>
  </engine>
  <engine>
    <desc>Demo 2</desc>
    <port>4814</port>
    <config>/arch/RF/hprf/hprf.xml</config>
    <daily>02:00</daily>
    <copy>dataserver:/arch/RF/hprf</copy>
  </engine>
</engines>
```

Listing 7.1: Example Archive Daemon Configuration

The configuration lists all the ArchiveEngines that the daemon should manage on the local computer. One “engine” element per ArchiveEngine specifies the configuration of each engines. Specifically, the following tags are allowed:

desc

This mandatory element is used for the “-description” option of the Archive Engine, see section 3.2.3.

port

This mandatory element determines the port number of the engine’s HTTP server, see section 3.2.4.


```

<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for the ArchiveDaemon Configuration -->
<!ELEMENT daemon (port , mailbox? , engine*)>

<!ELEMENT port    (#PCDATA)> <!-- TCP port number    -->
<!ELEMENT mailbox (#PCDATA)> <!-- /path              -->

<!ELEMENT engine (desc , port , config , restart? , dataserver?)>
<!ATTLIST engine directory CDATA #REQUIRED>

<!ELEMENT desc    (#PCDATA)> <!-- Text                -->
<!ELEMENT config  (#PCDATA)> <!-- path                -->

<!ELEMENT restart (#PCDATA)> <!-- path                -->
<!ATTLIST restart type (weekly|daily|hourly|timed) #REQUIRED>
<!-- weekly  Mo|Tu|...|Su HH:MM
      daily   HH:MM
      hourly  (double) hours
      timed   HH:MM/HH:MM
              (start/duration)
-->

<!ELEMENT dataserver (host?)>
<!ELEMENT host (#PCDATA)>

```

Listing 7.2: XML DTD for the Archive Daemon Configuration

NOTE: The ArchiveDaemon itself requires a TCP port number for its HTTP server. The port numbers used by the ArchiveDaemon and all the Archive Engines need to be different. You cannot use the same port number more than once per computer.

config

This mandatory element must contain the path to the configuration file of the respective ArchiveEngine, see section 3.1. This can be either the full path to the engine configuration file or a path beneath the current working directory in which the ArchiveDaemon is running.

daily

This optional element configures the ArchiveDaemon to restart the ArchiveEngine periodically. The element must contain a time in the format “HH:MM” with 24-hour hours HH and minutes MM. One example would be “02:00” for a restart at 2 am each morning.

weekly

Weekly is similar to daily, but using an element that contains the day of the week (Mo, Tu, We, Th, Fr, Sa, Su) in addition to the time on that day in 24-hour format, e.g. “We 08:00”. In this example, the daemon will attempt a restart every Wednesday, 8’o clock in the morning.

timed

In this case, the element needs to contain a start/duration time pair in the format “HH:MM/HH:MM”. The first, pre-slash 24-hour time stamp indicates the start time, and the second 24-hour time, trailing the slash, specifies the runtime. The engine will be launched at the requested start time and run for the duration of the runtime. As an example, “08:00/01:00” requests that the daemon starts the engine at 08:00 and stops it after one hour, probably around 09:00.

hourly

This optional element configures the ArchiveDaemon to restart the ArchiveEngine periodically. The element must contain a number specifying hours: A value of 2.0 will cause a restart every 2 hours. The hourly restart is quite inefficient and primarily meant for testing.

7.0.2 Starting and Running

The ArchiveDaemon is a perl script that is typically started like this:

```
$ cd wherever_you_placed_ArchiveDaemon.xml
$ perl ArchiveDaemon.pl -f ArchiveDaemon.xml
Read ArchiveDaemon.xml, will disassociate from terminal
and from now on only respond via
    http://localhost:4610
You can also monitor the log file:
    ArchiveDaemon.log
```

One can use any web browser to connect to the daemon’s HTTP server under the URL shown in the above status message. Fig. 7.1 shows one example. The ArchiveDaemon offers a command line option for selecting a specific TCP port. Whenever running more than one ArchiveDaemon per computer, they need to be started with different TCP port numbers. Furthermore, each ArchiveEngine needs a different TCP port number.

```
USAGE: ArchiveDaemon [ options ]
Options:
  -p <port>      : TCP port number for HTTPD
  -f <file>       : config. file
  -i <URL>        : path or URL to indexconfig.dtd
```

