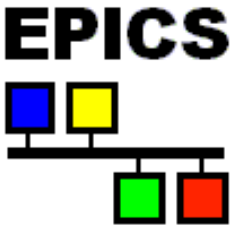
Channel Archiver Manual



R3.14.4 Version,
March 29, 2004.

Involvements

Bob Dalesio designed of 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.

Contents

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 deadband (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.
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?”

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 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

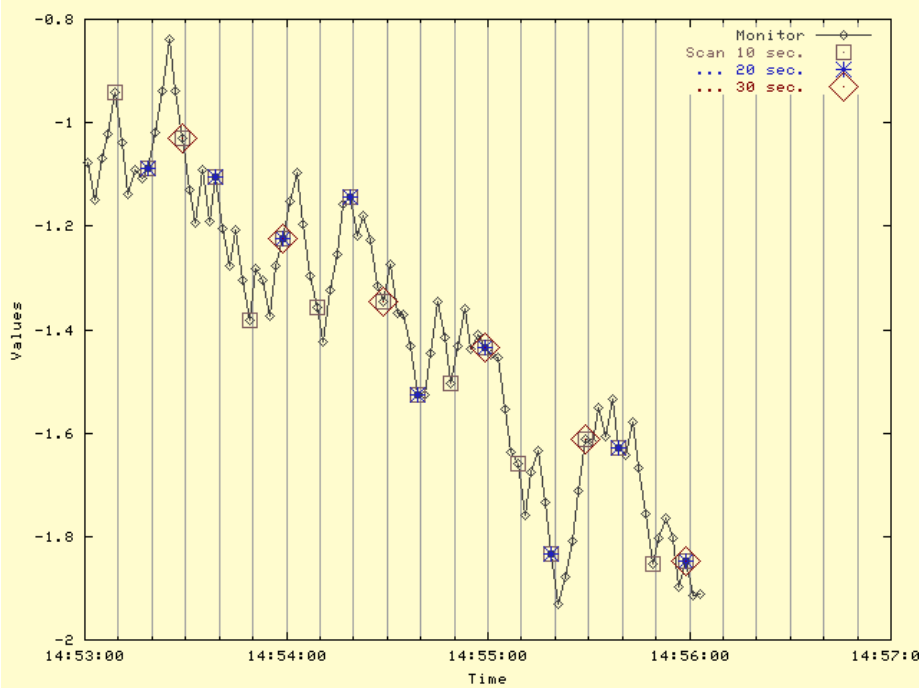


Figure 2.1: Time Stamps and Sampling

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 temperatures 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 position 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 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.

2.6.1 "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.

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.

2.6.2 "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.

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.

2.6.3 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 2.6.2, 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	*0.0718241*	-0.086006
3/22/2000 17:02:37.400964000	0.0543581	*-0.086006*
3/22/2000 17:02:37.510961000	*0.0543581*	-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 is the preservation of original time stamps.

2.6.4 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.

