A Framework For Back-up and Restore Under The Experimental Physics And Industrial Control System

Nicholas T. Karonis[1]

December 1992

1 Introduction

In this paper we introduce the requirements of a back-up and restore system under the context of the *Experimental Physics and Industrial Control System (EPICS)*. Following that, we present a framework that could serve as a guideline in implementing a system that meets those requirements.

1.1 Requirements

EPICS is a system that allows one to design and implement a controls system. At its foundation, i.e., the level closest to the devices being controlled, are autonomous computers, each called an *Input/Output Controller* or *IOC*. In EPICS, devices controlled by an IOC are represented by software entities called *process variables*. All devices are monitored/controlled by reading/writing values from/to their associated process variables.

Under this schema, distributing processing over a number of IOCs and representing devices with process variables, there are a variety of ways one can view or group the information in the control system. Two of the more common groupings are by IOC (location) and by devices (function).

Simply stated, we require a system capable of restoring the state of the machine, in our case the Advanced Photon Source, to a known desired state from somewhere in the past. To that end, we propose a framework which describes a system that periodically records and preserves the values of key process variables so that later on, those values can be written to the machine in an attempt to restore it to that same state.

One of the more powerful notions that must be preserved in any system that solves this problem is the independence between the specification of what is monitored and the specification of what is written. In other words, grouping process variables for monitoring must be kept independent of the number of different ways to group process variables (e.g., by IOC, by device, etc.) when they are written.

1.2 Assumptions

In formulating this framework it became necessary to impose two new requirements onto EPICS that were not present at the time this document was prepared. They are 1) that each IOC has a unique identifier and 2) that the location, with respect to IOC, of every process variable we are interested in monitoring be known to the system. These two requirements stem almost entirely from the need for the system to have the ability to restore process variables on an IOC basis.

1.3 Time Doctrine

Time is a fundamental component of the system and therefore is worthy of some discussion. A complete description regarding the system's view of time is given in the sections that follow. However, the semantics associated with a user's request for information from a particular time in the past are discussed here.

Although a common view of time is that it progresses forward on a continuum, the system views the passage of time as a series of discrete events. Therefore, when a user requests a restoration from a given time

---

1. N.T. Karonis is a member of the Controls Group, Accelerator Systems Division, Advanced Photon Source, Argonne National Laboratory.

t, there is the possibility that t lies between two discrete time intervals. This discrepancy must be addressed. We do so with the following doctrine. The values associated with time t may only include information known at or before t, never after t.

2 Versions

One of the primary functions of the system is to periodically record, i.e., monitor, the values of certain process variables. It is therefore necessary to describe to the system which process variables we are interested in recording. For purposes associated with restoring IOCs, it is also necessary to identify which IOC each selected process variable resides on. Specifying process variables as well as identifying which IOC they reside on is done using an ordered pair, *(pvar, iocid)*.

We define a *version* as an ordered finite set of *(pvar, iocid)* pairs. Each element of a version represents a process variable that we wish to monitor.

$$version = \{(pvar, iocid)_1, \ldots, (pvar, iocid)_n\} \qquad \text{(EQ 1)}$$

Occasionally, it may be necessary to change the set of process variables that we are interested in monitoring. This may be caused by the addition or deletion of process variables from a number of IOCs, moving a set of process variables from one IOC to another, or re-booting an IOC with a new database. Over time, there may be more than one version, however it is also important to note that, at any instant, there is at most one version designated as the current version.

To keep track of all the versions ever known to the system, we introduce the *versions set.* Each element in the version set has a version identifier and the finite ordered list of *(pvar, iocid)* pairs. Each version identifier is unique with respect to the system for all time.

$$versionset = \{(versionid, version), \ldots, (versionid, version)\} \qquad \text{(EQ 2)}$$

2.1 Version Bitmasks

Each version defines the structure of a *version bitmask*. The version bitmask has a bit for each element of its associated version. Because a version is an ordered set, each one uniquely defines the *structure* of a version bitmask, i.e., the width of the bitmask is equal to the cardinality of its associated version and its bit order is defined by the order of the process variables in its version.

