# autosave v4.2

Autosave automatically saves the values of EPICS process variables (PVs) to files on a server, and restores those values when the IOC is rebooted. The original author is Bob Dalesio; I made some improvements; Frank Lenkszus made some more improvements, which I folded into the version I've been maintaining. A bunch of people contributed to getting the software running on PPC hardware, including Ron Sluiter, Andrew Johnson, and Pete Jemian (APS), Markus Janousch and David Maden (SLS), and I'm not sure who else.

Autosave is a two-part operation: run-time save, and boot-time restore. The run-time part (save_restore.c) is started by commands in the IOC startup file, and persists while the IOC is running. Its primary job is to save PV values to files on a server, but it also supports manual restore and other management operations. The boot-time part (dbrestore.c) is invoked during iocInit, via an EPICS initHook. It restores PV values from files written by the run-time part, and does not persist after iocInit returns.

In addition to the autosave software, the autosave module contains a client program, asVerify, to compare written autosave files with current PV values. This program can also write an autosave file from which PV values can be restored.

## Module contents

### asApp/src

save_restore.c
: saves PV values in files on a file server according to preset rules.

dbrestore.c
: restore PV values at boot time, using dbStaticLib

initHooks.c
: call restore routines at the correct time during boot.

fGetDateStr.c
: Frank Lenkszus' date-string routines

save_restore.h, fGetDateStr.h
: headers

asVerify.c
: Client-side tool to compare autosaved file with current PV values. Can also write an autosave file.

### asApp/Db

auto_settings.req, auto_positions.req
: Sample request files

save_restoreStatus.db
: database containing records save_restore uses to report status.

SR_test.db
: Test database for autosave and asVerify.

### asApp/op/adl

save_restoreStatus*.adl, save_restoreStatusLegend.adl, save_restoreStatus_more.adl, save_restoreStatus_tiny.adl, SR_X_Status.adl
: MEDM displays of save_restore status.

---

## How to use autosave

This software can be used in many different ways. I'll describe what you have to do to use it as it's commonly used at APS beamlines to save PV values periodically, and restore them on reboot. A complete example of how autosave is used at APS can be found in the synApps xxx module. The relevant files in that module are the following:

xxx/configure/RELEASE
  look for "AUTOSAVE"
xxx/xxxApp/src/Makefile
  look for "autosave"
xxx/xxxApp/src/xxxCommonInclude.dbd
  look for "asSupport.dbd"
xxx/iocBoot/iocvxWorks/save_restore.cmd
  the whole file
xxx/iocBoot/iocvxWorks/auto_positions.req
  the whole file
xxx/iocBoot/iocvxWorks/auto_settings.req
  the whole file
xxx/iocBoot/iocvxWorks/st.cmd
  look for "save_restore.cmd" before the call to iocInit, and "create_monitor_set" after the call to iocInit.
xxx/iocBoot/iocvxWorks/autosave
  a directory to hold autosave .sav files

Here's a step-by-step program for deploying autosave. Some of the steps are optional:

**1. Build (required)**

Build the module and include the resulting library, libautosave.a, and database-definition file, asSupport.dbd, in an IOC's build. For example, add

```
AUTOSAVE=<path to the autosave module>
```

to xxx/configure/RELEASE, add

```
xxx_LIBS += autosave
```

to xxxApp/src/Makefile, and add

```
include "asSupport.dbd"
```

to iocxxxInclude.dbd.

**2. Write request files (required)**

Create "request" files (e.g., auto_settings.req, auto_positions.req) specifying the PVs whose values you want to save and restore. The save files corresponding to these request files will have the ".req" suffix replaced by ".sav". Request files can include other request files (nested includes are allowed) and macro substitution can be performed on the included files (using William Lupton's macro library), with the following syntax:

```
file <request_file> <macro-substitution_string>
```

e.g.,

```
file motor_settings.req P=xxx:,M=m1
```

I've tried to defend against forseeable variations in syntax, so that include lines with embedded whitespace and/or quotes, macro strings without commas, empty macro strings, and lines with trailing comments will be parsed as one would want. Generally, quotes are ignored, whitespace implies a comma but otherwise is ignored, and everything after the second sequence of non-whitespace characters (i.e., after the file name) and before the (optional) comment character '#' is taken as the macro-substitution string. Macro substitution is performed on the entire line, so it's possible to parameterize names of included files, as well as PV names. It is also possible to define a macro that replaces its target with nothing.