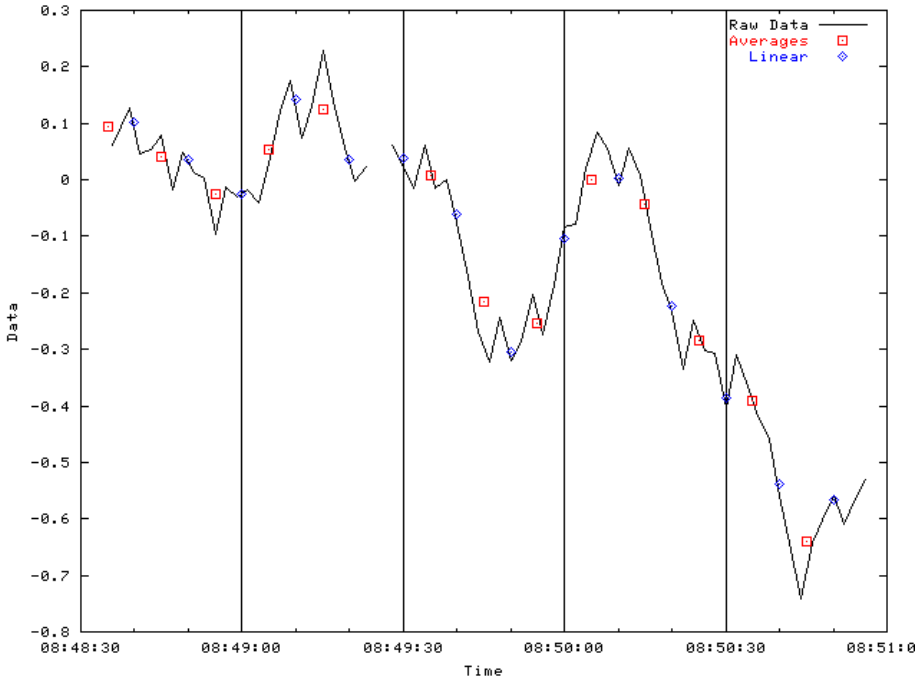


Figure 2.2: Averaging and Linear Interpolation, see text.

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 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 values 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.

2.6.5 Plot-Binning

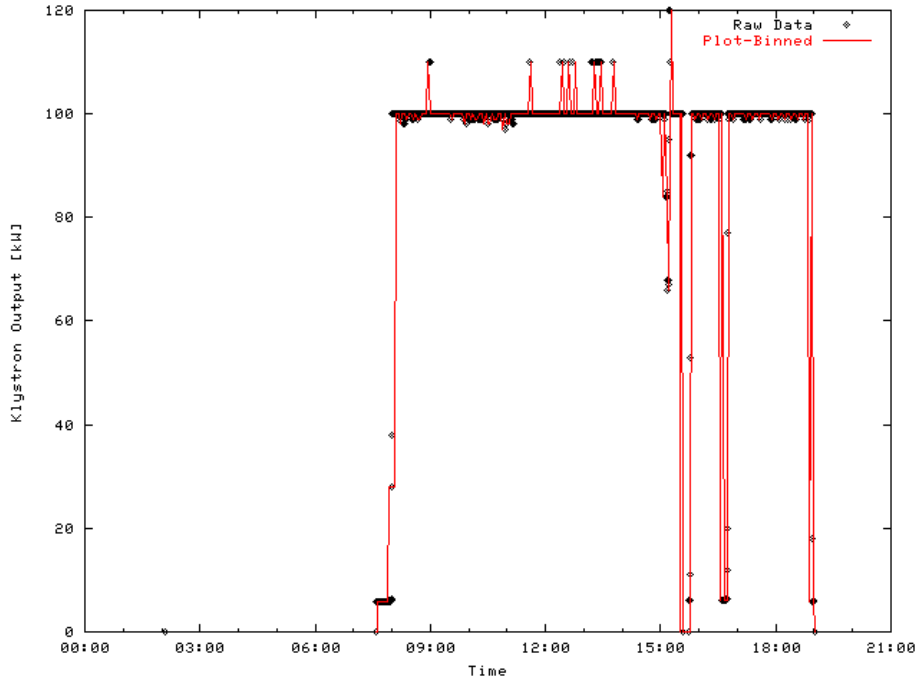


Figure 2.3: Plot-Binning, see text.

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 who's 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 were 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

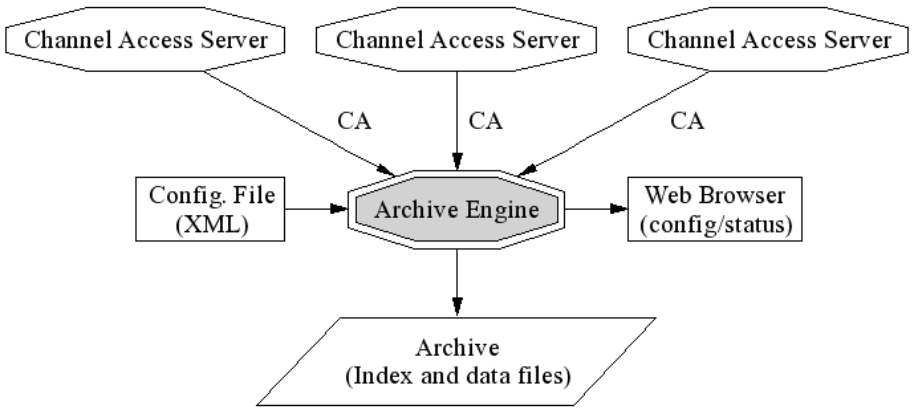


Figure 3.1: Archive Engine, refer to text.

