

Alarm Scanning for PPC

Improved performance

Wed, Jun 25, 2008

The alarm scan logic used in IRM software checks every analog channel and every digital bit for possible alarm checking, even though most channels and bits don't require it. In IRMs, nonvolatile memory access time is not significantly different from dynamic RAM access time. But in the PPC systems, nonvolatile memory access is much slower than DRAM access. This note describes the improved scheme in alarm scanning for PPC nodes, although it was first implemented for 68K nodes. This note was originally written in 2001 as a plan. This version is an attempt to bring it up to date.

Background

As a reference point, the time for an IRM to perform a complete alarm scan, which it does every 15 Hz cycle, was about 1.0–1.5 ms. The time for PPC systems was slower, perhaps 1.5–2.0 ms, due to the slow access time to nonvolatile memory. Both of these times are for systems that allocate 1K channels and 1K bits. For the Linac upgrade, however, we doubled the number of channels allocated in some nodes, and we also doubled the number of bits. This would double the alarm scan time so that it would approach 4 ms. This is the time we tried to reduce. Whether this is deemed significant enough to matter, for a job that is performed once every 66 ms, is a separate consideration.

The plan

The basic idea takes advantage of the fact that the vast majority of channels and bits in any system are *not* enabled for alarm scanning, so that if we could merely check alarms for those channels and bits that *are* enabled for alarm scanning, the time required could be much less.

The plan is to occasionally perform a scan that builds up a list of the active channels and bits. The alarm scanning logic is modified to check only those channels and bits that need to be checked. All others are completely ignored.

How often is it worth reconstructing this alarm channel/bit list? Ideally, we only have to reconstruct the list when a change is made to the alarm flags such that the Active bit becomes set. Such changes are very infrequent in normal operation. If only listtype-directed processing sets an Active flag, and not an informal memory access, then special code can monitor whether reconstructing the list is needed and set a flag to alert the Alarms task its next time around. Turning off the Active bit should also result in an immediate change, because a channel might refer to what is now an inactive entry, and it would not check that entry, whereas it needs to see it one more time to decide whether to emit a "good" message. Still, it may be useful to occasionally reconstruct the list just to be safe. If the alarm flags word is modified in memory, which is very unusual, at least the list of active entries would not remain invalid forever.

For a setting that specifies the alarm flags listype or the nominal value listype with 6 bytes of setting data in tow, a special routine is called. The routine that is called to set the analog alarm flags is SETANOM. The routine required to set the binary alarm flags word is simply SETBFLG. (SETANOM calls SETAFLG.) So, the special code for monitoring when alarm flags are changed is placed in SETFLG, a routine that is called by both SETAFLG and SETBFLG.

Since the SETAFLG logic preserves the active bit for one more alarm scan in the case that it is to be turned off, the reconstruction operation logic should be placed at the end of the alarm scan, not at its beginning. If the bit is being turned off, the Active bit will be forced set, and another bit will be set that will cause the alarm scan logic to remove the Active bit, once it has had a chance to determine whether a "good" message should be posted. The new action by SETAFLG triggers a reconstruction of the fast alarm scan list at the conclusion of the next alarm scan.

The reconstruction effort is scheduled separately for the analog and digital cases. If only a change in an analog channel's Active bit is done, only the analog list reconstruction is required.

The case of the comment alarms is abbreviated. We have a default table size of 64 entries, whereas we

have never used more than 2 entries. A reconstruction scan is also done for these entries, triggered by changes in the comment alarm flags.

The set-type routines signal the `Alarms` task to do the reconstruction. They do this by sending a task event to the `Alarms` task. Three separate task events are used to cover the three basic cases of analog, binary, and comment.

Alarm scanning

Alarm scanning logic, then, sequences through the array of channel numbers, obtaining a pointer to the `ADATA` entry for that channel, and continues with the usual logic: check for the Active bit, and if set—which it will very likely be—branch to the usual code for processing, after which the code returns to the top of the loop to pick up the next entry in the list of active channel numbers.

Similar logic is used for the binary and comment cases, using the appropriate list of indexes for each.

After all alarm scanning is complete, check for the three additional task events that might be set to enable reconstruction of the appropriate array of table indexes. These events are not waited on, but rather watched for following the 15 Hz alarm scan.