Most synApps modules contain autosave-request files that are intended to be included in an ioc's autosave-request file(s). For example, calc/calcApp/Db/scalcout_settings.req contains a list of the fields one might want to autosave for a single scalcout record. This file is *include*d in calc/calcApp/Db/userStringCalcs10_settings.req, which, in turn, is included in xxx/iocBoot/iocvxWorks/auto_settings.req.

### 3. Set request-file path (optional, recommended)

Specify one or more directories to be searched for request files, using one or more invocations of the function

```
        set_requestfile_path()
```

### 4. Set NFS host (optional, only available on vxWorks)

Specify the NFS host from which save files will be read at restore time, and to which they will be written at save time, by calling the function

```
        save_restoreSet_NFSHost()
```

When autosave manages its own NFS mount, as this command directs it to do, it can fix a stale file handle by dismounting and remounting the file system.

### 5. Use NFS (recommended, only available on vxWorks)

Use NFS, preferably as described above, or by including an nfsMount() command in your startup script. Save_restore is only tested with NFS, though it has in the past and may still work with vxWorks' netDrv (ftp or rsh).

When autosave runs under operating systems other than vxWorks, it simply uses whatever mount the operating system, or a system administrator, has provided.

### 6. Set save-file path (optional, recommended)

Specify the directory in which you want save files to be written, by calling the function

```
        set_savefile_path()
```

in your startup script, before the create_xxx_set() commands, to specify the path to the directory. If you are using NFS (strongly recommended), ensure that the path does not contain symbolic links. In my experience, VxWorks cannot write through a symbolic link. (I don't understand all the ins and outs of this limitation. It may be that symbolic links are OK if they contain absolute path names.)

### 7. Give write permission (required)

Give the IOC write permission to the directory in which the save files are to be written. If you forget this step, save_restore may be able to write save files, but the files will be corrupted because save_restore will not be permitted to change their lengths. Save_restore attempts to detect this condition, but cannot work around it if the file length must increase.

### 8. Set restore conditions (optional, recommended)

Specify which save files are to be restored before record initialization (pass 0) and which are to be restored after record initialization (pass 1), using the commands

```
          set_pass0_restoreFile()
          set_pass1_restoreFile()
```

Place these commands in the startup file before iocInit. If you don't call either of these functions (or if your calls fail completely to specify any restore files) the supplied initHooks routine will attempt to restore the file "auto_positions.sav" before record init, and the file "auto_settings.sav" both before and after record init.

Notes on restore passes:

1. Link fields cannot be restored (by dbStatic calls) after record initialization. If you want save/restore to work for link fields you must specify them in a pass-0 file.
2. Device support code for the motor record uses the value of the field DVAL, restored during pass 0, only if the value read from the hardware is zero. If the value from hardware is nonzero, it is used instead of the restored value.
3. Arrays cannot be restored during pass 0.
4. Scalar PV's which have type DBF_NOACCESS in the .dbd file, and are set to some other DBF type during record initialization, cannot be restored during pass 0.
5. It is not an error to attempt to restore PV's during the wrong pass. The default strategy, implemented with auto_settings.req and auto_positions.req, is to use both passes for everything except PV's that shouldn't be restored in pass 1. (Motor positions aren't restored in pass 1 because doing so would overwrite any values read from the hardware.)

**9. Load initHook routine (required)**

Load a copy of initHooks that calls reboot_restore() to restore saved parameter values. The copy of initHooks included in this distribution is recommended. This will happen automatically if the ioc's executable is built as described above.