3 Snapshots

A single version may exist for a long time. As operations stabilize, so too will the versions. Recall that a version is used to represent a set of process variables we wish to monitor. This is done by initially logging the values of all the process variables in the version, and then periodically logging any changes that occurred. Each such log, the initial log as well as the subsequent periodic logs, is called a *snapshot*. Therefore in the general case, a single version will have many snapshots. A snapshot is a set of *(pvar, val)* pairs.

$$snapshot = \{(pvar, val), \ldots, (pvar, val)\} \qquad \text{(EQ 3)}$$

Each snapshot has its own unique identifier with respect to its version. The set of valid snapshot identifiers must be chosen such that a linear order[1] may be imposed over them, e.g., the set of positive integers.

1. A partial order ∇ on set A is a *linear order* if and only if $a \nabla b$ or $b \nabla a$ for every $(a, b) \in A$.

3.1 Version Bitmask Sets

Each snapshot is taken under the context of a version. A snapshot and its associated version defines the *value* of a version bitmask.

Consider a snapshot *s* taken under the context of version *v*. If s is the first snapshot under *v*, it contains values for those process variables that have reported for the first time. If it is not the first snapshot, it contains those values for those process variables whose values changed since the last snapshot or process variables that have reported for the first time. In either case, each process variable that appears in *s* has its associated bit set in the version bitmask associated with *v* and all other bits cleared.

Associated with each version is a *version bitmask set*. The name of the set is constructed using the version identifier. Each element of the version bitmask set is a snapshot identifier with its associated bitmask.

(EQ 4)

$$versionbitmaskset = \{(snapshotid, bm), \ldots, (snapshotid, bm)\}$$

3.2 Time Set

The system requires a representation of time. Such a representation must assure a linear order may be imposed over its values and that $t_1 < t_2$ if and only if t_1 is before t_2 in time.

Consider a snapshot *s* with identifier s_{id} under version *v* with identifier v_{id} taken at time *t*. As pointed out above, the appropriate version bitmask will be constructed and added to the proper version bitmask set. Additionally, for each new snapshot, an element is added to a set we introduce here, the *time set*. Each element of the time set reflects a single snapshot, i.e., the time *t*, the snapshot identifier s_{id}, and the version identifier v_{id}.

(EQ 5)

$$timeset = \{ (t, versionid, snapshotid), \ldots, (t, versionid, snapshotid) \}$$

3.3 Value Sets

Associated with each snapshot are a set of process variables and their associated values, that is, the process variables and the values that have reported since the last snapshot with respect to a version. *Value sets* are sets of *(pvar, val)* pairs where all the process variables reside on the same IOC at the time of the snapshot.

(EQ 6)

$$valueset = \{(pvar, val), \ldots, (pvar, val)\}$$

Consider a snapshot *s* with identifier s_{id} under version *v* with identifier v_{id}. Recall that each process variable in *s* has an associated IOC. In fact, *s* quite probably contains values for process variables found on a number of different IOCs. For each IOC, with identifier ioc_{id}, associated with the values in *s* a new value set is created. Its name is constructed using the version identifier v_{id}, the snapshot identifier s_{id}, and the IOC identifier ioc_{id}. All the *(pvar, val)* pairs associated with IOC ioc_{id} are placed into the set. In other words, a single snapshot set containing values from *n* IOCs would generate *n* value sets.

4 Operations

In operation, the system is responsible for three basic functions; taking snapshots, and using those snapshots to restore selected process variables or selected IOCs. The following is a general description of how the information described in the previous sections is used to accomplish these functions.

4.1 Initialization

At any time during operation, the system will be aware of exactly one time set. This time set is initialized as empty. Additionally, the system will be aware of exactly one version set. This version set should be initialized with the first version of process variables and their associated IOCs. This first version is called the *current version.* At any point in time, there will be one version, an element of the version set, that is denoted as the current version.

As there is always a current version, its version identifier is used to identify a current version bitmask set. It is initialized as empty.

Initialization of the system concludes by noting there are no value sets defined.

4.2 Taking A Snapshot

Much of the operations associated with taking a snapshot have been discussed in the previous sections. In this section we examine the operations formally.