List memory

Where is the memory found for holding the index arrays? We allocate a block of memory for each of the three cases for this purpose. Upon initialization of the `Alarms` task, three pointer variables are set to `NULL`. When processing each case, if the pointer is `NULL`, skip the alarm scan, but allocate the block instead and set the event bit. After the alarm scan logic, check for the appropriate event bit set. If the event was set, construct the array of indices according to which table entries have the Active bit set in the alarm flags field. During the next cycle, the valid pointer will enable the alarm scan to be performed. The point is that the list reconstruction logic is done as part of the `Alarms` task.

By using allocated memory blocks for the lists, we have to work a bit to find the addresses of the arrays being scanned for alarms. Find the `Alarms` task variables, and the 3 pointers should be there. At this writing, for a PPC node, look up the task Id for the `Alarms` task, in entry #1 of the Task Table based in low memory at `0x000700`. This is an address that points to a `VxWorks` task control block. Take the address found at the beginning of this block, subtract 200 bytes, and one should find the three list block pointer variables, for the analog, binary, and comment cases. As a check, following the third pointer should be a copy of the `BADDR` table entry for byte `0x14`, which corresponds to the alarm options byte that includes Bits `0x00A0–0x00A7`. At this writing, it is `0x4800FF14`.

For 68K nodes, the procedure is different. Take the task Id for the `Alarms` task, as above, but then add `0x46`, take the pointer found there and subtract 80 bytes. This is the address of the three list block pointers, which should be followed by `0x40FF14`.

Each of the three allocated active record blocks uses the same header structure that provides some simple diagnostics.

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
<code>MBlkSize</code>	2	Size of allocated block
<code>MBlkSpar</code>	4	(spare words)
<code>MBlkType</code>	2	Memory block type# =000A
<code>ActivT</code>	2	Time to perform reconstruction of active entry array in μ s
<code>ActivN</code>	2	Current #active entries
<code>ActivMn</code>	2	<code>ActivN</code> minimum
<code>ActivMx</code>	2	<code>ActivN</code> maximum
<code>ActivC</code>	4	Count of alarm-flag-change-initiated reconstructions
<code>ActivCB</code>	4	Count of periodic backup reconstructions

ActivDT	8	BCD Date / time of last non-backup reconstruction
Activa	2048	Array of active entries (assuming 1024 allocated entries)

The diagnostics show how many times a reconstruction takes place, especially when it is a result of changing an alarm flags Active bit. It also shows when such a change was last done. In any case, it includes the time for performing the reconstruction.

The initial implementation was for the 68040-based nodes. In a system that has 65 active analog channels, one active bit and 2 comments, the execution time of the `Alarms` task is now about 400 μ s.

The benefit in using this new logic is quite dramatic for the PowerPC case. Running at 233 MHz cpu clock, the access time to the non-cacheable nonvolatile memory is about 1 μ s. This was the motivation for considering the new logic, in which only the entries that need to be examined are actually checked for alarm conditions. See the timing results for the PPC later in this note.

One may have considered a different approach, that of copying the `ADATA` table into ordinary DRAM, where the access time is very fast. But keeping these two sets of entries in sync would have been difficult. An alarm scan can cause several bits in an alarm flags word to change. If the system dies, we need the nonvolatility of the entire table, including the alarm flags, settings, nominal and tolerances. The approach followed seems less risky, where the actual nonvolatile entries are still scanned for alarms. The performance improvement arises from skipping all the entries whose Active bits are off.

More details

Each change in an alarm flags word uses a special set-type routine invoked via the relevant listype. This routine determines whether the alarm flags Active bit is being changed. If it is, it signals the `Alarms` task via the appropriate task event. Although the `Alarms` task is not actually waiting on these events, it will detect them after its next alarm scan. This is done because a user who tries to turn off the Active bit will cause the "bypass control" bit to be set and the Active bit to remain set, so the next alarm scan can detect this case, and if necessary, emit a "good" alarm message.

In addition to a reconstruction resulting from changing an Active bit, it is forced to occur periodically, just as a precaution. The period as implemented is about 9 minutes, or 8192/15 seconds at 15 Hz. (This is done by monitoring the low 13 bits of the global cycle counter.) All 3 reconstructions take place periodically, but they do not occur on the same 15 Hz cycle.