**10. Select save-file options (optional, recommended)**

Tell save_restore to writed dated backup files. At boot time, the restore software writes a backup copy of the ".sav" file from which it restored PV's. This file can either be named xxx.sav.bu, and be rewritten every reboot, or named xxx.sav_YYMMDD-HHMMSS, where "YY..." is a date. Dated backups are not overwritten. If you want dated backup files, put the following line in your st.cmd file before the call toiocInit():

```
          save_restoreSet_DatedBackupFiles(1)
```

Note: If a save file is restored in both pass 0 and pass 1, the boot-backup file will be written only during pass 0.

Tell save_restore to save sequence files. The commands:

```
          save_restoreSet_NumSeqFiles(3)
          save_restoreSet_SeqPeriodInSeconds(600)
```

will cause save_restore to maintain three copies of each .sav file, at ten-minute intervals.

**11. Start the save process (required)**

Invoke the "save" part of this software as part of the EPICS startup sequence, by calling create_XXX_set() -- e.g., adding lines of the form

```
        create_monitor_set("auto_positions.req", 5, "P=xxx:")
        create_monitor_set("auto_settings.req", 30, "P=xxx:")
```

to your EPICS startup file after iocInit. You can call the files anything you want, but note that the file names "auto_positions" and "auto_settings" receive special treatment: they are default restore-file names specified by initHooks() if set_pass<n>_restoreFile() were never called. The third argument to create_monitor_set() is a macro-substitution string, as described above in the discussion of request files (step 2). If supplied, this macro-substitution string supplements any macro strings supplied in include-file directives of request files read for this save set.

For each "create_monitor_set(<name>.req, <time>, <macro>)" command, the save_restore process will write the files <name>.sav and <name>.savB every <time> seconds, if any of the PVs named in the file <name>.req have changed value since the last write. Other create_xxx_set() commands do the same thing, but with different conditions triggering the save operation.

Note that in versions prior to 2.7, create_monitor_set() used an argument of type double to specify the period (in seconds). This doesn't work on the PPC, so the arguments for this and similar functions were changed to int.

If your IOC takes a really long time to boot, it's possible the PVs you want to save will not have the correct values when the save_restore task first looks at them. (If you are restoring lots of long arrays, this is even more likely.) Under vxWorks, you can avoid this by putting a

```
        taskDelay(<number_of_60_Hz_clock_ticks>)
```

before create_monitor_set().

### 12. Maintain save files

Autosave is not completely bulletproof. Most APS beamlines have at least one autosave related problem every year. If autosave fails, you might be able to detect it, or work around it, using asVerify. This client-side program reads an autosave .sav file and independently verifies that the values it contains agree with current PV values. The program can also be used to write a .sav file, given a list of PVs. The list of PVs is normally an autosave .sav file, but a file containing nothing but PV names, one per line, would also work. Here's asVerify's command line:

```
usage: asVerify [-vrd] <autosave_file>
         -v (verbose) causes all PV's to be printed out
            Otherwise, only PV's whose values differ are printed.
         -r (restore_file) causes a restore file named
            '<autosave_file>.asVerify' to be written.
         -d (debug) increment debug level by one.
         -rv (or -vr) does both
examples:
    asVerify auto_settings.sav
        (reports only PVs whose values differ from saved values)
    asVerify -v auto_settings.sav
        (reports all PVs, marking differences with '***'.)
    asVerify -vr auto_settings.sav
        (reports all PVs, and writes a restore file.)
    asVerify auto_settings.sav
    caput <myStatusPV> $?
        (writes number of differences found to a PV.)

NOTE: For the purpose of writing a restore file, you can specify a .req
file (or any file that contains PV names, one per line) instead of a
.sav file.  However, this program will misunderstand any 'file' commands
that occur in a .req file.  (It will look for a PV named 'file'.)
```

Note that asVerify cannot read an autosave request file; it will understand any PV names contained in the file, but it cannot parse the "file" command, perform macro substitutions, or include other request files.