The current version identifies those process variables we wish to monitor. Periodically, a snapshot is taken with respect to that current version. A snapshot is a set of process variables with their values. Those process variables in the snapshot are whose values have changed since the last snapshot under this version or who have reported their values for the first time under this version.

Consider a snapshot s under current version v with version identifier v_{id} taken at time t. Snapshot s is issued a snapshot identifier that is unique and greater than all other snapshot identifiers with respect to the current version, s_{id}.

After generating a snapshot identifier, we must create all the appropriate value sets as well as set the value of a bitmask for this snapshot. A bitmask *bm* is constructed based on version v and initialized by clearing all the bits.The snapshot s is a set of process variables and value pairs, *(pvar, val)*. Each process variable in s is related to exactly one IOC. Therefore, s and v define a set *IOC*, a set of IOC identifiers,

$$IOC = \{iocid | \exists (pvar, val, iocid)\ [(pvar, val) \in s \wedge (pvar, iocid) \in v]\} \tag{EQ 7}$$

Note $v = \varnothing \Rightarrow s = \varnothing$ that and that $s = \varnothing \Leftrightarrow IOC = \varnothing$.

Each $iocid \in IOC$ represents a new value set. The name of the value set is constructed using the version identifier v_{id}, the snapshot identifier s_{id}, and the IOC identifier *iocid*. The newly created value set is initialized empty and constructed as follows.

Given the current version v and *iocid*, construct a process variable index set, *PVS*, a set of *(pvar, idx)* pairs. These represent all the process variables and their associated indices from the current version that reside on the IOC identified by *iocid*.

$$PVS = \{ (pvar, idx) | (pvar, iocid)_{idx} \in v\} \tag{EQ 8}$$

Then, for each $(pvar, val) \in s$ such that $(pvar, idx) \in PVS$, add *(pvar, val)* to the value set and set bit *idx* in *bm*.

Each value set, one for each IOC represented in this snapshot, is constructed this way. After all value sets have been constructed, the bitmask for this snapshot has the proper bits set and is ready to be added to the version bitmask set identified by v_{id}. This simply requires adding the ordered pair (s_{id}, bm) to that version bitmask set.

The third and final step in taking a snapshot is updating the time set. This is done by adding (t, v_{id}, s_{id}) to the time set.

4.3 Retrieving By Process Variable Name

This section describes the process of retrieving the value of a single process variable at a particular time. The extension to sets of process variables representing devices is straightforward.

Consider the problem of retrieving the *value* of a process variable *pvar* at some time t. The first step is to identify the version and snapshot that is closest to, but not after, time t. Given our representation of time, we need to search the time set for a (t', v_{val}, s_{start}) such that

1) t' represents a snapshot at or before t,

$$(t', v_{val}, s_{start}) \in timeset \wedge t' \leq t \tag{EQ 9}$$

and 2) t' is the closest such time to t.

$$\forall\ ((t'', v, s) \in timeset)\ [t'' \leq t \Rightarrow t'' \leq t'] \tag{EQ 10}$$

Using v_{val}, we identify the appropriate version bitmask set *bmset* and retrieve the version v from the version set. Now having v, we identify the *(iocid, idx)* of *pvar* in v. The next step is to find the most recent snapshot under this version that has a value set for this process variable. That is, we need to search *bmset* for a (s_{val}, bm) such that

1) s_{val} is the identifier a snapshot taken at or before s_{start} where bit *idx* is set in *bm*,

$$(s_{val}, bm) \in bmset \wedge s_{val} \leq s_{start} \wedge bm\,[idx] = 1 \tag{EQ 11}$$

and 2) s_{val} is the closest such identifier to s_{start}.

$$\forall\ ((s,b) \in bmset)\ [\ (s \leq s_{start} \wedge b\,[idx] = 1) \Rightarrow s \leq s_{val}] \tag{EQ 12}$$

Using v_{val}, s_{val}, and *iocid* we identify a value set. It is that value set that has the *value* of the process variable *pvar* that is closest to, but not after, time t.

4.4 Retrieving By IOC Identifier

This section describes the process of retrieving all the values of a single IOC at a particular time.

