Appendix C. Phase-Lock Loop Simulator

In order to more fully understand the workings of the kernel phase-lock loop (PLL), a stand-alone simulator kern.c is included in the kernel distributions. This is an implementation of an adaptive-parameter, first-order, type-II phase-lock loop. The local clock is implemented using a set of variables and algorithms defined in the simulator and driven by explicit offsets generated by the simulator. The algorithms include code fragments identical to those in the modified kernel routines and operate in the same way, but the operations can be understood separately from any licensed source code into which these fragments may be integrated. The code segments themselves are not derived from any licensed code.

In the simulator the hardupdate() fragment is called by the ntp_adjtime() system call as each update is determined using the synchronization protocol. Note that the time constant is in units of powers of two, so that multiplies can be done by simple shifts. The phase variable is computed as the offset multiplied by the time constant. Then, the time since the last update is computed and clamped to a maximum (for robustness) and to zero if initializing. The offset is multiplied (sorry about the ugly multiply) by the result and by the square of the time constant and then added to the frequency variable. Finally, the frequency variable is clamped not to exceed the tolerance. Note that all shifts are assumed to be positive and that a shift of a signed quantity to the right requires a little dance.

With the defines given, the maximum time offset is determined by the size in bits of the long type (32) less the SHIFT_UPDATE scale factor or 18 bits (signed). The scale factor is chosen so that there is no loss of significance in later steps, which may involve a right shift up to 14 bits. This results in a maximum offset of about ±130 ms. Since time_constant must be greater than or equal to zero, the maximum frequency offset is determined by the SHIFT_KF (20) scale factor, or about ±130 ppm. In the addition step, the product of the offset and the time interval is represented in 18 + 10 = 28 bits, which will not overflow a 32-bit long add. There could be a loss of precision due to the right shift of up to 8 bits, since time_constant is bounded at 6. This results in a net worst-case frequency error of about 2^{16} μs or well down into the system phase noise. While the time_offset value is assumed checked before entry, the time_phase variable is an accumulator, so is clamped to the tolerance on every call. This helps to damp transients before the oscillator frequency has been determined, as well as to satisfy the correctness assertions if the time-synchronization protocol comes unstuck.

The hardclock() fragment is inserted in the hardware timer interrupt routine at the point the local clock is to be incremented. Previous to this fragment the time_update variable has been initialized to the value computed by the adjtime() system call in the stock Unix kernel, normally the value of *tick* plus/minus the *tickadj* value, which is usually in the order of 5 μs. When the kernel PLL is in use, as indicated by calls on ntp_adjtime(), adjtime() is normally not in use, so the time_update value at this point is the value of tick. This value, the phase adjustment (time_adj) and the clock phase (time_phase) are summed and the total tested for overflow of the microsecond. If an overflow occurs, the microsecond (tick) is incremented or decremented, depending on the sign of the overflow.

The second_overflow() fragment is inserted at the point following the hardclock() fragment where the microseconds field of the system time variable is being checked for overflow of the second. On rollover the maximum error is increased by the tolerance and the time offset is divided by the phase weight (SHIFT_KG) and time constant. The time offset is then reduced by the result and the result is scaled and becomes the value of the phase adjustment. The phase adjustment is then corrected for the calculated frequency offset and a fixed offset determined from the *fixtick* variable in some kernel implementations. On rollover of the day, the leap-warning indicator is checked and the apparent time adjusted ±1 s accordingly. The microtime() routine insures that the reported time is always monotonically increasing.