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 6 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 29, 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)*, 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 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 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 3.1.8.

name

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

3.1.8 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 or one second, the channel will in fact be monitored with an estimated period of 1 second.

monitor

Either "monitor" or "scan" needs to be provided as part of a channel configuration to select the sampling method. In the case of "monitor", the channel will be monitored, that is: Each change received via ChannelAccess will be stored. 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

scan

As an alternative to the "monitor" tag, "scan" can be used to select periodic sampling. The preceding "period" tag determines the sampling period.

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.

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 (yet), 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.

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.lck" 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

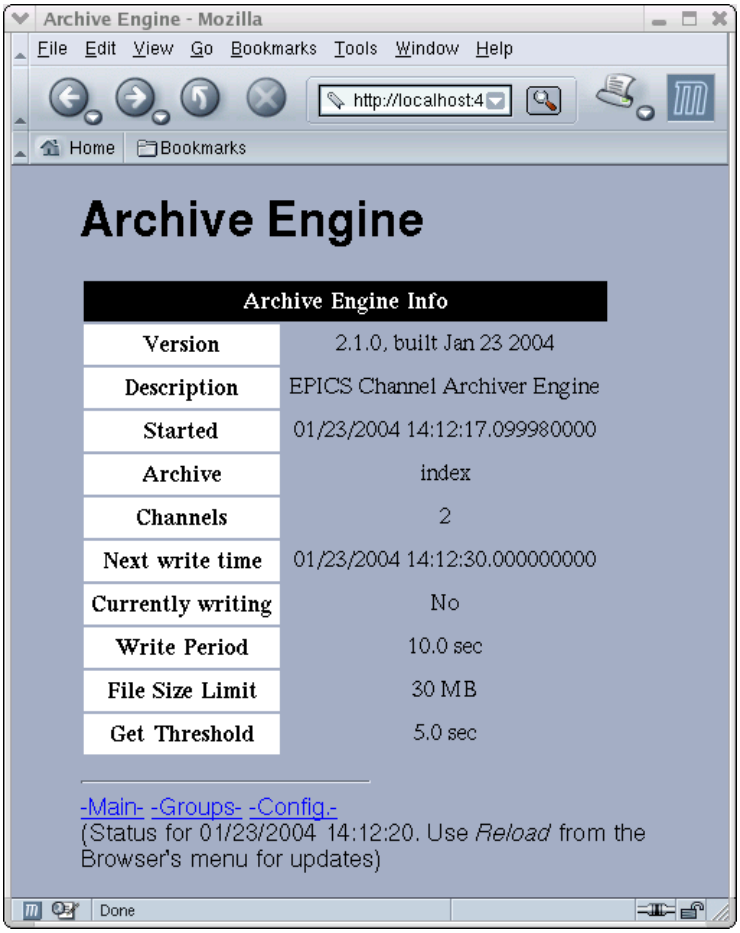


Figure 3.2: Main Page of Archive Engine's HTTPD

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 5) 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 is 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 17.

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

ArchiveDaemon

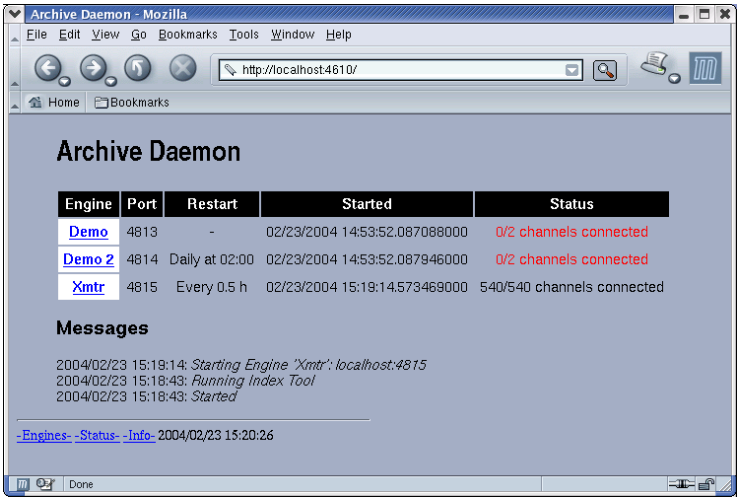


Figure 4.1: Archive Daemon, refer to text.

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. 4.1.

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 each sub-archive of newly created ArchiveEngines to a configuration file for the ArchiveIndexTool and runs the latter periodically, so that all the sub-archives can be accessed as if they were one big archive.

