```
**********************************************************
   74605 Fri Apr 11 14:23:38 2014
new/usr/src/uts/common/os/clock.c
4748 use an enum for tod_faulted global
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
/*        All Rights Reserved   */

/*
 * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc.  All rights reserved.
 */

#include <sys/param.h>
#include <sys/t_lock.h>
#include <sys/types.h>
#include <sys/tuneable.h>
#include <sys/sysmacros.h>
#include <sys/systm.h>
#include <sys/cpuvar.h>
#include <sys/lgrp.h>
#include <sys/user.h>
#include <sys/proc.h>
#include <sys/callo.h>
#include <sys/kmem.h>
#include <sys/var.h>
#include <sys/cmn_err.h>
#include <sys/swap.h>
#include <sys/vmsystm.h>
#include <sys/class.h>
#include <sys/time.h>
#include <sys/debug.h>
#include <sys/vtrace.h>
#include <sys/spl.h>
#include <sys/atomic.h>
#include <sys/dumphdr.h>
#include <sys/archsystm.h>
#include <sys/fs/swapnode.h>
#include <sys/panic.h>
#include <sys/disp.h>
#include <sys/msacct.h>
#include <sys/mem_cage.h>

#include <vm/page.h>
#include <vm/anon.h>
#include <vm/rm.h>
```



```
#include <sys/cyclic.h>
#include <sys/cpupart.h>
#include <sys/rctl.h>
#include <sys/task.h>
#include <sys/sdt.h>
#include <sys/ddi_periodic.h>
#include <sys/random.h>
#include <sys/modctl.h>

/*
 * for NTP support
 */
#include <sys/timex.h>
#include <sys/inttypes.h>

#include <sys/sunddi.h>
#include <sys/clock_impl.h>

/*
 * clock() is called straight from the clock cyclic; see clock_init().
 *
 * Functions:
 *      reprime clock
 *      maintain date
 *      jab the scheduler
 */

extern kcondvar_t       fsflush_cv;
extern sysinfo_t        sysinfo;
extern vminfo_t vminfo;
extern int      idleswtch;      /* flag set while idle in pswtch() */
extern hrtime_t volatile devinfo_freeze;

/*
 * high-precision avenrun values.  These are needed to make the
 * regular avenrun values accurate.
 */
static uint64_t hp_avenrun[3];
int     avenrun[3];             /* FSCALED average run queue lengths */
time_t  time;   /* time in seconds since 1970 - for compatibility only */

static struct loadavg_s loadavg;
/*
 * Phase/frequency-lock loop (PLL/FLL) definitions
 *
 * The following variables are read and set by the ntp_adjtime() system
 * call.
 *
 * time_state shows the state of the system clock, with values defined
 * in the timex.h header file.
 *
 * time_status shows the status of the system clock, with bits defined
 * in the timex.h header file.
 *
 * time_offset is used by the PLL/FLL to adjust the system time in small
 * increments.
 *
 * time_constant determines the bandwidth or "stiffness" of the PLL.
 *
 * time_tolerance determines maximum frequency error or tolerance of the
 * CPU clock oscillator and is a property of the architecture; however,
 * in principle it could change as result of the presence of external
 * discipline signals, for instance.
 *
 * time_precision is usually equal to the kernel tick variable; however,
 * in cases where a precision clock counter or external clock is
```

```
 * available, the resolution can be much less than this and depend on
 * whether the external clock is working or not.
 *
 * time_maxerror is initialized by a ntp_adjtime() call and increased by
 * the kernel once each second to reflect the maximum error bound
 * growth.
 *
 * time_esterror is set and read by the ntp_adjtime() call, but
 * otherwise not used by the kernel.
 */
int32_t time_state = TIME_OK;   /* clock state */
int32_t time_status = STA_UNSYNC;       /* clock status bits */
int32_t time_offset = 0;                /* time offset (us) */
int32_t time_constant = 0;              /* pll time constant */
int32_t time_tolerance = MAXFREQ;       /* frequency tolerance (scaled ppm) */
int32_t time_precision = 1;     /* clock precision (us) */
int32_t time_maxerror = MAXPHASE;       /* maximum error (us) */
int32_t time_esterror = MAXPHASE;       /* estimated error (us) */

/*
 * The following variables establish the state of the PLL/FLL and the
 * residual time and frequency offset of the local clock. The scale
 * factors are defined in the timex.h header file.
 *
 * time_phase and time_freq are the phase increment and the frequency
 * increment, respectively, of the kernel time variable.
 *
 * time_freq is set via ntp_adjtime() from a value stored in a file when
 * the synchronization daemon is first started. Its value is retrieved
 * via ntp_adjtime() and written to the file about once per hour by the
 * daemon.
 *
 * time_adj is the adjustment added to the value of tick at each timer
 * interrupt and is recomputed from time_phase and time_freq at each
 * seconds rollover.
 *
 * time_reftime is the second's portion of the system time at the last
 * call to ntp_adjtime(). It is used to adjust the time_freq variable
 * and to increase the time_maxerror as the time since last update
 * increases.