---

## About save files

PV values in a save file have been converted to strings, in most cases simply by having been read as strings -- e.g., `ca_get(DBR_STRING,...)`. Most data types are read using channel access and written using dbStaticLib calls. Four data types get special attention:

double
: read as double, converted to string with the format "%.14g". (Otherwise, the record's .PREC field would limit the precision.)

float
: Same as double, but the formst is "%.7g".

menu and enum
: Channel access does not distinguish these types. dbStaticLib cannot write enums as strings, because enum strings may not be defined until after record initialization--long after the values must have been restored.

arrays of any kind
: Arrays are read and written using database access. Channel access cannot read only the defined portion of an array, dbStaticLib cannot write an array. (However, asVerify uses channel access to read arrays.)

Here is a sample save file. Characters in blue are documentation comments, and are not part of the file:

```
# save/restore V4.9     Automatically generated - DO NOT MODIFY - 060720-154526
! 1 channel(s) not connected - or not all gets were successful
xxx:SR_ao.DISP 0 (uchar)
xxx:SR_ao.PREC 1 (short)
xxx:SR_bo.IVOV 2 (ushort)
xxx:SR_ao.SCAN 3 (enum - saved/restored as a short)
xxx:SR_ao.VAL 4.1234567890123 (double, printed with format "%.14g")
xxx:SR_scaler.RATE 1.234568 (float, printed with format "%.7g")
xxx:SR_ao.DESC description (string)
xxx:SR_ao.OUT xxx:SR_bo.VAL NPP NMS (link)
xxx:SR_ao.RVAL 4 (long)
xxx:SR_bi.SVAL 2 (ulong)
#i_dont_exist.VAL Search Issued (no such PV)
xxx:SR_char_array @array@ { "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" }
xxx:SR_double_array @array@ { "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" }
xxx:SR_float_array @array@ { "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" }
xxx:SR_long_array @array@ { "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" }
xxx:SR_short_array @array@ { "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" }
xxx:SR_string_array @array@ { "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" }
xxx:SR_uchar_array @array@ { "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" }
xxx:SR_ulong_array @array@ { "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" }
xxx:SR_ushort_array @array@ { "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" }
<END>
```

Save files are not intended to be edited manually. If you, nevertheless, do edit a save file, you must end it with the text

```
<END>
```

followed by one or two arbitrary characters (normally '\n' or '\r\n'). If the file does not end with this text, reboot_restore() will assume the crate crashed while the file was being written, or that some other bad thing happened, and will not use the file. Once a save file has been created successfully, save_restore will not overwrite the file unless a good ".savB" backup file exists. Similarly, it will not overwrite the ".savB" file unless the save file was successfully written.

You can comment out lines in a .sav file by beginning them with '#'.

---

## User-callable functions

`int manual_save(char *request_file)`
: If a manual save set for the request file `request_file` was created with create_manual_set(), this command will cause current PV values to be saved.

`int set_savefile_name(char *request_file, char *save_file)`
: If a save set has already been created for the request file, this function will change the save file name.

`int create_periodic_set(char *request_file, int period, char *macrostring)`
: Create a save set for the request file. The save file will be written every `period` seconds.

  This function can be called at any time after iocInit.

`int create_triggered_set(char *request_file, char *trigger_channel, char *macrostring)`
: Create a save set for the request file. The save file will be written whenever the PV specified by `trigger_channel` is posted. Normally this occurs when the PV's value changes.

  This function can be called at any time after iocInit.

`int create_monitor_set(char *request_file, int period, char *macrostring)`
: Create a save set for the request file. The save file will be written every `period` seconds, if any PV in the save set was posted (changed value) since the last write.

  This function can be called at any time after iocInit.

`int create_manual_set(char *request_file, char *macrostring)`
: Create a save set for the request file. The save file will be written when the function `manual_save()` is called with the same request-file name.

  This function can be called at any time after iocInit.

`int fdbrestore(char *save_file)`
: If `save_file` refers to a save set that exists in memory, then PV's in the save set will be restored from values in memory. Otherwise, this functions restores the PV's in <saveRestorePath>/<save_file> and creates a new backup file "<saveRestorePath>/<save_file>.bu". The effect probably will not be the same as a boot-time restore, because caput() calls are used instead of static database access dbPutX() calls. Record processing will result from caput()'s to inherently process- passive fields.