The modifications to the `ASCAN`, `BSCAN`, and `CSCAN` routines in the `Alarms` task are made as much alike as possible. Two new routines were added; the `ACTBLOCK` routine allocates an active index block, and the `ACTBUILD` routine reconstructs the block.

Minimize accesses to nonvolatile memory

Since the nonvolatile access time is so slow, we looked at a means of limiting the required accesses to improve the time for the actual alarm checking. This is especially true for the `ASCAN` routine that accesses entries in the `ADATA` table.

Fields of an `ADATA` table entry that are accessed for alarm scanning are these:

- Reading
- Nominal
- Tolerance
- Flags
- Count

The `Flags` field must be looked at each time. The `Count` field only needs to be accessed when alarm changes are detected. The `Reading`, `Nominal` and `Tolerance` fields need to be checked each time, of course. We may modify the `Flags` and/or `Count` words, but none of the other fields. We can read

the `Reading` word separately, then read the `Nominal` and `Tolerance` words using a single long word access. This will save the time needed for one access every time. We can also read the `Flags` and `Count` words as a single long integer, also saving one access some of the time. (It may not be necessary to examine the `Count` word in many cases.) We may have to modify one or both of them, but often, we only have to read it.

Examples from the previous code:

Near the end of the `ASCAN` loop, there was a place where the `#times` nibble, which is part of the `Count` field, is to be reset. If we have already made a copy of this `Count` field, we should first see whether the nibble is already set, before updating the `Count` word. Also, with a copy of that field already handy, we do not have to both read and write it to clear the nibble, but can rather write the updated `Count` value with the hi nibble cleared. (We already know the value of the low 12 bits.)

New viewpoint:

Copy out the `Flags` and `Count` words as a long word, then separate them into separate local variables for the flags word and the count word. Any place in the code used in checking for alarms where this information needs to be accessed or modified, do it to these local copies. At the end of the loop, check whether any change has been made to any of the potentially-alterable fields. If the final value of the local flags word is different from the original value, or if the count value has changed, perform the appropriate long integer write to update both the `Flags` and `Count` fields.

Delay accessing the `Reading`, `Nominal` and `Tolerance` fields until we know that these fields need to be referenced, based upon the floating point flag bit in `Flags`. Only then do we know whether the `ADATA` table needs to be accessed for these fields, or whether it is instead the `FDATA` table that should be accessed. Both tables are in nonvolatile memory. For `ADATA`, one can access the `Nominal` and `Tolerance` words as a single long word. For `FDATA`, two 32-bit accesses are required to get the two floating point fields.

A simple example can illustrate the potential savings of time. For the integer case of a reading that is currently in the "good" alarm state and remains so, there were 7 accesses to nonvolatile memory. In the new way, there might be only 3, the `Flags/Count` long word, the `Reading` word, and the `Nominal/Tolerance` long word. In this simple case, neither the `Flags` word nor the `Count` word needs to change. In cases where the tries nibble needs to be changed, say, only one more access is required.

In some of the Linac nodes, there may be 150 readings checked, including both analog and binary. Assuming that access to nonvolatile memory costs 1 μ s, then the time required for 7 accesses each would cost 1 ms, whereas with the new way, less than half that time does the job.

With the code modified according to this scheme, and if later on, a faster form of nonvolatile memory were found, the code would still run faster, maybe almost 3 μ s faster than before. A complete alarm scan done in well under 1 ms is ok for a 66 ms cycle time.

Again, the time required in a PPC node with 45 signals was about 2.2 ms. This is reducible to well under 0.5 ms with the changes outlined here. As an example, in `node0610`, with 72 active channels and 81 active bits, the `Alarms` task completes its scan in about 150 μ s. In `node0611`, with 91 active channels and 58 active bits, the alarm scanning time is also about 150 μ s. In `node061C`, with 15 active channels and 19 active bits, the scan time is only 60 μ s.

Lesson

Knowing that each nonvolatile memory reference might require 200 cpu wait cycles can have a major impact on how the code should be best organized. Local variable accesses, since they are in faster cacheable dynamic ram, almost don't count. The time needed for intensive use of nonvolatile tables depends almost entirely on the number of accesses made to nonvolatile memory. An algorithm that minimizes the number of such accesses will win.