 */
int32_t time_phase = 0;         /* phase offset (scaled us) */
int32_t time_freq = 0;          /* frequency offset (scaled ppm) */
int32_t time_adj = 0;           /* tick adjust (scaled 1 / hz) */
int32_t time_reftime = 0;               /* time at last adjustment (s) */

/*
 * The scale factors of the following variables are defined in the
 * timex.h header file.
 *
 * pps_time contains the time at each calibration interval, as read by
 * microtime(). pps_count counts the seconds of the calibration
 * interval, the duration of which is nominally pps_shift in powers of
 * two.
 *
 * pps_offset is the time offset produced by the time median filter
 * pps_tf[], while pps_jitter is the dispersion (jitter) measured by
 * this filter.
 *
 * pps_freq is the frequency offset produced by the frequency median
 * filter pps_ff[], while pps_stabil is the dispersion (wander) measured
 * by this filter.
 *
 * pps_usec is latched from a high resolution counter or external clock
 * at pps_time. Here we want the hardware counter contents only, not the
 * contents plus the time_tv.usec as usual.
```



```
 *
 * pps_valid counts the number of seconds since the last PPS update. It
 * is used as a watchdog timer to disable the PPS discipline should the
 * PPS signal be lost.
 *
 * pps_glitch counts the number of seconds since the beginning of an
 * offset burst more than tick/2 from current nominal offset. It is used
 * mainly to suppress error bursts due to priority conflicts between the
 * PPS interrupt and timer interrupt.
 *
 * pps_intcnt counts the calibration intervals for use in the interval-
 * adaptation algorithm. It's just too complicated for words.
 */
struct timeval pps_time;        /* kernel time at last interval */
int32_t pps_tf[] = {0, 0, 0};   /* pps time offset median filter (us) */
int32_t pps_offset = 0;         /* pps time offset (us) */
int32_t pps_jitter = MAXTIME;   /* time dispersion (jitter) (us) */
int32_t pps_ff[] = {0, 0, 0};   /* pps frequency offset median filter */
int32_t pps_freq = 0;           /* frequency offset (scaled ppm) */
int32_t pps_stabil = MAXFREQ;   /* frequency dispersion (scaled ppm) */
int32_t pps_usec = 0;           /* microsec counter at last interval */
int32_t pps_valid = PPS_VALID;  /* pps signal watchdog counter */
int32_t pps_glitch = 0;         /* pps signal glitch counter */
int32_t pps_count = 0;          /* calibration interval counter (s) */
int32_t pps_shift = PPS_SHIFT;  /* interval duration (s) (shift) */
int32_t pps_intcnt = 0;         /* intervals at current duration */

/*
 * PPS signal quality monitors
 *
 * pps_jitcnt counts the seconds that have been discarded because the
 * jitter measured by the time median filter exceeds the limit MAXTIME
 * (100 us).
 *
 * pps_calcnt counts the frequency calibration intervals, which are
 * variable from 4 s to 256 s.
 *
 * pps_errcnt counts the calibration intervals which have been discarded
 * because the wander exceeds the limit MAXFREQ (100 ppm) or where the
 * calibration interval jitter exceeds two ticks.
 *
 * pps_stbcnt counts the calibration intervals that have been discarded
 * because the frequency wander exceeds the limit MAXFREQ / 4 (25 us).
 */
int32_t pps_jitcnt = 0;         /* jitter limit exceeded */
int32_t pps_calcnt = 0;         /* calibration intervals */
int32_t pps_errcnt = 0;         /* calibration errors */
int32_t pps_stbcnt = 0;         /* stability limit exceeded */

kcondvar_t lbolt_cv;

/*
 * Hybrid lbolt implementation:
 *
 * The service historically provided by the lbolt and lbolt64 variables has
 * been replaced by the ddi_get_lbolt() and ddi_get_lbolt64() routines, and the
 * original symbols removed from the system. The once clock driven variables are
 * now implemented in an event driven fashion, backed by gethrtime() coarsed to
 * the appropriate clock resolution. The default event driven implementation is
 * complemented by a cyclic driven one, active only during periods of intense
 * activity around the DDI lbolt routines, when a lbolt specific cyclic is
 * reprogramed to fire at a clock tick interval to serve consumers of lbolt who
 * rely on the original low cost of consulting a memory position.
 *
 * The implementation uses the number of calls to these routines and the
 * frequency of these to determine when to transition from event to cyclic
```

```
 260  * driven and vice-versa. These values are kept on a per CPU basis for
 261  * scalability reasons and to prevent CPUs from constantly invalidating a single
 262  * cache line when modifying a global variable. The transition from event to
 263  * cyclic mode happens once the thresholds are crossed, and activity on any CPU
 264  * can cause such transition.