Consider the problem of constructing a *value set*, a set of *(pvar, val)* pairs, of all the process variables on an IOC identified by *iocid* at some time t. The first step is to initialize the *value set* as empty and to identify the version and snapshot that is closest to, but not after, time t. This is done by searching the time set for (t', v_{val}, s_{start}) in the same manner described in the previous section.

The version identifier v_{val} identifies version v and the version bitmask set *bmset*, while the snapshot identifier s_{start} identifies snapshot s. Using the same technique described in Section 4.2 we construct a process variable index set, *PVS*, a set of *(pvar, idx)* pairs using v and *iocid*.

For each $(pvar, idx) \in PVS$, we need to search *bmset* for the snapshot identifier s_{val} that represents the snapshot closest to, but not after, s_{start} under version v. This is done in the same manner as described in the previous section.

Using v_{val}, s_{val}, and *iocid* we identify a value set. It is that value set that has the value *val* of the process variable *pvar* that is closest to, but not after, time t. The ordered pair *(pvar, val)* is then added to the *value set*.

Once each of the elements of *PVS* are processed in this way, the *value set* contains all the *(pvar, val)* pairs of all the process variables on the IOC identified by *iocid* at some time closest to, but not after, time t.

5 Decay

Up until now, it has been assumed that snapshots would be preserved indefinitely. In practice, this turns out to be impractical due to storage requirements, and fortunately, unnecessary. Hence we introduce *decay*. Simply stated, decay is the process of eliminating all traces of a snapshot. Choosing a snapshot is presumably done via some heuristic, perhaps one based on the passage of time. Independent of how the decision is made to choose a snapshot, the system must be capable of removing all traces of the snapshot while maintaining the consistency of its information and minimizing the loss of additional information.

5.1 Merging

Snapshots are deleted from the system through the process of *merging*. That is, taking all the information associated with a snapshot and merging it into one of its adjacent, with respect to time, snapshots. Given this, we need to decide which way to merge, forward or backward. Consider three consecutive snapshots s_1, s_2, s_3 under version v at times t_1, t_2, and t_3, respectively, where $t_1 < t_2 < t_3$. We wish to remove snapshot s_2.

First we inspect merging backward. In merging backward we take all the information associated with s_2 and merge it with s_1 and then remove all traces of s_2. Later a request is made to retrieve information at a time t where $t_1 < t < t_2$. In mapping back to the closest time to, but not after, t we encounter t_1. However, since we merged the information from s_2 back into s_1, s_1 now reflects information associated with time $t_2 > t$. This is a clear violation of the time doctrine we presented in Section 1.3, therefore merging backward is unacceptable.

Next we inspect merging forward. In merging forward we take all the information associated with s_2 and merge it with s_3 and then remove all traces of s_2. Later a request is made to retrieve information at a time t where $t_2 < t < t_3$. In mapping back to the closest time to, but not after, t we encounter t_1. Because $t > t_2$, we would expect the information associated with s_2 to be reflected in a request to retrieve information at time t. However, because we merged s_2 forward to s_3, its information is not found in s_1. In effect we have lost information.

The loss of information resulting from merging forward is viewed as significantly more acceptable than the violation of the time doctrine resulting from merging backward. Therefore, when removing snapshots from the system, their information will always be merged forward. Further, all merging is done under the context of a single version, that is, snapshots will never be merged across versions.

5.2 Deleting Snapshots From A Version

Recall that snapshots always exist in the context of a version, and as such, it exists as a member of an ordered sequence of snapshots with respect to that version. In deleting a snapshot from the system, we are in effect deleting a snapshot from its associated version. There are three distinct cases we must consider with respect to the snapshot we are deleting; 1) it is the last remaining one in this version, 2) it is not the last remaining one in the version, but it is the last one in the sequence, and 3) it is not the last remaining one in the version and it is not the last one in the sequence. Each of these cases is discussed below given that we wish to delete snapshot s with identifier s_{id} under version v with identifier v_{id}.

5.2.1 Case I - Deleting The Last Snapshot In A Version

In this case we are asked to delete the last remaining snapshot associated with version v. We must not only remove all traces of the snapshot, but remove all traces of the version also.