  This function can be called at any time after iocInit.

`int fdbrestoreX(char *save_file)`
: This function restores from the file <saveRestorePath>/<save_file>, which can look just like a save file, but which needn't end in <END>?. No backup file will be written. The effect probably will not be the same as a boot-time restore, because caput() calls are used instead of static database access dbPutX() calls. Record processing will result from caput()'s to inherently process-passive fields.

  This function can be called at any time after iocInit.

`void save_restoreShow(int verbose)`
: List all the save sets currently being managed by the save_restore task. If (verbose != 0), lists the PV's as well.

This function can be called at any time after iocInit.

`int set_requestfile_path(char *path, char *pathsub)`
: Called before create_xxx_set(), this function specifies the path to be prepended to request-file names. `pathsub`, if present, will be appended to `path`, if present, with a separating '/' whether or not `path` ends or `pathsub` begins with '/'. If the result does not end in '/', one will be appended to it.

You can specify several directories to be searched for request files by calling this routine several times. Directories will be searched in the order in which the `set_requestfile_path()` calls were made. If you never call the routine, the crate's current working directory will be searched. If you ever call it, the current directory ("./") will be searched only if you've asked for it explicitly.

`int set_savefile_path(char *path, char *pathsub)`
: Called before iocInit(), this function specifies the path to be prepended to save-file and restore-file names. `pathsub`, if present, will be appended to `path`, if present, with a separating '/' whether or not ends or `pathsub` begins with '/'. If the result does not end in '/', one will be appended to it.

If save_restore is managing its own NFS mount, this function specifies the mount point, and calling it will result in an NFS mount if all other requirements have already been met. If a valid NFS mount already exists, the file system will be dismounted and then mounted with the new path name. This function can be called at any time.

`int set_saveTask_priority(int priority)`
: Set the priority of the save_restore task.

`int remove_data_set(char *request_file)`
: If a save set has been created for `request_file`, this function will delete it.

`int reload_periodic_set(char *request_file, int period, char *macrostring)`
: This function allows you to change the PV's and the period associated with a save set created by `create_periodic_set()`.

`int reload_triggered_set(char *request_file, char *trigger_channel, char *macrostring)`
: This function allows you to change the PV's and the trigger channel associated with a save set created by `create_triggered_set()`.

`int reload_monitor_set(char * request_file, int period, char *macrostring)`
: This function allows you to change the PV's and the period associated with a save set created by `create_monitor_set()`.

`int reload_manual_set(char * request_file, char *macrostring)`
: This function allows you to change the PV's associated with a save set created by `create_manual_set()`.

Note: Don't get too ambitious with the remove/reload functions. You have to wait for one to finish completely (the save_restore task must get through its service loop) before executing another. If you call one before the previous function is completely finished, I don't know what will happen.

`int reboot_restore(char *save_file, initHookState init_state)`
: This should only be called from initHooks because it can only function correctly if called at particular times during iocInit.

`int set_pass0_restoreFile(char *save_file)`
: This function specifies a save file to be restored during iocInit, before record initialization. Up to eight files can be specified using calls to this function.

`int set_pass1_restoreFile(char *save_file)`
: This function specifies a save file to be restored during iocInit, after record initialization. Up to eight files can be specified using calls to this function.

`void save_restoreSet_Debug(int debug_level)`
Sets the value `(int) save_restoreDebug` (initially 0). Increase to get more informational messages printed to the console.

This function can be called at any time.

`void save_restoreSet_IncompleteSetsOk(int ok)`
Sets the value of `(int) save_restoreIncompleteSetsOk` (initially 1). If set to zero, save files will not be restored at boot time unless they are perfect, and they will not be overwritten at save time unless a valid CA connection and value exists for every PV in the list.

This function can be called at any time.

`void save_restoreSet_NumSeqFiles(int numSeqFiles)`
Sets the value of `(int) save_restoreNumSeqFiles` (initially 3). This is the number of sequenced backup files to be maintained.