```

-u <minutes> : Update period for master index
-d           : debug mode (stay in foreground etc.)

```

The first few lines of the ArchiveDaemon.pl script contain numerous configuration variables. They allow fine tuning of e.g. how often the daemon queries the Archive Engines and other customization options. In there you can also change many of the file names from their defaults up to the point where none of the following applies. With the original settings, the ArchiveDaemon will create or use the following files in the directory in which it was started:

- ArchiveDaemon.log
The log file of the ArchiveDaemon.
- indexconfig.xml
This is a configuration file for the ArchiveIndexTool. If the file already exists, the ArchiveDaemon will add every new sub-archive that it creates to the file. If the file does not exist, one will be created the next time a new sub-archive is started.

Note that the ArchiveDaemon will *not* search for sub archives or index files and automatically add them to indexconfig.xml. It will only add newly created sub-archives. If you already have sub-archives that need to be included in the master index, your initial indexconfig.xml needs to list them. Also note that the ArchiveDaemon reads this file on startup. In order to *remove* indices from indexconfig.xml, it is therefore required to stop the running daemon, since it will otherwise overwrite indexconfig.xml with its in-memory version.
- indexupdate.xml
This file is similar to indexconfig.xml, except that the index files for sub-archives that are older than the master index file are commented out, assuming that the master index already contains all the information from those older sub-archives. Running ArchiveIndexTool with this indexupdate file is naturally faster than using indexconfig.xml. The ArchiveDaemon uses the full index configuration from indexconfig.xml once after startup, and then switches to indexupdate.xml.
- ArchiveIndexTool.log
The log file of the last run of the ArchiveIndexTool.
- master_index
The ArchiveIndexTool is run with indexconfig.xml to update this master index file.

The ArchiveDaemon configuration file must list the full path names to the configuration files for the ArchiveEngines or use a path that is below the current working directory of the ArchiveDaemon. Within each of those directories, an ArchiveEngine is run and the following files will be created:

- ArchiveEngine.log
A log file for the ArchiveEngine running in that directory
- archive_active.lock
Lock file of the ArchiveEngine
- YYYY/MM_DD/index
A subdirectory for index and data files of the sub-archive. If the ArchiveDaemon is configured to perform daily restarts, the format uses the year, month and day to build the path name.

7.0.3 Daemon's Web Pages

The main web page of the ArchiveDaemon's HTTPD looks similar to Fig. 7.1. You can use any web browser to look at the daemon's web pages. The URL follows the format

```
http://host:port
```

where "host" is the name of the computer where the ArchiveDaemon is running. More often than not you will use "localhost". "Port" is the TCP port that was specified as a command-line option to the ArchiveDaemon program, otherwise it defaults to 4610. So the default URL will be http://localhost:4610.

The main page lists all the archive engines that this daemon controls with their status. The first column also contains links to the individual archive engines. The status shows any of the following:

- "N/M channels connected"
This means the ArchiveEngine is running and responding, telling us that N out of a total of M channels have connected. If not all channels could connect, you might want to follow the link to the individual engine to determine what channels are missing and why: Is an IOC down on purpose? Is an IOC disconnected because of network problems? Does a channel simply not exist, i.e. the engine's configuration is wrong?
- "Not Running"
This means that the respective ArchiveEngine did not respond when we queried it, and there is no "archive_active.lock" lock file. This combination usually means that the engine is really not running (except for the Note below).

The first step in debugging would be to check the engine's directory for a log file. Does it indicate why the engine could not start? Then check the daemon's log file. It should list the exact command used to start the engine. You can try that manually to check why it didn't work.
- "Unknown. Found lock file"
This means that the respective ArchiveEngine did not respond when we

queried it, but there is an “archive_active.lock” lock file. This could have two reasons. It could mean that the engine is running but it was temporarily unable to respond to the daemon’s request. An example would be that the engine is really busy writing and dealing with ChannelAccess, so that its web server had to wait and the daemon timed out. All should be fine again after some time.

If, on the other hand, the situation persists, it usually means that the engine is hung or has crashed, so that it does not respond and the lock file was left behind. See Crashes on page 90.

NOTE: The daemon queries only every once in a while and leave the engines alone for most of the time. Especially after startup, all engines will show up as “Not Running” in the daemon’s web page while in fact most of them are already running. Then you will see many disconnected channels while the engines did in fact already connect to all channels. If you are impatient, you can click on the links to the individual engines to get a more up-to-date snapshot of each engine’s status.

7.0.4 Disabling Engines

The web interface of the daemon contains a link for each engine that disables the engine. This places a file “DISABLED.txt” in the engine directory and stops the engine. The daemon will not attempt to start engines as long as the “DISABLED” file is found. This is a convenient way for temporarily disabling engines without removing them from the daemon’s configuration.

7.0.5 Stopping and More

To stop the ArchiveDaemon, access the “/stop” URL of the daemon’s HTTPD, e.g. “http://localhost:4610/stop”. Similar to the ArchiveEngine’s HTTPD, this URL is not accessible by following links on the HTTPD’s web pages. You will have to type the URL. This is to prevent web robots or a monkey who is sitting in front of the computer and clicking on every link from accidentally stopping the daemon. Finally, the daemon will respond to the URL “/postal” by stopping every ArchiveEngine controlled by the daemon, followed by stopping itself.

Chapter 8

Setup, Installation

In general, the archiver toolset should build on Unix-type operating systems that are supported by EPICS base 3.14. Specific instructions for Linux (RedHat and Mandrake) as well as the Apple Mac OS X follow.

8.1 Compilation

The archiver tools use the EPICS build system as for example described in the “EPICS: Input/Output Controller Application Developer’s Guide” for Release 3.14.4. This means you need the following prerequisites:

1. EPICS Base R3.14.4 (or later) needs to be built and installed.
Unless you are running Linux, this might require getting a compiler, perl and gnumake.
2. An EPICS extensions setup: “configure” directory with the RELEASE file appropriately configured to point to your EPICS base installation.
3. ChannelArchiver sources, placed in the “src” subdirectory of your EPICS extensions directory tree.

All the above is either pretty obvious to those who know it already or beyond this manual to explain, in which case we have to refer you to the EPICS web site <http://www.aps.anl.gov/epics>.

You need to read and maybe adjust Tools/ToolsConfig.h and LibIO/ArchiverConfig.h to suit your needs. The most important parameter in there is “CONVERSION_REQUIRED”. Assert that it is correctly configured! If you use the wrong setting for CONVERSION_REQUIRED, you might not notice any problems for some time, but your data files will be invalid when transferred to another operating system; your network data server will only provide garbage data to network clients.

One good thing to do for a sanity check is to run the ArchiveExport tool on the data provided in the DemoData subdirectory. Try to retrieve the PV “DoublePV”; you should see values in the range of 0...10 with time stamps in recent years:

```
ArchiveExport DemoData/index DoublePV - text
# Generated by ArchiveExport 2.1.4
# Method: Raw Data

# Time                               DoublePV [a.u.]
03/05/2004 18:54:41.742248000        2.7
03/05/2004 18:57:50.543731200        3
03/05/2004 18:57:50.563760000        3.5
03/05/2004 18:57:50.583788800        3.9
...
```

You will of course only be able to test ArchiveExport after you have successfully built the archiver toolset, so read on. In addition to the configuration of the archiver sources themselves, some open source tools and libraries are required which are listed in the following subsections. They are included in the “ThirdParty” subdirectory of the archiver sources.

With all the required “ThirdParty” components in place, building the ChannelArchiver should be reduced to typing “make” in the ChannelArchiver source directory, followed by the optional setup of the Matlab/octave glue code which is described in a README file in the Matlab subdirectory of the ChannelArchiver sources.

8.1.1 XML-RPC

The archiver’s network data server uses XML-RPC. The XML-RPC Setup requires the installation of at least the C/C++ support. The Java archive data client includes the JAR files for XML-RPC access from Java. If you want to access the archive data server from e.g. perl, this would mean you have to install XML-RPC support for perl, too (one of which is included in the ThirdParty subdirectory of the archiver sources).

For C and C++, we use xmlrpc-c from <http://xmlrpc-c.sourceforge.net>. The Makefiles in ChannelArchiver/XMLRPCServer assume this to be installed in the default location, that is under /usr/local.

With RedHat 6.2, xmlrpc-c compiled out of the box. Under RedHat 9.0 and Mandrake 10, it ran into a compile-time error that could be fixed by uncommenting “using namespace std;” in the header file which reported the error. Under Fedora Core 2 and Mandrake 10, there were additional errors that can be fixed by replacing includes for “strstream.h” with “strstream” in the affected files. Both RedHat AS and Mandrake 10 needed additional packages, see below.

NOTE: The default serialization code in xmlrpc-c-0.9.9 will serialize sufficiently small numbers as zero, which means that e.g. many of your archived vacuum pressure readings of 1.510^{-8} end up as 0 when accessed via XML-RPC (see also section 4.4.5). The “ThirdParty” subdirectory contains the sources for xmlrpc-c-0.9.9 together with a patch files that corrects the “using namespace std;” and the serialization issue as well as the “strstream” problem. Under Red-Hat 9 respectively Mandrake 10, the complete installation would then look as follows:

```
cd ChannelArchiver/ThirdParty
tar vzxvf xmlrpc-c-0.9.9.tar.gz
cd xmlrpc-c-0.9.9
./configure
# Patch for serialization and name spaces
patch -p1 <../patch_xmlrpc-c-0.9.9
# Fedora and Mandrake 10 patch for strstream
# SKIP THIS PATCH ON REDHAT 9!
patch -p1 <../patch_xmlrpc-c-0.9.9_strstream
make
su
make install
```

Since the first patch also affects Makefiles which are created as a result of “configure”, you might prefer to read that patch file and apply the changes manually when you’re not on RedHat 9 or Mandrake 10.

For Mac OS X, a version with patched configure scripts [6] that also includes the “small numbers” patch mentioned above is included in a different tar file:

```
tar vzxvf xmlrpc-c-0.9.10_darwin.tgz
cd xmlrpc-c-0.9.10
./configure
make
sudo make install
```

The XML-RPC library depends on other packages. *The “configure” step will report errors in case those are missing.* For RedHat, those packages are usually included in the distribution but might not have been installed by default, so look for the RPMs on your RedHat CDs. The following are also provided in the ThirdParty subdirectory:

w3c-libwww

This is needed to compile the XML-RPC library. For Redhat or Fedora, you can instead install the libwww and libwww-devel RPMs that come with the OS. For Mandrake 10, you can use the w3c-libwww sources as is:

```
cd ChannelArchiver/ThirdParty
```



```
tar vzxw w3c-libwww-5.4.0.tgz
cd w3c-libwww-5.4.0
./configure
make
su
make install
```

Mac OS X requires a patch to the configure script [6]:

```
cd w3c-libwww-5.4.0
patch <../w3c-libwww-5.4.0_osx_patch
./configure --enable-shared --enable-static \
    --with-zlib --with-ssl
make
```

8.1.2 Xerces XML Library

The Xerces library is used to parse the XML configuration files of the Archive-Engine, IndexTool, and the network data server. See “Xerces C++” under

<http://xml.apache.org/index.html>

or try this direct link:

<http://xml.apache.org/xerces-c/index.html>

to get the sources. The Makefiles assume this to be installed under /usr/local. Example installation under RH9:

```
tar vzxw xerces-c-current.tar.gz
cd xerces-c-src2_4_0
export XERCECROOT='pwd'
cd $XERCECROOT/src/xercesc
autoconf
./runConfigure --plinux --cgcc --xg++ \
    --minmem --nsocket \
    --tnative --rpthread \
    --P/usr/local

make
su
make install
```

For Mac OS X the runConfigure looks like this:

```
./runConfigure -p macosx -n native -P /usr/local
```

Newer compilers (like gcc 4.0 as used on Mac OS 10.4) need to patch this:

```
Add
    #include <xercesc/framework/MemoryManager.hpp>
to
    src/xercesc/util/RefArrayOf.hpp
```

8.1.3 Expat

As an inferior alternative to Xerces, the Expat library is supported after changing Tools/FUX.h. Expat comes with e.g. RedHat 9, otherwise see

<http://expat.sourceforge.net>.

Expat might be a little faster and easier to install, but it does not offer validation, so it will be up to you to assert that all XML configuration files are 100% perfect.

8.1.4 XML-Simple

This XML library for perl is used by the ArchiveDaemon. It is available from

<http://www.cpan.org>.

```
tar vzxvf XML-Simple-2.09.tar.gz
cd XML-Simple-2.09
perl Makefile.PL
su
make install
```

8.1.5 Frontier

Frontier is an XML-RPC library for perl. It is used for tests of the XML-RPC Archive Data Server, including the ArchiveDataClient.pl test script. Under RedHat 9.0, Fedora 2 and Mandrake 10, it was sufficient to install Frontier-RPC-0.07b4 like this:

```
tar vzxvf Frontier-RPC-0.07b4.tar.gz
cd Frontier-RPC-0.07b4
perl Makefile.PL
sudo make install
```

RedHat 6.2 was hopeless because many of the required perl packages were missing.

8.2 Installation

There are no specific installation procedures for the ArchiveEngine, ArchiveExport, and most other Channel Archiver components. The binaries for them end up in the standard EPICS extension directories, which should therefore be included in the search path. If the archiver libraries were build as shared libraries, most Unix systems will require the extensions' lib directory be added to the LD_LIBRARY_PATH. The same applies to other helper libraries like Xerces that might be in the form of shared libraries.

The usage of the ArchiveEngine and other archiver tools might require configuration files, the format of which is described as part of the tools section in this manual.

The ArchiveDataServer requires integration with your web server. The process is exemplified in the Data Server chapter starting on page 31.

8.2.1 DTD Files

Many of the configuration files use XML, and document type definitions are provided in the form of DTD files (See ArchiveDataServer configuration in 4.1, ArchiveEngine config. in 3.1, ArchiveDaemon in 7.2, ArchiveIndexTool config. in 5.1). You are *strongly* encouraged to reference these DTD files in all your XML files, and to use the validating Xerces XML library, so that all your XML get validated while the ChannelArchiver tools use them. This means that your XML files need to include a DOCTYPE declaration that points to the location of the respective DTD file. In practice, there are at least three ways to accomplish this:

1. Whereever you create an XML file, you copy the DTD into the same directory. Then you can refer to the DTD like this:

```
<!DOCTYPE engineconfig SYSTEM "engineconfig.dtd">
```

Not the best idea because you need multiple copies of the DTD and this is hard to maintain in case the DTD gets updated.

2. You install the DTD files in a common location in the local file system, e.g. in "/arch". Then you can refer to the DTD like this:

```
<!DOCTYPE engineconfig  
SYSTEM "/arch/engineconfig.dtd">
```

This setup is in use at the SNS. If you use the archive tools on more than one computer, each machine might require a copy of the DTDs.

3. You install the DTD files in the directory tree of a web server that is accessible to all your computers. Then you can refer to the DTD via a URL like this:

```
<!DOCTYPE engineconfig  
SYSTEM "http://webserver/archdtd/engineconfig.dtd">
```

This centralizes the installation, but you now have the added dependency on the web server.

Chapter 9

Common Errors and Questions

The following explains error messages and commonly asked questions.

9.1 Why is there no data in my archive?

The ArchiveEngine should report warning messages whenever the connection to a channel goes down or when there is a problem with the data. So after a channel was at least once available, there should be more or less meaningful messages. After the initial startup, however, there won't be any information until a channel is at least once connected. So if a channel never connected, the debugging needs to fall back to a basic CA error search:

Is the data source available? Can you read the channel with other CA client tools (probe, EDM, cau, caget, ...)? Can you do that from the computer where you are running the ArchiveEngine? Does it work with the environment settings and user ID under which you are trying to run the ArchiveEngine?

9.2 Why do I get #N/A, why are there missing values in my spreadsheet?

There are several possible reasons for not having any data: There might not have been any data available because the respective channel was disconnected or the archive engine was off. Consequently there is no *value*, and a spreadsheet might show '#N/A' in the value column. When you look at the *status* of the channel, those cases might reveal themselves by status values like "Disconnected" or "Archive Off". (The ArchiveExport tool, Java client and other programs usually have a "status" or "text" option that you need to use in order to see the status. By default, you might only see the value).

The ArchiveEngine might also have crashed, not getting a chance to write “Archive Off”. That would be a likely case if no channel has data for your time range of interest. The most common reason for missing values, however, simply results from the fact that we archive the original time stamps and you are trying to look at more than one channel at a time. See the section on time stamp correlation on page 9.

9.3 Why do I not get what I think I should get from the network data server?

I do not know a good way to debug the archive data server when it runs as a CGI tool. Since it is started by the web server, one cannot easily attach a debugger to it. About the only thing that one can check:

```
/tmp/archserver.log
```

Per default, the archive data server will append to the archserver.log file whenever run. As a result of a 'get_values' request, it will log the request and some basic result information:

```
----- ArchiveServer Started -----
archiver.get_values
how=3, count=10
get_channel_data
Start: 03/23/2004 10:47:46.000000000
End : 03/23/2004 10:50:57.000000000
Method: Plot-Binning, 19.1 sec bins
Open index, key 1 = '../DemoData/index'
Handling 'fred'
40 values
Handling 'janet'
40 values
ArchiveServer ran 0.00754285 seconds
```

In case of problems, one can check if the data server received the correct request. On the machine where the data server is running, one can then try to reproduce the request with the command-line ArchiveExport tool.

To disable this data server log file, look for 'LOGFILE' in the data server sources.

9.4 Back in time?

The archiver relies on the world going forward in time. When retrieving samples from an archive, we expect the time stamps to be monotonic and non-decreasing. Time stamps going back in time break the lookup mechanism.

Data files with non-monotonic time stamps are useless. Unfortunately, the clocks of IOCs or other computers running CA servers can be mis-configured. The ArchiveEngine attempts to catch some of these problems, but all it can do is drop the affected samples, there is no recipe for correcting the time stamps.

Bottom line: You need to have the clocks of all CA servers properly configured (also see page 2). There are other reasons for back-in-time warnings that have no good solution:

- When a channel disconnects or when the ArchiveEngine is shut down, the engine will add “Disconnected” or “Archive Off” values to the archive. Those values will carry the current value of the host’s clock, the current time stamp of the computer that is running the engine. The host’s clock is rarely perfectly synchronized with the IOC’s clocks, so if we had just received a sample from the IOC and the host’s clock is only a little late, we’ll get a “back in time” warning.

To resolve this, the engine will tweak the time of the “Disconnected” or “Archive Off” values so that they’re stamped just after the last sample in the archive.

- When a channel gets disabled, it really gets disabled because *another* channel in its group, one that was configured as “disabling”, turned true and thus disabled the whole group. The engine writes a “Disabled” value for each disabled channel, and as a time stamp it uses the time stamp of the channel that caused the disable. This will later help you to see exactly when channels were disabled and why because the disabling and the disabled channels will all have the same time stamp.

A problem arises especially when the disabling channel is from a different IOC. Assume we just received samples for several channels and added them to the archive, and then some splitseconds later the value of the disabling channel arrives, delayed by the network. That disabling time stamp might be a little older than the stamps of the values that we already wrote, so the attempt to add a “Disabled” value results in back-in-time warnings (which again get resolved by hacking the time stamps).

- Some channels seldom change. Examples include “Setpoint” channels which are only modified by operator input. They might stay the same for days or weeks. When an ArchiveEngine stops, it will add an “Archive Off” event. When it is then re-started, it will receive the current value of a channel, which might older than the “Archive Off” event. So the current value causes a back-in-time situation, again resolved by using the last time stamp in the archive instead of the original stamp of the sample.

In summary, most back-in-time warnings can be ignored as long as the clocks of the hosts and IOCs are reasonably in sync.

9.5 Found an existing lock file 'archive_active.lock'

When the ArchiveEngine is started, it creates a lock file in the current directory. The lock file is an ordinary text file that contains the start time when the engine was launched. When the engine stops, it removes the file.

The idea here is to prevent more than one archive engine to run in the same directory, writing to the same index and data files and thus creating garbage data: Whenever the archive engine sees a lock file, it refuses to run with the above error message.

Under normal circumstances, one should not find such lock files left behind after the engine shuts down cleanly. The presence of a lock file indicates two possible problems:

- a) There is in fact already an archive engine running in this directory, so you cannot start another one.
- b) The previous engine crashed, it was stopped without opportunity to close the data files and remove the lock file. It *might* be OK to simply remove the lock file and try again, but since the crash could have damaged the data files, it is advisable to back them up and run a test before removing the lock file and starting another engine.

9.6 Crashes of the ArchiveEngine, ...

Though some care was taken in testing the ArchiveEngine, several problems will remain. Before you go all the way to implement a better if not perfect ArchiveEngine or other piece of the ChannelArchiver yourself, you might want to see if you can reproduce the crash and aid in debugging it.

Under Linux, this usually means: Allow the generation of “core” files and investigate them. With recent Linux distributions, add “set ulimit -c unlimited” to your login-script. When the ArchiveEngine crashes, it should now generate a core dump, often called “core.<PID>”, using the process ID of the crashed process. Then use “gdb” to locate the area of code that caused the crash:

```
cd /where/the/core/file/is
gdb /full/path/to/ArchiveEngine core.12345
# Generate a stack trace
bt
# You should see a stack trace leading up
# to the crash, something like
# 0  write (...)
# 1  printf (...)
# 2  MsgLogger(..)
# 3  ArchiveChannel::handle_value (...)
# 4  ...
```



```
# Now you can select some stack frames and see
# what is happening in there
frame 3
list
# ... shows the code in
#   ArchiveChannel::handle_value(..
# that leads up to the crash
quit
```

Please email the result so that we can try to eliminate the problem.

9.7 'ChannelName': Cannot add event because data type is unknown

The ArchiveEngine tried to write an event to the data file. Examples include a “Disconnected” or “Archiver Off” event. Even though this event does not have a value, it only indicates a status change or warning, it nevertheless is written into the same data buffer where ordinary values are written. A problem arises when we never got a connection to the CA server, therefore we do not know the value type of the channel and thus we cannot allocate a data buffer in which to write this special event-type of value.

9.8 Cannot create a new data file within file size limit

You specified a rather small file size limit (file_size option in the ArchiveEngine configuration), and the currently required buffer size for a single channel already exceeds that file size limit. The Engine will actually go ahead and create a bigger file in the hope that this avoids data loss.

One example that could cause this: You try to archive array channels, where each individual sample is already quite big, and picked a tiny file_size.

9.9 Found an existing 'indextool_active.lck' lock file

When an ArchiveIndexTool is started, it creates a lock file similar to the ArchiveEngine, in the hope of preventing more than one Index Tool from modifying a master index at the same time. See preceding description of archiver_active.lck file.

Chapter 10

Legacy

The main difference between the current version of the Channel Archiver toolset and previous versions are:

- EPICS Base: Current toolset requires R3.14.4 or later.
- Index: Current toolset uses index files, previous toolset used Directory files.
- Configurations: All configuration files are based on XML, while they used to be generic ASCII files.

10.1 Directory Files

In case you have existing archives based on Directory files, you can use the ArchiveDataTool to create an index file for each existing directory file and that way migrate to the new toolset. The ArchiveDataTool can also create Directory files from index files, in case you need to use an older tool to investigate data in new archives.

10.2 ArchiveManager

The ArchiveManager command line tool for managing archives that use the older Directory files is still available. It does *not* work with the new index files. After migrating the data to the new index format, ArchiveDataTool replaces the ArchiverManager. You can use the ArchiveDataTool to create index files for existing directory files.

10.3 Archive Engine ASCII Configurations

The ChannelArchiver/Engine directory contains ConvertEngineConfig.pl, a perl script that attempts to convert old-style ASCII configuration files for the previous ArchiveEngine into the new XML files:

USAGE:
 ConvertEngineConfig [options] ASCII-config { ASCII-config }

Options:

- d <DTD> : Specify DTD
- o <filename> : Set output file name

This tool reads the ASCII configuration file for an Engine and converts it into the XML config file.

One can use either one ASCII config file (with !group entries) or supply a list of ASCII files, where each one will then define a group.

Some sites might actually prefer to use the ASCII configuration files and convert them to the XML format with this tool, so here is a brief description of the ASCII format. Each ASCII file defines a group of channels.

- “!write_period <seconds>”
 Configures the write_period parameter. Most of the global engine parameters described in section 3.1 are allowed via the format “!<parameter> <value>”.
- “!group <file-name>”
 Read another group file. To use this, you should create one “main” configuration file that contains only “!group ...” lines. Preferably, do not use the “!group ...” command at all. Simply create one ASCII file per group, and list all the group files that you want one engine to handle on the ConvertEngineConfig command line.
- “<channel> <period>”
 Selects scanned sampling of the given channel.
- “<channel> <period> Monitor”
 Selects monitored sampling of the given channel.
- “<channel> <period> Monitor Disable”
 Selects monitored sampling of the given channel; channel disables its group..

10.4 CAManager, CAbgManager

The ArchiveDaemon offers the basic functionality of the CAbgManager: It monitors, starts and restarts ArchiveEngines and offers a web page for status information. There is no graphical configuration tool comparable to the CAManager, you have to create the XML configuration file for the ArchiveDaemon by hand.

Chapter 11

Changes

This chapter describes the version numbers and changes since the beginning of the R3.14 port.

- 01/30/2006? — Version 2.8.0:

Like a pendulum swinging back, I've overcome my initial disappointment with C++ exceptions and added them back in, because as long as the compiler supports them, their use eases the error handling: Detect and describe errors at the low level in the code where they happen, and pass them up to the high level code in the form of an exception.

In this concersion process, many unit-tests have been added and checked under valgrind.

- 12/13/2005 — Version 2.7.0:

Error reporting: When a command-line tool failed, one could see a rather specific error message like "Cannot open DataFile 20040323". It also ended up in the log file of the network data server, but was not reported well via the XML-RPC protocol.

Now an XML-RPC fault with the more specific error message is returned when an index or data file access fails. A request for a channel that is not found at all is now also considered an error by the network data server. When there is simply no data for the requested time, the meta information is returned without any actual samples, as before.

- 10/31/2005 — Version 2.6.0:

The toolset now compiles and runs on a 64 bit Linux system, specifically RedHat Enterprise Linux WS release 4 with gcc 3.4.4.

The goal here was to keep binary compatibility, so that index and data files can be moved from 32 bit computers to 64 bit machines and back. This means that the 32 bit limitations on file sizes still apply, even when running on a 64 bit computer.

Several new self-tests were added to check the following on the 64 bit system as well as 32 bit Linux and Mac OS X 10.3:

- All compiles with the 64 bit compiler, no warnings on the test system.
- 'Tools' library self-test passes.
- 'Storage' library self-test passes.
- ArchiveExport can dump demo data, matching results on all architectures.
- ArchiveDataTool can copy demo data, matching results on all architectures.
- DataServer runs from within shell script that simulates a CGI environment.
- ArchiveEngine runs, HTTPD responds, collected data passes simple tests.

- 09/21/2005 — Version 2.5.0:
New engine URL