First we remove all the value sets associated with this snapshot. We consider the set of valid IOC identifiers, IOC. For each $iocid \in IOC$, we delete the value set named by the version identifier v_{id}, the snapshot identifier s_{id}, and the IOC identifier *iocid,* if it exists. Next we remove the version bitmask set identified by the version identifier v_{id}. We remove the single element from the time set associated with the version identifier v_{id} and the snapshot identifier s_{id}. Finally, we remove (v_{id}, v) from the version set.

5.2.2 Case II - Deleting A Snapshot Amongst Others That Has No Successor

In this case we are asked to delete a snapshot that is not the last in *v*, but has no successors to merge forward into. Here we simply remove all traces of the snapshot without merging forward.

The first step is to remove all the value sets associated with this snapshot. This is done in the same manner as described in the previous section. Next, we remove the element associated with the snapshot identifier s_{id} from the version bitmask set identified by the version identifier v_{id}. Finally, we remove the single element from the time set associated with the version identifier v_{id} and the snapshot identifier s_{id}.

5.2.3 Case III - Snapshot A Snapshot Amongst Others That Has A Successor

In this case we are asked to delete a snapshot that is not the last in *v* and that does have a successor under *v* that can be merged forward into. This is the only case where forward merging occurs. In this case we merge the snapshot forward into its successor and then remove all traces of the snapshot in the same manner described in the previous section. Following is a description of the merging forward process.

The first step is to merge the value sets forward. The version identifier v_{id} identifies the version bitmask set *bmset*. Using the snapshot identifier s_{id} we can find (s_{id}, bm_{id}) in *bmset*. We can also find the closest successor to s_{id} in *bmset* (s_{next}, bm_{next}) in *bmset* such that

1) s_{next} is a successor of s_{id},

(EQ 13)

$$(s_{next}, bm_{next}) \in bmset \wedge s_{next} > s_{id}$$

and 2) s_{next} is the closest successor of s_{id}.

(EQ 14)

$$\forall\, ((s,b) \in bmset)\ [s > s_{id} \Rightarrow s \geq s_{next}]$$

Using bm_{id} and *v*, we construct a set of IOC identifiers, *IOC*, where each IOC represented in *s* has its IOC identifier in *IOC*.

(EQ 15)

$$IOC = \{ iocid | (pvar, iocid)_{idx} \in v \wedge bm_{id}[idx] = 1 \}$$

Each element of *IOC* represents a value set we need to merge forward. For each $iocid \in IOC$, we merge the value set named by the version identifier v_{id}, the snapshot identifier s_{id}, and the IOC identifier *iocid* in the following way.

Let vs_{id} be the name of the value set named by the version identifier v_{id}, the snapshot identifier s_{id}, and the IOC identifier *iocid* and let vs_{next} be the name of the value set named by the version identifier v_{next}, the snapshot identifier s_{id}, and the IOC identifier *iocid*.

If vs_{next} does not exist, simply rename vs_{id} to vs_{next}. If vs_{next} does exist, we must add those *(pvar, val)* pairs that exist in vs_{id} that do not already exist in vs_{next}.

(EQ 16)

$$\{ (pvar, val) | \ ((pvar, val) \in vs_{id} \wedge \neg \exists v\, [(pvar, v) \in vs_{next}]) \}$$

Finally, the bits in bm_{id} must be reflected in bm_{next}. This is done by re-assigning $bm_{next} = \langle bm_{next} | bm_{id} \rangle$ and replacing the new (s_{next}, bm_{next}) into *bmset*. This case is concluded by removing all traces of the snapshot in the same manner as described in the previous section.

6 Commentary

We believe the framework we have presented meets the requirements of the system and we conclude with the following observations.

Through the notion of versions, the framework allows the users to change the set of process variables they are interested in monitoring over time. Additionally, taking snapshots and making retrievals are independent mechanisms. This independence allows us to run as many snapshot and retrieval processes as necessary. Also, it allows us to vary the periodic rate at which snapshots are taken as well as varying the decay heuristic without impacting the operations of the retrieval mechanism.

For purposes of optimization with respect to its implementation, we have introduced bitmasks. This is done under the assumption that searching a single bitmask set under a version for the value of a specific process variable is significantly more efficient than conducting a systematic search over a number of value sets.