Before using the ArchiveDaemon, one should be familiar with the configu-

ration of the ArchiveEngine (sec. 3), and how to start and stop it. Furthermore, one needs to be familiar with the ArchiveIndexTool (sec. 6.3).

4.0.1 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 4.1, needs to follow the DTD from listing 4.2.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE engines SYSTEM "ArchiveDaemon.dtd">
<engines>
   <engine>
      <desc>Demo</desc>
      <port>4813</port>
      <config>/data/arch1/engineconfig.xml</config>
   </engine>
   <engine>
      <desc>Demo 2</desc>
      <port>4814</port>
      <config>/data/arch2/engineconfig.xml</config>
      <daily>02:00</daily>
   </engine>
</engines>
```

Listing 4.1: Example Archive Daemon Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for the ArchiveDaemon Configuration -->
<!ELEMENT engines (engine+)>
<!ELEMENT engine (desc,port,config,(daily|hourly)?)>
<!ELEMENT desc   (#PCDATA)> <!-- Text -->
<!ELEMENT port   (#PCDATA)> <!-- TCP port number -->
<!ELEMENT config (#PCDATA)> <!-- path -->
<!ELEMENT daily  (#PCDATA)> <!-- HH:MM-->
<!ELEMENT hourly (#PCDATA)> <!-- (double) hours -->
```

Listing 4.2: XML DTD for the 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.
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.

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.

4.0.2 Starting and Running

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

```
$ cd whereever_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. 4.1 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
```

The first few lines of the ArchiveDaemon.pl script contain numerous configuration variables. They allow fine tuning of e.g. the time period between runs of the ArchiveIndexTool, 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.lck
  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.

4.0.3 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 5

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 2.6.

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.

5.1 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 2.6 on page 7 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'
```

5.2 Java Archive Client

This tool is meant to be the main data retrieval tool. It provides a graphical user interface to allow data browing. It is based on Java and hence usable on many operating systems. It accesses the data via the DataServer described in section 5.3, 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. 5.1 then allows you to select one of the archives served by the network data server, investigate the available channels etc.

In short, it’s the greatest thing since canned beer. Once it’s done.

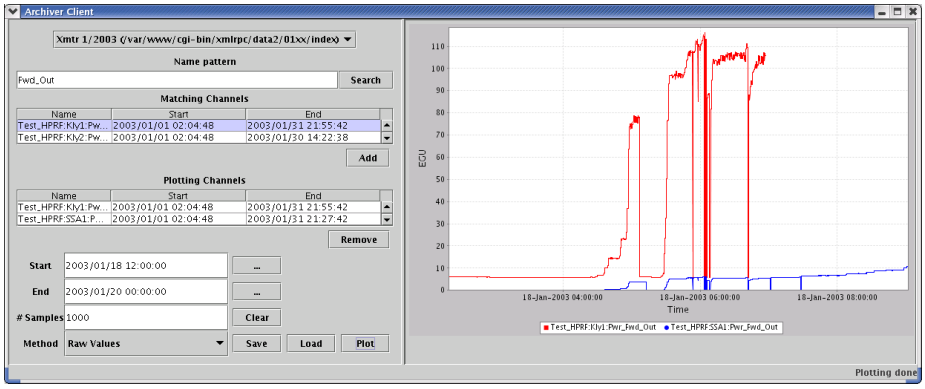


Figure 5.1: Java Archive Cient.

5.3 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.

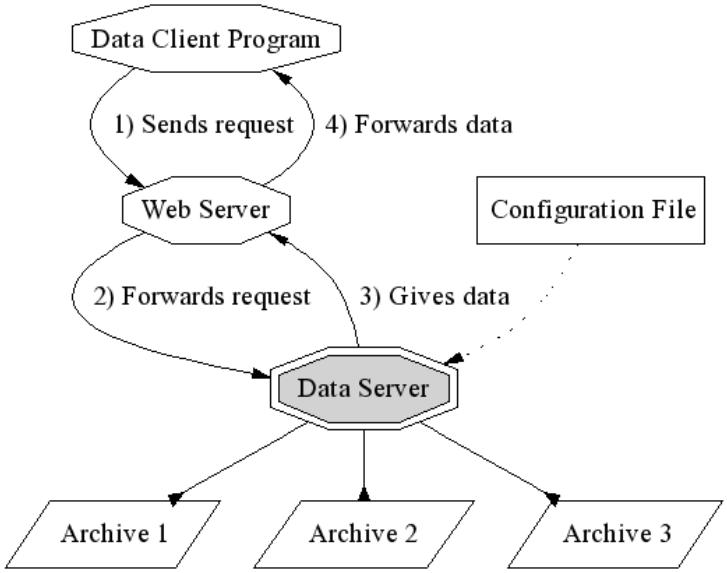


Figure 5.2: Data Server, refer to text.

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. 5.2, 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 5.4 on

page 35. 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 5.2) as a generic, interactive data client.

5.3.1 Installation

After successful compilation in ChannelArchiver/XMLRPCServer, you will have a program "ArchiveDataServer". You need to copy that binary as "ArchiveDataServer.cgi" into your web server's CGI directory and assert that the web server can execute the ArchiveDataServer. The ".cgi" extension is important, because otherwise your web server might not recognize your CGI program as such. What follows is an example setup for the Apache Web Server under Linux:

1. Check your Apache configuration file, which can often been found in "/etc/httpd/conf/httpd.conf", for the location of your cgi-bin directory. In my case, there was already a CGI directory /var/www/cgi-bin.

2. Create a new CGI directory so that we can specifically configure it. Adding the following to the Apache config file worked for me:

```
<Directory /var/www/cgi-bin/xmlrpc>
    SetEnv LD_LIBRARY_PATH /usr/local/lib:...
    SetEnv SERVERCONFIG /var/www/cgi-bin/ \
                    xmlrpc/serverconfig.xml
</Directory>
```

   The LD_LIBRARY_PATH needs to include all the directories that contain shared libraries which your ArchiveDataServer.cgi uses. In many cases, this means the EPICS base and extensions' lib directories as well as the install location of the expat and XML-RPC libraries. The SERVERCONFIG variable needs to point to your server configuration file, more about which next.

5.3.2 Setup

You need to prepare an XML-formatted configuration file for the ArchiveDataServer that follows the DTD from listing 5.1 (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 5.2 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 5.1: XML DTD for the Data Server Configuration

```
<?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 5.2: Example Data Server Configuration

5.4 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.

5.4.1 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 5.4.4.

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 5.1.

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 5.1. Instead, it provides the repeat count N. Table 5.2 lists the typical content of the “sevr” array, table 5.3 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 5.1: 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	false	false
3856	Repeat	false	false
3904	Disconnected	false	true
3872	Archive_Off	false	true
3848	Archive_Disabled	false	true

Table 5.2: 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 5.3: Examples for decoding samples returned from the archiver.values() call based on their Status and Severity

5.4.2 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.

5.4.3 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.

5.4.4 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 2.6 and following, beginning on page 7.

how = 0 (raw): Get raw data from archive (see 2.6.1), 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 2.6.3). 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 5.2 and 5.1).

how = 2 (averaged): Get averaged data from the archive, starting w/ ’start’, up to either ’end’ time or max. ’count’ samples (see 2.6.4). 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 2.6.

how = 3 (plot binning): Uses the plot-binning method based on ’count’ bins (see 2.6.5).

how = 4 (linear): Get linearly interpolated data from the archive, starting w/ 'start', up to either 'end' time or max. 'count' samples (see 2.6.4). 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 2.6.

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 5.4.1 and the tables 5.1, 5.2 and 5.3.