```
.... castatus?/tmp/x
```

will cause the engine to dump the `ca_client.status()` output into the given file the next time it writes to the disk.

- 08/29/2005 — Version 2.4.0:
LinearReader rounds start time, because otherwise a request for multiple channels, all using linear interpolation, could loose lockstep, and when the result is put in a spreadsheet form, there are many more lines than expected from the per-channel view of the data, which confused the data viewer.

XML-RPC, being XML, sends all numbers as strings. Unfortunately, XML-RPC also insists in the simple 'dot' notation and prohibits exponential notation. A number like $1e-300$ would turn into "0.000000..." with 300 zeroes, which is too long for the internal print buffer of the xml-rpc C library. Since such huge and tiny numbers can't be transferred, they are replaced by 0 with stat/sevr UDF/INVALID.

The cut-off point is somewhat arbitrary. The XML-RPC library uses an internal print buffer of about 120 characters. Since PVs are usually scaled to be human-readable, with only vacuum readings using exp. notation for data like "1e-8 Torr", exponents of +-50 seemed reasonable.

- 07/18/2005 — Version 2.3.0:
Added ArchiveDataServerStandalone.
- 05/31/2005 — Version 2.2.1:
At the SNS with EPICS R3.14.7, there is an instance of an engine crashing in `ipAddrToAsciiEnginePrivate.cpp` in the `transactionComplete` callback while `pCurrent` was 0, which shouldn't happen:

```

if (!this->pCurrent)
    continue;
{
    epicsGuardRelease < epicsMutex > unguard ( guard );
    this->pCurrent->pCB->transactionComplete ( this->nameTmp );
}

```

In the process of debugging it, an uninitialized use of the HTTPD run duration variable and a scanlist memory leak on exit have been found by 'valgrind'. None of this should matter in the real world, but I prefer as little valgrind complaints as possible. Remaining memory leaks on exit are in the EPICS base code.

- 03/31/2005 — Version 2.2.0:
Added a "Write Duration" status to the main page of the Engine HTTPD. It is a running average of the time spent writing to the disk. ExampleSetup includes a new script, "engine_write_durations.pl", which queries all engines for their write duration. The intent is to get an idea of how much time is spent doing disk I/O, and to get a grasp on which engine is doing worst.
- 03/21/2005 — Version 2.1.9:
The ArchiveDaemon.pl script now creates a soft-link "current_index" in the engine subdirectories, pointing to the currently used index. The idea is to have a common name for the active index in case the multi-index is broken or only updated slowly.

The next issue was that the relation of an index to its data files must not change. If an index is in the same directory as its data files, OK, but this soft link was actually in a different location, so in order to still find the data files, the retrieval now follows sym-links to index files on level deep (not following arbitrary chains of soft links).
- 02/01/2005 — Version 2.1.8:
At the SNS, one ArchiveEngine running a configuration with 'disabling' channels tended to hang up just after a channel 'disabled' a group. While I was never able to capture this in the debugger or otherwise reproduce it, I did find a violation of the lock order in the code related to 'disabling' a group, so this might have been the reason.
- 10/26/2004 — Version 2.1.7:
In monitored mode, engine will always try to add two initial samples: One with the original time stamp, one with the host time stamp. This solves the restart problem:

Assume a setpoint PV is 3 days old. When we start an engine today, it'll write that 3-day old value. When we then stop and restart tomorrow in a new directory, a "Disconnected" value will be added before shutdown and then the same happens in the new directory. When now looking at

both days, you would get the 3-day old value, a disconnect/off value and nothing else.

With this update, you will see one value with the host's time stamp each day in addition to the original time stamp, so even when viewed via a master-index, one sample per day should be visible.

ArchiveDaemon.pl no longer complains about an undefined "opt_i", thanks to Paul Sichta for pointing this out.

- 09/07/2004 — Version 2.1.6:
Added Enable/Disable code to ArchiveDaemon. "ListIndex" bug fixes (sub-index was closed even though we're still accessing channels in the index).
- 08/26/2004 — Version 2.1.5:
Fixed bug in the engine code: PVs which never change were never written when sampling with period `j` `get.threshold`.
Working on a "ListIndex", allowing the retrieval tools to use an `indexconfig.dtd`-type list of sub-archives, querying them one by one. The result is slower than using a "real" index, but much easier to setup and maintain.
- 07/26/2004 — Version 2.1.4:
Bug in DataReader that affected all the retrieval code: The "find" uses the start time in an at-or-before sense, which is intentional for direct calls to `find()`. This happened, too, when it was used internally for the purpose of switching to a new data block ref'd by a (master) index. In that case, however, the start time given by the index must be observed in an at-or-after sense, otherwise we can go back in time.
Added the weekly option to the ArchiveDaemon. Not detailed in manual until we get some mileage on it.
- 07/23/2004 — Version 2.1.3:
Index update had a flaw: When an engine stops, that last value received via CA might have a time stamp of 10:00 and then we stop at 11:00, so the last stored value is the "off" value at 11:00. When now a new engine starts, the first data is still 10:00, and that data is hidden under the last data block of the previous engine (...11:00). When then the new engine added more data, eventually beyond 11:00, that new data stayed hidden unless one rebuilt the master index from scratch. Hopefully fixed this.
- 06/30/2004 — Version 2.1.2:
Added tags and channel names to the Data files, so that in the future one could try to write a rescue tool. Patch for XML-RPC C/C++ lib. allows small numbers. DataTools' "index2dir" option now actually works. Matlab/Octave glue code can handle an array channel (when requesting a single channel, raw data). ArchiveDaemon generates `indexupdate.xml` for each re-index run; "-u" option.

When retrieval uses a "master" index and reaches its end, it will try to continue by following links of the last data block in the sub-archive. This will allow us to get closer to "now" between updates of the master index. The first attempt to implement this failed because of errors in the handling of the path names (master index has path to data files, but when we follow the links inside the sub-archive's data file, those are relative to where the sub-archive resides). Still not fully tested.

- 04/01/2004 — Version 2.1.1:
Many little updates have been checked into CVS while the tools reported "2.1.1". In the end, the engine supported sampling, monitoring and sampling-based-on-monitors, and ran without known problems under valgrind. The XML-RPC Data Server seemed to work fine, supporting raw data, plot-binning, spreadsheets, averaged and linear interpolation. Java Archive client is useable. Switched index file to CAI2, where the name hash includes a filename for the RTree. For now it's left empty, but the file format now allows for a further extension where certain RTrees are in separate files as soon as the index file gets too big.
- 01/27/2004 — Version 2.1.0:
Uses new RTree, initial XML-RPC Data Server, XML configuration files.
- 09/05/2003 — Version 2.0.1:
Some bug fixes: The "scanned" operation didn't work, and when all was monitored, the empty scan lists lead to a high CPU load. Still not perfect, the ChannellInfo code should be split into really monitored, scanned using CA monitor and scanned using CA get.
- 04/04/2003 — Version 2.0:
Starting to work on R3.14 port.

Bibliography

- [1] *EPICS Web Page*, <http://www.aps.anl.gov/epics/>. 1
- [2] J. O. Hill: *Channel Access: A Software Bus for the LAACS*, ICALEPCS 1989, Vancouver. 1
- [3] K. U. Kasemir, L. R. Dalesio: *Overview of the Experimental Physics and Industrial Control System (EPICS) Channel Archiver*, Internat. Conf. on Accel. and Large Experim. Phys. Control Systems (ICALEPCS) 2001, San Jose, CA.
- [4] Antonin Guttman: *R-Trees: A Dynamic Index Structure for Spatial Searching*, Proc. 1984 ACM-SIGMOD Conference on Management of Data (1985), 47-57. 51
- [5] Marty Kraimer et al: *IOC Application Developer's Guide R3.14.4*, <http://www.aps.anl.gov/epics/>.
- [6] *Web Page of the DarwinPorts Project*, providing help with porting various open-source packages to Mac OS X, <http://darwinports.opendarwin.org>. 82, 83

Index

'#N/A', 9, 87

archiveconfig.csv, 65
archserver.log, 88
Averaging, 11

CA, 1
ChannelAccess, 1
ChannelAccess servers, 3

data sources, 3
daylight saving time, 8
DISABLED.txt, 79

EPICS, 1
EPICS Time, 7
EPICS_TS_MIN_WEST, 7
error messages, 87
event, 91
Expat, 84

Filling, 10

Global Option, 18
Greenwich Mean Time, 7

IOC, 3

Linear Interpolation, 11
Local Time, 7
lock file, 90

make_archive_dirs.pl, 66
make_archive_web.pl, 66
make_compress_script.pl, 71
Master Index, 50
Master Indices, 55
meta information, 4

onlineconfig.xml, 25

PCAS, 3

raw spreadsheet format, 9
records, 3

sampling options, 3
Spallation Neutron Source (SNS), 64
Staircase Interpolation, 10
Sub-Archive, 50
Sub-Archive Priorities, 50

time stamps, 2

UTC, 7

Xerces, 83
XML-RPC Setup, 81