 265  *
 266  * The lbolt_hybrid function pointer is called by ddi_get_lbolt() and
 267  * ddi_get_lbolt64(), and will point to lbolt_event_driven() or
 268  * lbolt_cyclic_driven() according to the current mode. When the thresholds
 269  * are exceeded, lbolt_event_driven() will reprogram the lbolt cyclic to
 270  * fire at a nsec_per_tick interval and increment an internal variable at
 271  * each firing. lbolt_hybrid will then point to lbolt_cyclic_driven(), which
 272  * will simply return the value of such variable. lbolt_cyclic() will attempt
 273  * to shut itself off at each threshold interval (sampling period for calls
 274  * to the DDI lbolt routines), and return to the event driven mode, but will
 275  * be prevented from doing so if lbolt_cyclic_driven() is being heavily used.
 276  *
 277  * lbolt_bootstrap is used during boot to serve lbolt consumers who don't wait
 278  * for the cyclic subsystem to be intialized.
 279  *
 280  */
 281 int64_t lbolt_bootstrap(void);
 282 int64_t lbolt_event_driven(void);
 283 int64_t lbolt_cyclic_driven(void);
 284 int64_t (*lbolt_hybrid)(void) = lbolt_bootstrap;
 285 uint_t lbolt_ev_to_cyclic(caddr_t, caddr_t);

 287 /*
 288  * lbolt's cyclic, installed by clock_init().
 289  */
 290 static void lbolt_cyclic(void);

 292 /*
 293  * Tunable to keep lbolt in cyclic driven mode. This will prevent the system
 294  * from switching back to event driven, once it reaches cyclic mode.
 295  */
 296 static boolean_t lbolt_cyc_only = B_FALSE;

 298 /*
 299  * Cache aligned, per CPU structure with lbolt usage statistics.
 300  */
 301 static lbolt_cpu_t *lb_cpu;

 303 /*
 304  * Single, cache aligned, structure with all the information required by
 305  * the lbolt implementation.
 306  */
 307 lbolt_info_t *lb_info;


 310 int one_sec = 1; /* turned on once every second */
 311 static int fsflushcnt;  /* counter for t_fsflushr */
 312 int     dosynctodr = 1; /* patchable; enable/disable sync to TOD chip */
 313 int     tod_needsync = 0;       /* need to sync tod chip with software time */
 314 static int tod_broken = 0;      /* clock chip doesn't work */
 315 time_t  boot_time = 0;          /* Boot time in seconds since 1970 */
 316 cyclic_id_t clock_cyclic;       /* clock()'s cyclic_id */
 317 cyclic_id_t deadman_cyclic;     /* deadman()'s cyclic_id */

 319 extern void     clock_tick_schedule(int);

 321 static int lgrp_ticks;          /* counter to schedule lgrp load calcs */

 323 /*
 324  * for tod fault detection
 325  */
```



```
 326 #define TOD_REF_FREQ            ((longlong_t)(NANOSEC))
 327 #define TOD_STALL_THRESHOLD     (TOD_REF_FREQ * 3 / 2)
 328 #define TOD_JUMP_THRESHOLD      (TOD_REF_FREQ / 2)
 329 #define TOD_FILTER_N            4
 330 #define TOD_FILTER_SETTLE       (4 * TOD_FILTER_N)
 331 static enum tod_fault_type tod_faulted = TOD_NOFAULT;
 331 static int tod_faulted = TOD_NOFAULT;

 333 static int tod_status_flag = 0;         /* used by tod_validate() */

 335 static hrtime_t prev_set_tick = 0;      /* gethrtime() prior to tod_set() */
 336 static time_t prev_set_tod = 0;         /* tv_sec value passed to tod_set() */

 338 /* patchable via /etc/system */
 339 int tod_validate_enable = 1;

 341 /* Diagnose/Limit messages about delay(9F) called from interrupt context */
 342 int                     delay_from_interrupt_diagnose = 0;
 343 volatile uint32_t       delay_from_interrupt_msg = 20;

 345 /*
 346  * On non-SPARC systems, TOD validation must be deferred until gethrtime
 347  * returns non-zero values (after mach_clkinit's execution).
 348  * On SPARC systems, it must be deferred until after hrtime_base
 349  * and hres_last_tick are set (in the first invocation of hres_tick).
 350  * Since in both cases the prerequisites occur before the invocation of
 351  * tod_get() in clock(), the deferment is lifted there.
 352  */
 353 static boolean_t tod_validate_deferred = B_TRUE;

 355 /*
 356  * tod_fault_table[] must be aligned with
 357  * enum tod_fault_type in systm.h
 358  */
 359 static char *tod_fault_table[] = {
 360         "Reversed",                     /* TOD_REVERSED */
 361         "Stalled",                      /* TOD_STALLED */
 362         "Jumped",                       /* TOD_JUMPED */
 363         "Changed in Clock Rate",        /* TOD_RATECHANGED */
 364         "Is Read-Only"                  /* TOD_RDONLY */
 365         /*
 366          * no strings needed for TOD_NOFAULT
 367          */
 368 };
______unchanged_portion_omitted_
```