5.5 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
```

Chapter 6

Data Storage

6.1 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 leave 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.

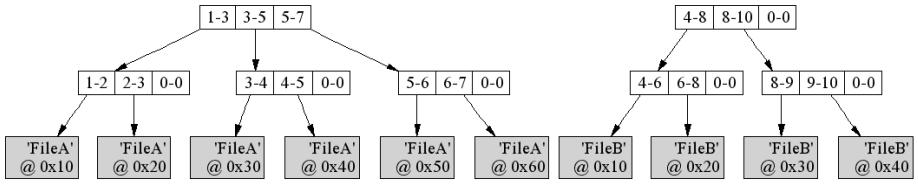


Figure 6.1: RTree Demo, refer to text.

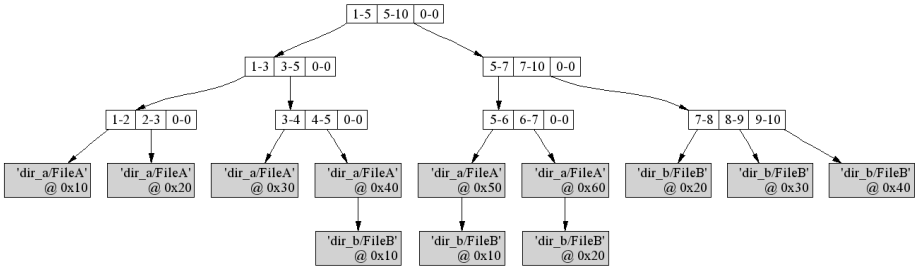


Figure 6.2: RTree Demo, refer to text.

Fig. 6.1 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. 6.2 demonstrates how the two trees from Fig. 6.1 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 neglegible 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 6.5 will present some quantitative details on the performance of the RTree index.

6.1.1 Implementation Details

Table 6.1 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 6.1: 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. 6.1 and 6.2, 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.

6.2 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.

6.2.1 Implementation Details

Offset	Content
...	...
0x1000	Numeric CtrlInfo
	display limits, units, ...
...	...
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, ...
...	...
0x4000	Data Header
	prev buffer: "X", 0x2000
	next buffer: "Y", 0x4000
	CtrlInfo: 0x1000
	...
	Buffer: dbr_time_double, dbr_time_double, ...
...	...

Table 6.2: Data file: Example layout for a data file "X"

Table 6.2 shows the basic layout of a data file "X". Most important, the data file only stores data. It doesn't 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 useless without the accompanying index file.
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 6.2, 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 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.

6.3 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 6.1 (see section 8.2.1 on DTD file installation). Listing 6.2 provides an example.

```
<?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 6.1: XML DTD for the Archive Index Tool Configuration

We refer to chapter 7 for an example of how to use the ArchiveIndexTool in collaboration with the other Channel Archiver tools. 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 appropriately formatted XML:

```
<?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 6.2: Example Archive Index Tool Configuration

```
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.
```

6.4 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 some consistency tests
```

6.5 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 again follows logarithmically from 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.

6.5.1 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.

6.5.2 Index Performance

Performance and size of the index depend on the M value configuration of the RTree. Fig. 6.3 displays the file size and the time needed to create an index for

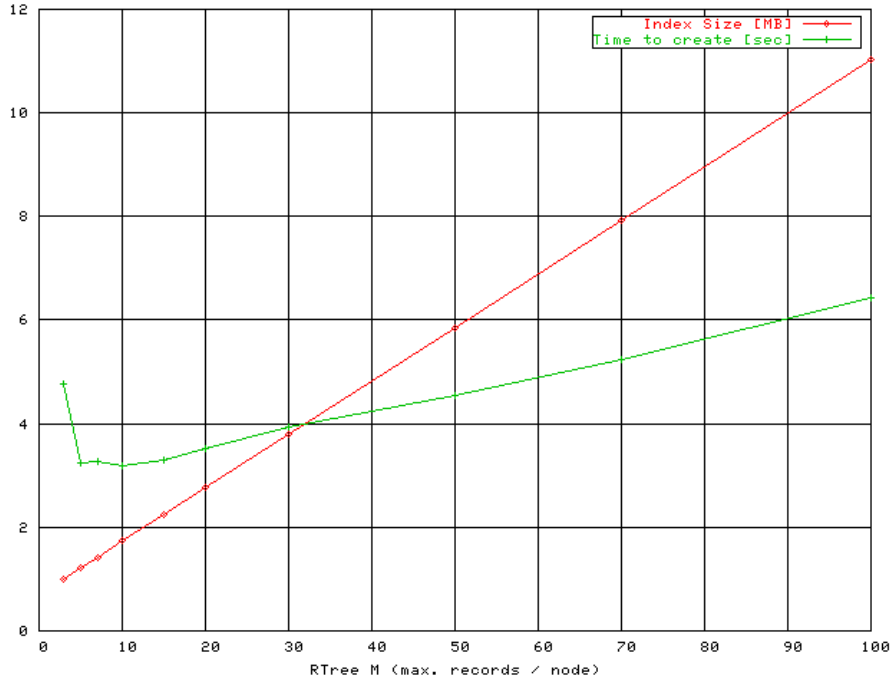


Figure 6.3: RTree M value tuning for small index, see text.