The simulator has been used to check the PLL operation over the design envelope of ±128 ms in time error and ±100 ppm in frequency error. This confirms that no overflows or significant roundoff errors occur and that the loop initially converges in about 15 minutes for timer interrupt rates from 50 Hz to 1024 Hz. The loop has a normal overshoot of about seven percent and a final convergence time of several hours, depending on the initial time and frequency error.

```
/*
 * This program simulates a first-order, type-II phase-lock loop using
 * actual code segments from modified kernel distributions for SunOS,
 * Ultrix and OSF/1 kernels. These segments do not use any licensed
 * code.
 */

#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include <sys/time.h>

#include "timex.h"

/*
 * Phase-lock loop definitions
 */
#define HZ 100                        /* timer interrupt frequency (Hz) */
#define MAXPHASE 128000               /* max phase error (us) */
#define MAXFREQ 100                   /* max frequency error (ppm) */
#define MINSEC 16                     /* min interval between updates (s) */
#define MAXSEC 1200                   /* max interval between updates (s) */

/*
 * Function declarations
 */
void hardupdate();
void hardclock();
void second_overflow();

/*
 * Kernel variables
 */
int tick;                             /* timer interrupt period (us) */
int fixtick;                          /* amortization constant (ppm) */
struct timeval timex;                 /* ripoff of kernel time variable */

/*
 * Phase-lock loop variables
 */
int time_status = TIME_BAD;             /* clock synchronization status */
long time_offset = 0;                   /* time adjustment (us) */
long time_constant = 0;                 /* pll time constant */
long time_tolerance = MAXFREQ;          /* frequency tolerance (ppm) */
long time_precision = 1000000 / HZ;     /* clock precision (us) */
long time_maxerror = MAXPHASE;          /* maximum error (us) */
long time_esterror = MAXPHASE;          /* estimated error (us) */
long time_phase = 0;                    /* phase offset (scaled us) */
long time_freq = 0;                     /* frequency offset (scaled ppm) */
long time_adj = 0;                      /* tick adjust (scaled 1 / HZ) */
long time_reftime = 0;                  /* time at last adjustment (s) */

/*
 * Simulation variables
 */
double timey = 0;                       /* simulation time (us) */
long timez = 0;                         /* current error (us) */
long poll_interval = 0;                 /* poll counter */

/*
 * Simulation test program
 */
void main()
{
	double ffreq, fdenom;

	tick = 1000000 / HZ;
	fixtick = 1000000 % HZ;
	timex.tv_sec = 0;
	timex.tv_usec = MAXPHASE;
	time_freq = MAXFREQ << SHIFT_KF;
	time_constant = 0;
	fdenom = 1 << SHIFT_KF;
	ffreq = (double)time_freq / fdenom;
	printf("tick %d us, fixtick %d us\n", tick, fixtick);
	printf("     time    offset     freq   _offset     _freq      _adj\n");

	/*
	 * Grind the loop until ^C
	 */
	while (1) {
		timey += (double)1000000 / HZ;
		if (timey >= 1000000)
			timey -= 1000000;
		hardclock();
		if (timex.tv_usec >= 1000000) {
			timex.tv_usec -= 1000000;
```

```
                        timex.tv_sec++;
                        second_overflow();
                        poll_interval++;
                        if (poll_interval >= (1 << (time_constant + 4))) {
                                poll_interval -= 1 << (time_constant + 4);
                                timez = (long)timey - timex.tv_usec;
                                if (timez > 500000)
                                        timez -= 1000000;
                                if (timez < -500000)
                                        timez += 1000000;
                                hardupdate(timez);
                                printf("%10li%10li%10.2f  %08lx  %08lx  %08lx\n",
                                   timex.tv_sec, timez,
                                   (double)time_freq / fdenom,
                                   time_offset, time_freq, time_adj);
                        }
                }
        }
}

/*
 * This routine simulates the ntp_adjtime() call
 */
void hardupdate(offset)
long offset;
{
        long ltemp, mtemp;

        time_offset = offset << SHIFT_UPDATE;
        mtemp = timex.tv_sec - time_reftime;
        time_reftime = timex.tv_sec;
        if (mtemp > MAXSEC)
                mtemp = 0;

        /* ugly multiply should be replaced */
        if (offset < 0)
                time_freq -= (-offset * mtemp) >> (time_constant + time_constant);
        else
                time_freq += (offset * mtemp) >> (time_constant + time_constant);
        ltemp = time_tolerance << SHIFT_KF;
        if (time_freq > ltemp)
                time_freq = ltemp;
        else if (time_freq < -ltemp)
                time_freq = -ltemp;
        if (time_status == TIME_BAD)
                time_status = TIME_OK;
}
```

```
/*
 * This routine simulates the timer interrupt
 */
void hardclock()
{
        int ltemp, time_update;

        time_update = tick;             /* computed by adjtime() */
        time_phase += time_adj;
        if (time_phase < -FINEUSEC) {
                ltemp = -time_phase >> SHIFT_SCALE;
                time_phase += ltemp << SHIFT_SCALE;
                time_update -= ltemp;
        }
        else if (time_phase > FINEUSEC) {
                ltemp = time_phase >> SHIFT_SCALE;
                time_phase -= ltemp << SHIFT_SCALE;
                time_update += ltemp;
        }
        timex.tv_usec += time_update;
}

/*
 * This routine simulates the overflow of the microsecond field
 */
void second_overflow()
{
        int ltemp;

        time_maxerror += time_tolerance;
        if (time_offset < 0) {
                ltemp = -time_offset >> (SHIFT_KG + time_constant);
                time_offset += ltemp;
                time_adj = -(ltemp << (SHIFT_SCALE - SHIFT_HZ - SHIFT_UPDATE));
        } else {
                ltemp = time_offset >> (SHIFT_KG + time_constant);
                time_offset -= ltemp;
                time_adj = ltemp << (SHIFT_SCALE - SHIFT_HZ - SHIFT_UPDATE);
        }
        time_adj += (time_freq >> (SHIFT_KF + SHIFT_HZ - SHIFT_SCALE))
           + (fixtick << (SHIFT_SCALE - SHIFT_HZ));

        /* ugly divide should be replaced */
        if (timex.tv_sec % 86400 == 0) {
                switch (time_status) {

                        case TIME_INS:
                        timex.tv_sec--;         /* !! */
                        time_status = TIME_OOP;
                        break;

                        case TIME_DEL:
                        timex.tv_sec++;
                        time_status = TIME_OK;
                        break;

                        case TIME_OOP:
                        time_status = TIME_OK;
                        break;
                }
        }
}
```