This function can be called at any time.

`void save_restoreSet_SeqPeriodInSeconds(int period)`
Sets the value of `(int) save_restoreSeqPeriodInSeconds` (initially 60). Sequenced backup files will be written with this period.

This function can be called at any time.

`void save_restoreSet_DatedBackupFiles(int ok)`
Sets the value of `(int) save_restoreDatedBackupFiles` (initially 1). If zero, the backup file written at reboot time (a copy of the file from which parameter values are restored) will have the suffix '.bu', and will be overwritten every reboot. If nonzero, each reboot will leave behind its own backup file.

This function can be called at any time.

`void save_restoreSet_status_prefix(char *prefix)`
Specifies the prefix to be used to construct the names of PV's with which save_restore reports its status.

This function must be called before the first call to `create_xxx_set()`.

`void save_restoreSet_NFSHost(char *hostname, char *address)`
Specifies the name and IP address of the NFS host. If both have been specified, and `set_savefile_path()` has been called to specify the file path, save_restore will manage its own NFS mount. This allows save_restore to recover from a reboot of the NFS host (i.e., a stale file handle) and from some kinds of tampering with the save_restore directory.

## Example of use

---------- begin excerpt from st.cmd ----------------------

```
.
.
.
### autoSaveRestore setup
save_restoreSet_Debug(0)

# status-PV prefix, so save_restore can find its status PV's.
save_restoreSet_status_prefix("xxx:")

# ok to restore a save set that had missing values (no CA connection to PV)?
# ok to save a file if some CA connections are bad?
```

```
save_restoreSet_IncompleteSetsOk(1)

# In the restore operation, a copy of the save file will be written.  The
# file name can look like "auto_settings.sav.bu", and be overwritten every
# reboot, or it can look like "auto_settings.sav_020306-083522" (this is what
# is meant by a dated backup file) and every reboot will write a new copy.
save_restoreSet_DatedBackupFiles(1)

# specify where save files should go
set_savefile_path(startup, "autosave");

## specify where request files can be found
# current directory
set_requestfile_path(startup, "")
# We want to include request files that are stored with the databases they
# support -- e.g., in stdApp/Db, mcaApp/Db, etc.  The variables std and mca
# are defined in cdCommands.  The path is searched in the order in which
# directories are specified.
set_requestfile_path(startup)
set_requestfile_path(std, "stdApp/Db")
set_requestfile_path(motor, "motorApp/Db")
set_requestfile_path(mca, "mcaApp/Db")
set_requestfile_path(ip, "ipApp/Db")
set_requestfile_path(ip330, "ip330App/Db")
# [...]

# specify what save files should be restored when
# up to eight files can be specified for each pass
set_pass0_restoreFile("auto_positions.sav")
set_pass0_restoreFile("auto_settings.sav")
set_pass1_restoreFile("auto_settings.sav")
# [...]

# Number of sequenced backup files (e.g., 'auto_settings.sav0') to write
save_restoreSet_NumSeqFiles(3)

# Time interval between sequenced backups
save_restoreSet_SeqPeriodInSeconds(600)

# NFS host name and IP address
save_restoreSet_NFSHost("oxygen", "164.54.52.4")
.
.
.
dbLoadDatabase("../../dbd/xxxApp.dbd")
.
.
.
dbLoadRecords("$(AUTOSAVE)/asApp/Db/save_restoreStatus.db", "P=xxx:")
.
.
.
iocInit
.
.
.
### Start up the save_restore task and tell it what to do.
# The task is actually named "save_restore".
#
```

```
# save positions every five seconds
create_monitor_set("auto_positions.req", 5, "P=xxx:")
# save other things every thirty seconds
create_monitor_set("auto_settings.req", 30, "P=xxx:")
.
.
.
```

---------- end excerpt from st.cmd ----------------------

*Suggestions and Comments to:*
*Tim Mooney : (mooney@aps.anl.gov)*
*Last modified: July 24, 2006*