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. 6.3 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. 6.4 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

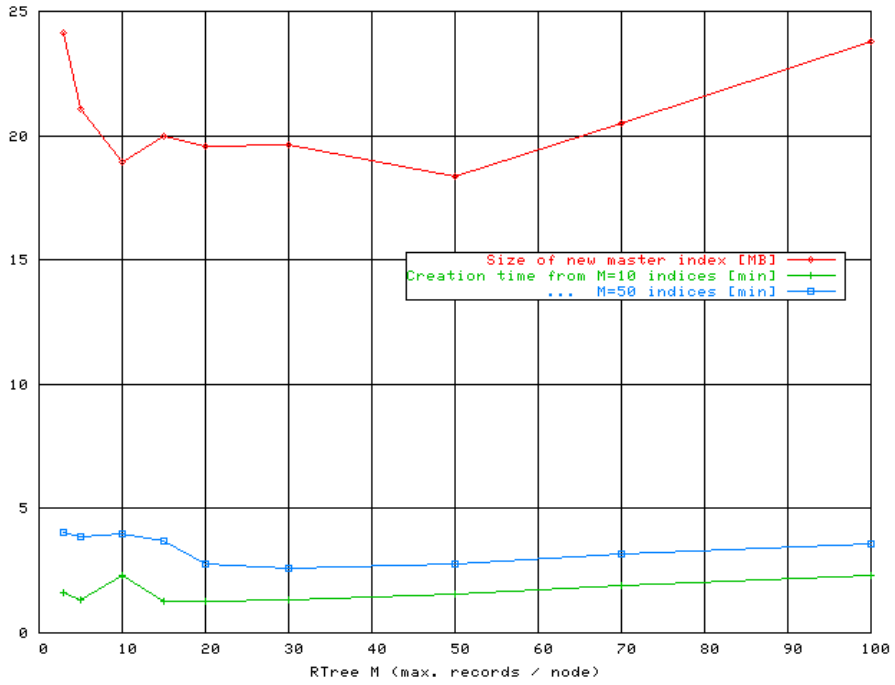


Figure 6.4: RTree M value tuning for master index, see text.

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. 6.4 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

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.

With the 92-sub-archive index, the following was tested:

- Time to list all channel names: $<$1 second.
  This took about 7 seconds with the old "multi archive" file that required opening the individual sub-archives.
- Time to find channels that match a pattern and show their start/end times: 0.2 seconds (4 channels out of 500 matched).
  This took about 6 seconds with the previous "multi archive".
- Seek test, i.e. find a data block for a given start time: Index file requires $<$0.5 seconds, old index uses 1.5 seconds.

6.5.3 Data Management 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 into 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.

6.5.4 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. The output was redirected to /dev/null to avoid additional penalties.

- Dump all the 143000 values for a channel: 4 seconds, translating into 35700 values per second.

Chapter 7

Example Setup

The following is meant as an example of how to use the various tools together. Assume that we have channels related to vacuum readings and channels related to a cooling system. As a result, we created the following two configuration files for two different archive engines:

```
/data/vacuum/engineconfig.xml
/data/cooling/engineconfig.xml
```

Alternatively, we could have created only a single configuration file, containing one “Vacuum” and one “Cooling” group of channels. But subsystems are often assigned to different engineers, so it is likely that we received these two configuration files from the respective subsystem engineer and it is easiest to keep them separate.

7.1 Engines and Sub-Archives

By running ArchiveEngines in those two directories, we will get two sub archives, that is: An index file and one or more data files will be generated in /data/vacuum and /data/cooling. In addition, we might want to stop and restart the ArchiveEngines periodically, let’s say each day. So even if one engine crashes, the worst that can happen is that we loose data for one subsystem and one day. As a result, we will now end up with multiple sub-archives. Their indices could be this:

```
/data/vacuum/2004/02_19/index
/data/vacuum/2004/02_20/index
/data/cooling/2004/02_19/index
/data/cooling/2004/02_20/index
```

There will also be data files like /data/vacuum/2004/02_19/20040219, but for the most part we can identify a sub-archive via its index file, so we listed only those. For data retrieval, we can invoke ArchiveExport with the path to any of

the four index files. We can also list all four index files in the configuration for our network data server. This is, however, inconvenient because we will only see data for one day of one subsystem at a time.

7.2 Master Indices

The Index Tool allows creation of a master index that covers more than one sub archive. For example, we can create these two configuration files for the index tool, either manually or with the help of make_indexconfig.pl:

```
/data/vacuum/indexconfig.xml:
    Lists 2004/02_19/index and 2004/02_20/index
/data/cooling/indexconfig.xml:
    Lists 2004/02_19/index and 2004/02_20/index
```

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 out data. By creating "/data/in-

dexconfig.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:

```
/data/indexconfig.xml: Lists
    vacuum/2004/02_19/index
    vacuum/2004/02_20/index
    cooling/2004/02_19/index
    cooling/2004/02_20/index
```

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

7.3 Automatization

The ArchiveDaemon program will automate most of what we described so far. By creating a file "/data/ArchiveDaemon.xml" as follows and running the archive daemon in "/data", the daemon will

1. Start an ArchiveEngine in "/data/vacuum" that writes to a daily sub-archive like "/data/vacuum/2004/02_19/index". Same for a second engine in "/data/-cooling" ...
2. Stop each ArchiveEngine at 02:00 AM and restart it in a new subdirectory.
3. Generate or update "/data/indexconfig.xml" whenever a new sub-archive is created for the vacuum or cooling data.
4. Periodically run ArchiveIndexTool on "/data/indexconfig.xml", generating or updating "/data/master_index".
5. Provide a web page that lists the status of the two archive engines.

This is the example ArchiveDaemon.xml:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE engines SYSTEM "ArchiveDaemon.dtd">
<engines>
  <engine>
    <desc>Vacuum</desc>          <port>4813</port>
    <config>/data/vacuum/engineconfig.xml</config>
    <daily>02:00</daily>
  </engine>
  <engine>
    <desc>Cooling</desc>         <port>4814</port>
    <config>/data/cooling/engineconfig.xml</config>
    <daily>02:00</daily>
  </engine>
</engines>
```

7.4 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 /data/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.

At this time, there is no better tool nor a wrapper around the ArchiveDataTool available, so a typical monthly data management run will involve about 30 invocations of the ArchiveDataTool where one needs to carefully adjust the sub archive paths, start times and end times to suit the current month and year.

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 /data/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.

Chapter 8

Setup, Installation

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: config 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 Tools/ArchiverConfig.h to suit your needs. In addition, 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.

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, it ran into a compile-time error that could be fixed by un-commenting "using namespace std;" in the header file which reported the error.

8.1.2 Xerces XML Library

The Xerces library is used to parse the XML configuration files of the ArchiveEngine, 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 vzoxf xerces-c-current.tar.gz
cd xerces-c-src2_4_0
export XERCESCROOT=`pwd`
cd $XERCESCROOT/src/xercesc
autoconf
./runConfigure -plinux -cgcc -xg++\
               -minmem -nsocket \
               -tnative -rpthread \
                -P/usr/local
make
su
make install
```

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

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 32.

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 5.1, ArchiveEngine config. in 3.1, ArchiveDaemon in 4.2, ArchiveIndexTool config. in 6.1). 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 valiated 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 “/archiver/dtd”. Then you can refer to the DTD like this:

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

   This might still require copies of the DTD on multiple computers.

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">
```

Chapter 9

Common Errors and Questions

The following explains error messages and commonly asked questions.

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?

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

There are two main 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. When you look at the status of the channel, those cases might reveil themselves by status values like "Disconnected" or "Archive Off". 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 7.

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: When you receive time-stamp related warnings, it is too late. You need to have the clocks of all CA servers properly configured (also see page 2).

Found an existing lock file 'archive_active.lck'

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.

'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.

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.

Found an existing lock file 'indextool_active.lck'

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

10.1 Directory Files

In case you have existing sub-archives that use the Directory files, refer to the ArchiveDataTool for converting them into nidex files.

10.2 Archive Engine Configurations

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

```
USAGE: ConvertEngineConfig [-d DTD] old-config new-config

This tool reads an old-type ArchiveEngine ASCII config.
file and converts it into the current XML config file.
```

Chapter 11

Changes

This chapter describes the version numbers and changes since the beginning of the R3.14 port. New ”versions” are marked by some additional functionality. The ”patches” are then used to debug those new things.

- In the works — Version 2.1.1:
  Mostly working on ArchiveEngine: Add scanning back in. XML-RPC Data Server undergoes more testing, Java Archive client begins to be useful. 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 ChannelInfo 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. 43

[5] Marty Kraimer et al: *IOC Application Developer's Guide R3.14.4*, http://www.aps.anl.gov/epics/.

Index